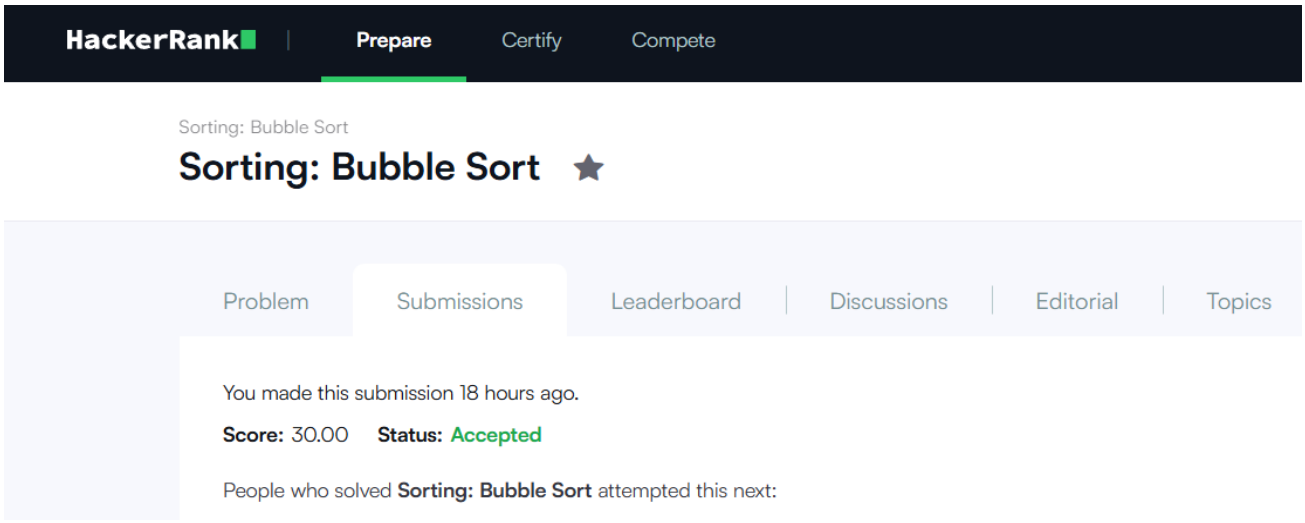
1)Bubble Sort Algorithm:



C++ code:

```cpp
// Function to perform Bubble Sort and return number of swaps
void bubble_sort(int array[], int size)
{
  int i, j, temp, swap = 0;
  for (i = 0; i < size - 1; i++)         // Outer loop runs (size-1) times
  {
    for (j = 0; j < size - i - 1; j++)   // Inner loop compares adjacent elements
    {
      if (array[j] > array[j + 1])       // If current element is greater than next element, swap them
      {
        temp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = temp;
        swap++;                          // Count each swap made
      }
    }
  }
  cout << "Array is sorted in " <<swap << " swaps.\n";
  cout << "First Element: " << array[0] << endl;
  cout << "Last Element: " << array[size - 1] << endl;

}
```

Bubble Sort is a simple sorting algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order. It continues passing through the array until no swaps are needed, meaning the array is sorted. With each pass, the largest unsorted element **"bubbles"** to its correct position at the end. **The time complexity is $O(n^2)$** and **the space complexity is O(1).**

Why this algorithm solves the problem efficiently?

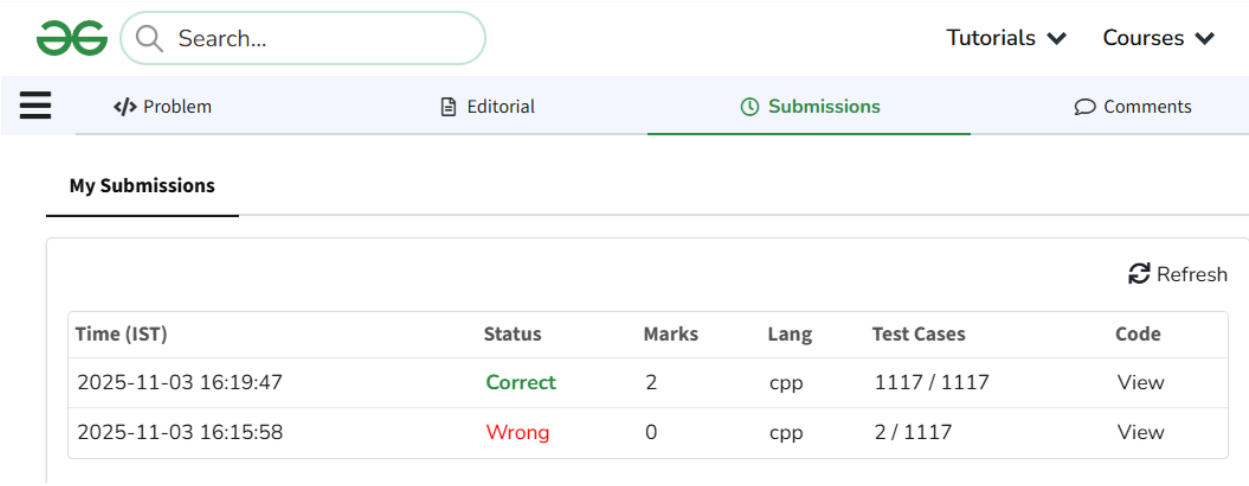
Simple and Easy to Implement → Ideal for learning and small datasets.

In-place Sorting → Requires no extra memory (O(1) space).

Adaptive with Optimization → Can stop early if the array is already sorted (best case O(n)).

Predictable Behavior → Always compares adjacent elements systematically until the array is sorted.

2)Binary Search



Time (IST)	Status	Marks	Lang	Test Cases	Code
2025-11-03 16:19:47	Correct	2	cpp	1117 / 1117	View
2025-11-03 16:15:58	Wrong	0	cpp	2 / 1117	View

C++ code:

```cpp
class Solution {
  public:
    int binarysearch(vector<int> &arr, int k) {
    int l = 0, r = arr.size() - 1, m;  //left=0,right=array size

    while(l <= r)                      // Binary search loop
    {
        m = (l + r) / 2;              // middle index
        if(arr[m] < k) l = m + 1;     // search right half
        else r = m - 1;               // search left half (including m)
    }

    // After loop, l is the position where k might exist
    return (l < arr.size() && arr[l] == k) ? l : -1; // if found return index, else -1
}
};
```

Binary Search is an efficient algorithm to find an element in a **sorted array** by repeatedly dividing the search interval in half. It compares the target with the middle element; if equal, it returns the index. If the target is smaller, it searches the left half; if larger, the right half. This process continues until the element is found or the interval is empty. The **time complexity is O(log n)** and **the space complexity is O(1)**.

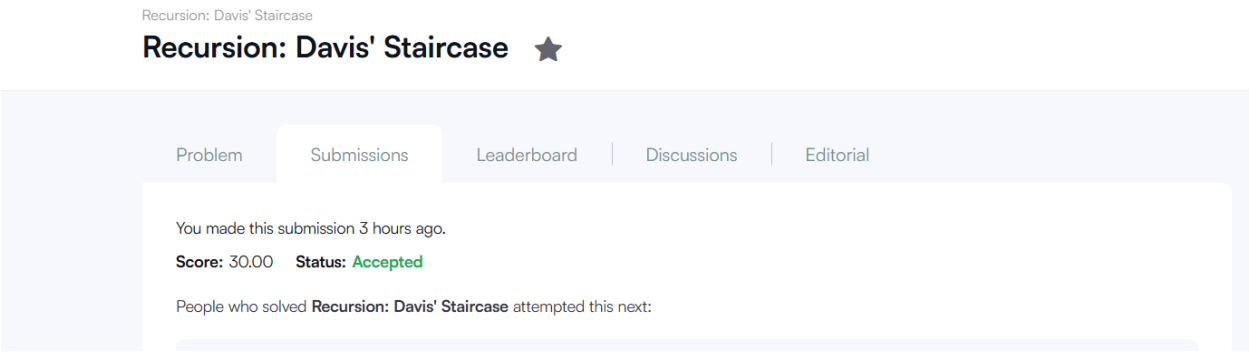
Why this algorithm solves the problem efficiently ?

Efficient Time Complexity → O(log n) compared to O(n) for linear search.

Predictable and Systematic → Always compares the middle element to decide the next search range.

Minimal Extra Space → Can be implemented iteratively with O(1) space.

3)Tower of Hanoi



C++ code:

```cpp
const ll mod=1e10+7;            // modulus value
ll dp[50];                      // memoization/dp array

ll fnc(ll n){
    if(n<0)return 0;            // negative index gives 0
    if(n==0)return 1;           // base case: only 1 way for n = 0
    if(dp[n]!=0)return dp[n];   // return stored result if already computed

    // recursive relation
    return dp[n]=fnc(n-1)%mod+fnc(n-2)%mod+fnc(n-3)%mod;
}
int main()
{
    int t;
    cin >> t;                   // number of test cases
    while (t--)
    {
        memset(dp,0,sizeof(dp)); // reset dp for each test case
        ll n;
        cin>>n;                 // input n
        n%=mod;                 // reduce n by mod
        cout<<fnc(n)%mod<<endl; // print answer
    }
}
```

This algorithm uses recursion with **memoization (dp[])** to compute a sequence where each term is the sum of the previous three terms. It checks if a result is already stored in dp[] to avoid recalculating it. For each test case, it resets the dp array and calls **fnc(n)** to compute the result modulo **1e10+7**. The recursion breaks when n < 0 or n == 0. The **time complexity is O(n)** (due to memoization), and the **space complexity is O(n)** .

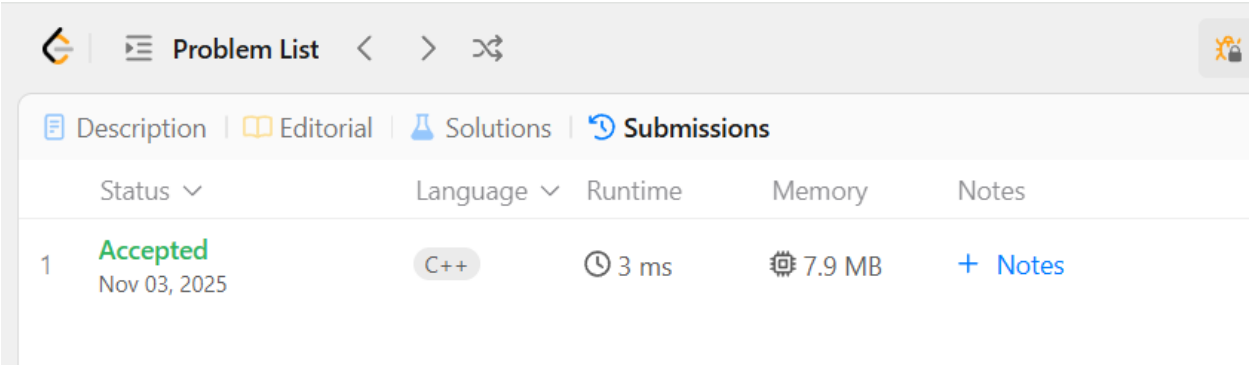
Why this algorithm solves the problem efficiently ?

Recursive Problem Solving → Breaks the problem into smaller subproblems by moving n-1 disks.

Systematic and Predictable → Always follows the rule: move smaller disks first.

Demonstrates Recursion → Excellent example to understand recursive algorithms.

4)Fibonacci Series



C++ code:

```cpp
class Solution {
public:
    int fib(int n) {
        long long f[33];        // array to store Fibonacci numbers up to index 32
        f[0]=0;                 // base case: Fib(0) = 0
        f[1]=1;                 // base case: Fib(1) = 1
        for(int i=2;i<=n;i++)  // fill Fibonacci values up to Fib(n)
        {
            f[i]=f[i-1]+f[i-2]; // recurrence: Fib(n) = Fib(n-1) + Fib(n-2)
        }
        return f[n];            // return nth Fibonacci number
    }
};
```

This C++ program calculates the **nth Fibonacci number** using a **bottom-up approach**. It initializes an array f with base cases f[0] = 0 and f[1] = 1. A for loop fills the array up to f[n] using the recurrence f[i] = f[i-1] + f[i-2]. Finally, the function returns f[n] as the result.

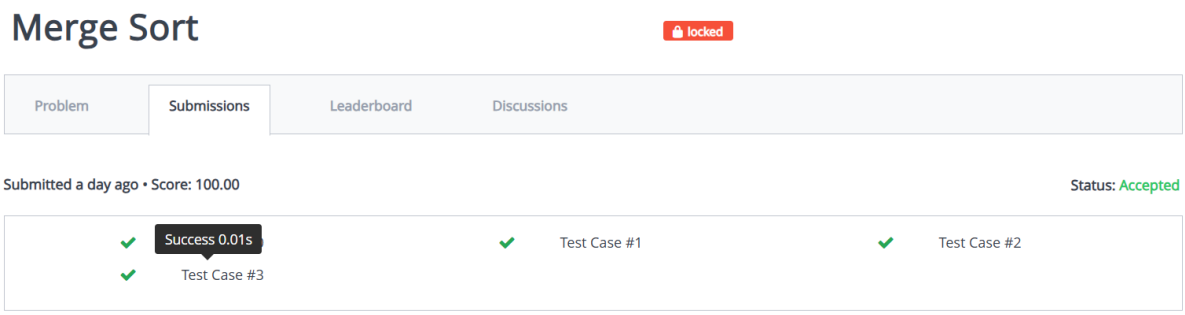
Why this algorithm solves the problem efficiently?

Simple Recurrence → Each term is the sum of the two previous terms.

Flexible → Works for recursion, iteration, or formula-based methods.

Predictable and Systematic → Generates the sequence in order efficiently.

5) Merge Sort



C++ code:

```cpp
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1, n2 = r - m;                    // sizes of left and right subarrays
    int L[n1], R[n2];                                  // temporary arrays
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];    // copy left half
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j]; // copy right half
    int i = 0, j = 0, k = l;                           // i,j for L and R; k for main array
    while (i < n1 && j < n2)                           // merge until one side ends
        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];   //add the small value and shift the pointer
    while (i < n1) arr[k++] = L[i++];                  // copy remaining left elements
    while (j < n2) arr[k++] = R[j++];                  // copy remaining right elements
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {                                       // continue only if more than one element
        int m = l + (r - l) / 2;                       // find middle index
        mergeSort(arr, l, m);                          // sort left half
        mergeSort(arr, m + 1, r);                      // sort right half
        merge(arr, l, m, r);                           // merge sorted halves
    }
}
```

Merge Sort is **a divide and conquer algorithm** that splits an array into two halves, recursively sorts each half, and then merges the sorted halves back together. It divides until each subarray has one element, which is inherently sorted. During merging, it compares elements from both halves and arranges them in order. This process continues until the entire array is sorted. **The time complexity is O(n log n)** and **the space complexity is O(n).**

Why this algorithm solves the problem efficiently ?

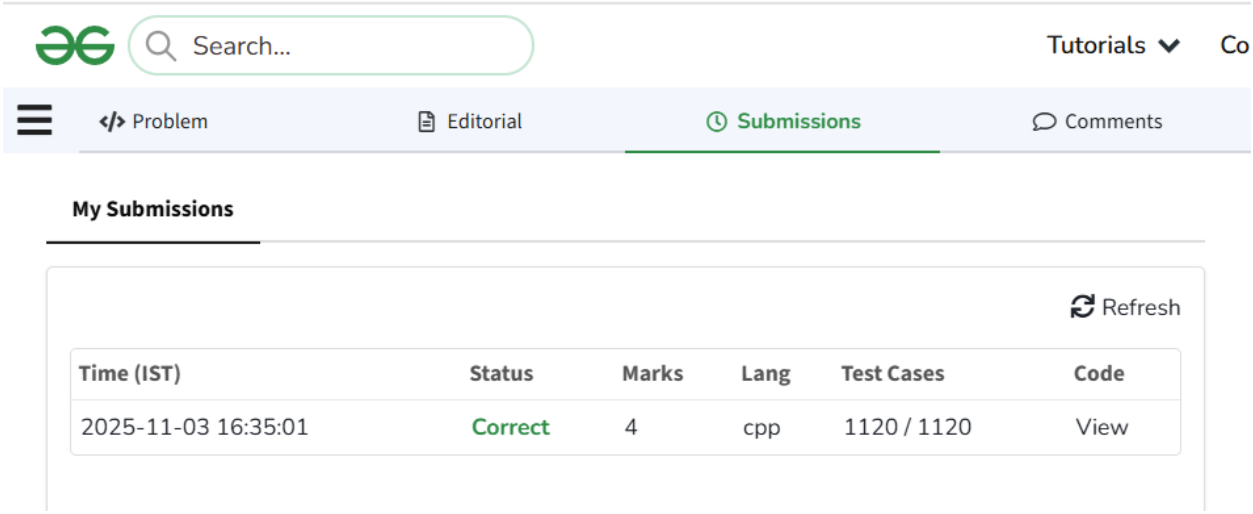
Divide and Conquer → Splits the array into halves recursively for easier sorting.

Efficient Time Complexity → Always O(n log n) regardless of input order.

Stable Sorting → Maintains relative order of equal elements.

Predictable and Systematic → Merges sorted halves step by step.

6) Maximum Subarray Sum (Kadane's Algorithm)



C++ code:

```cpp
class Solution {
  public:
    int maxSubarraySum(vector<int> &arr) {
        int i, n = arr.size(), curr = arr[0], mx = arr[0]; // curr: current subarray sum, mx: max sum found
        for(i = 1; i < n; i++) {
            curr = max(arr[i], curr + arr[i]);             // either extend subarray or start new
            mx = max(mx, curr);                            // update maximum value
        }
        return mx;                                         // return maximum subarray sum
    }
};
```

This algorithm implements **Kadane's Algorithm** to find the maximum subarray sum in an array. It keeps track of the current subarray sum (curr) and the maximum sum (mx) found so far. At each step, it decides whether to extend the current subarray or start a new one from the current element. Finally, mx stores the largest possible sum of any contiguous subarray. The **time complexity is O(n)** and **the space complexity is O(1).**

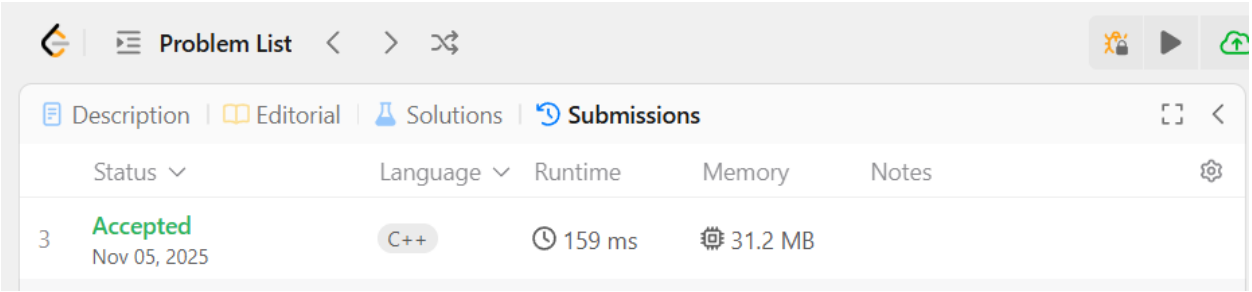
Why this algorithm solves the problem efficiently ?

Efficient and Linear → Finds maximum sum in O(n) by tracking current and global maxima.

In-place / Constant Space → Uses only a few variables, no extra memory.

Predictable and Simple → Iterates once through the array systematically.

7) 0/1 Knapsack Problem



C++ code:

```cpp
class Solution {
public:
    int dp[205][20005];                      // DP table to store states

bool fnc(int i,int sum,vector<int> &v)
{
    if(sum==0)return true;                   // target sum achieved
    if(i<0)return false;                     // no elements left
    if(dp[i][sum]!=-1)return dp[i][sum];     // return stored result

    bool tmp1=false,tmp2=false;              // tmp1: skip element, tmp2: take element
    tmp1=fnc(i-1,sum,v);                     // not taking v[i]
    if(sum-v[i]>=0)tmp2=fnc(i-1,sum-v[i],v); // taking v[i] if possible

    return dp[i][sum]=(tmp1|tmp2);           // store and return result
}

bool canPartition(vector<int>& nums) {
    memset(dp,-1,sizeof(dp));                // initialize DP with -1
    int i,sum=0;
    for(i=0;i<nums.size();i++)sum+=nums[i];  // total sum of array
    if(sum%2)return false;                   // if sum is odd, can't partition
    sum/=2;                                  // target sum per subset
    return fnc(nums.size()-1,sum,nums);
};
```

This algorithm solves the Partition Equal Subset Sum problem using dynamic programming. It first checks if the total sum is even — if not, partitioning is impossible. Then it uses a 1D DP array (dp[j]) where dp[j] indicates if a subset with sum j can be formed. For each number, it updates dp in reverse to avoid reusing the same element twice. The **time complexity is O(n × sum/2**) and **the space complexity is O(sum/2).**

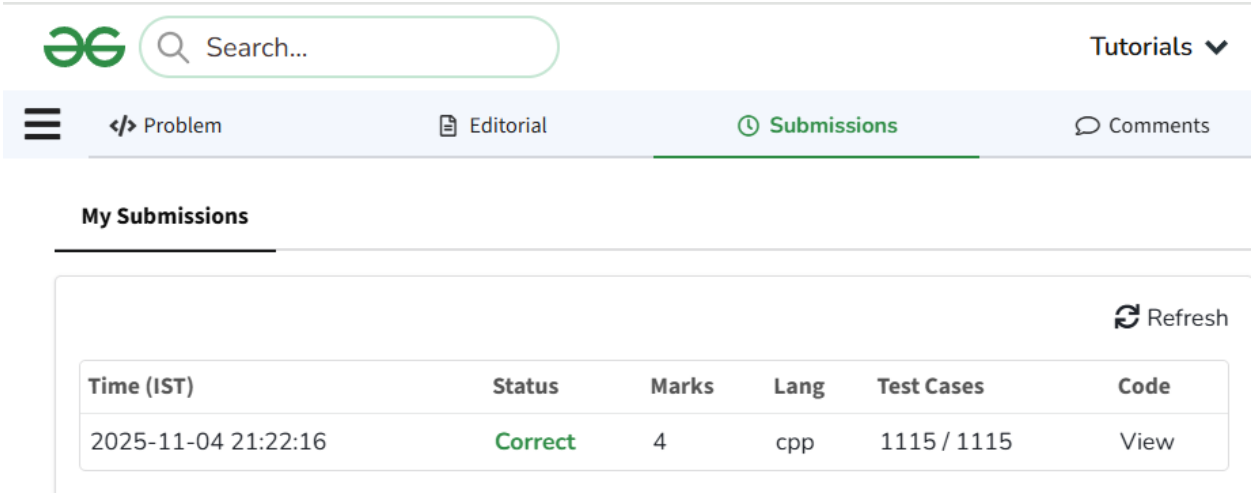
Why this algorithm solves the problem efficiently ?

Optimal Substructure → Can be solved by combining solutions of smaller subproblems.

Overlapping Subproblems → Dynamic programming avoids redundant calculations.

Flexible → Works for any set of items with weights and values.

8) **Longest Common Subsequence**



C++ code:

```cpp
class Solution {
  public:
    int lcs(string &s1, string &s2) {
        int n=s1.size(),m=s2.size(),i,j;                    // lengths and loop vars
        vector<vector<int>> v(n+1,vector<int>(m+1));        // DP table

        for(i=1;i<=n;i++) {
            for(j=1;j<=m;j++) {
                if(s1[i-1]==s2[j-1])                        // characters match
                    v[i][j]=v[i-1][j-1]+1;                  // extend LCS
                else
                    v[i][j]=max(v[i-1][j],v[i][j-1]);       // skip one character
            }
        }
        return v[n][m];                                     // final LCS length
    }
};
```

The Longest Common Subsequence (LCS) algorithm uses dynamic programming to find the longest sequence common to two strings. It builds a 2D DP table where dp[i][j] stores the LCS length of the first i characters of one string and the first j of the other. If the current characters match, dp[i][j] = 1 + dp[i-1][j-1]; otherwise, it takes the max of top or left cells. After filling the table, dp[n][m] gives the final LCS length. The **time complexity is O(n × m)** and the **space complexity is O(n × m).**

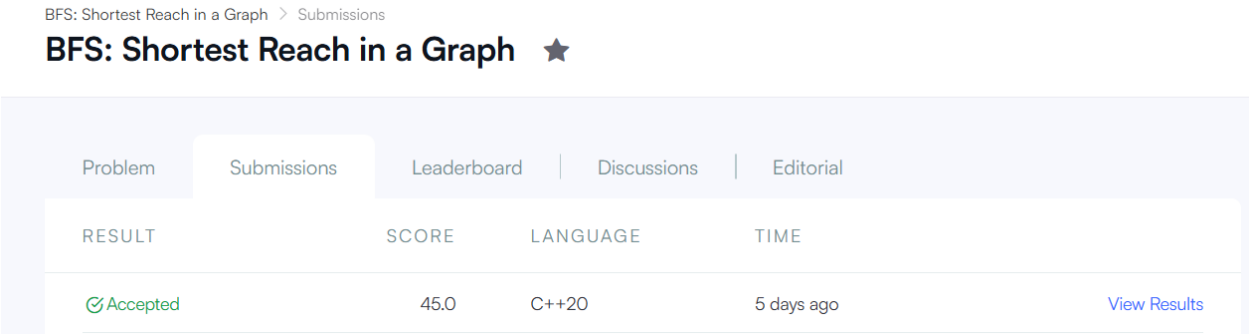
Why this algorithm solves the problem efficiently ?

Optimal Substructure → Can be broken into smaller subproblems, solved with DP.

Overlapping Subproblems → Reuses results to avoid redundant computations.

Predictable and Systematic → DP table ensures all possibilities are considered efficiently

9) BFS and DFS traversal of a graph



C++ code:

```cpp
ll bfs(ll s, ll n, vl a[])
{
    vector<bool> vis(n+1,false);   // visited array to track visited nodes
    vl ans(n+1,-1);                // stores shortest distance from s
    queue<ll> q;                   // queue for BFS

    vis[s] = true;                 // mark starting node as visited
    ans[s] = 0;                    // distance to start node = 0
    q.push(s);                     // push starting node into queue

    ll tmp;
    while(!q.empty())
    {
        tmp = q.front();           // current node
        q.pop();

        for(ll i = 0; i < a[tmp].size(); i++) // explore all neighbors
        {
            if(vis[a[tmp][i]] == 0)          // if neighbor not visited
            {
                vis[a[tmp][i]] = 1;          // mark as visited
                ans[a[tmp][i]] = 6 + ans[tmp]; // update distance (edge weight = 6)
                q.push(a[tmp][i]);           // push neighbor to queue
            }
        }
    }

    // print distances to all nodes except the starting node
    for(ll i = 1; i <= n; i++)
    {
        if(ans[i] != 0) cout << ans[i] << " ";
    }
    cout << endl;

    return 0; // return value not used, just for signature
}
```

This algorithm performs **Breadth-First Search (BFS)** to find the shortest distance from a starting node to all other nodes in an unweighted graph. It uses an adjacency list to store connections and a queue for level-wise traversal. Each edge contributes a fixed distance of 6 units when moving to a neighbor node. Unreachable nodes are marked with -1 after traversal. **The time complexity is O(V + E)** and **the space complexity is O(V),** where V = vertices and E = edges.

Why this algorithm solves the problem efficiently ?

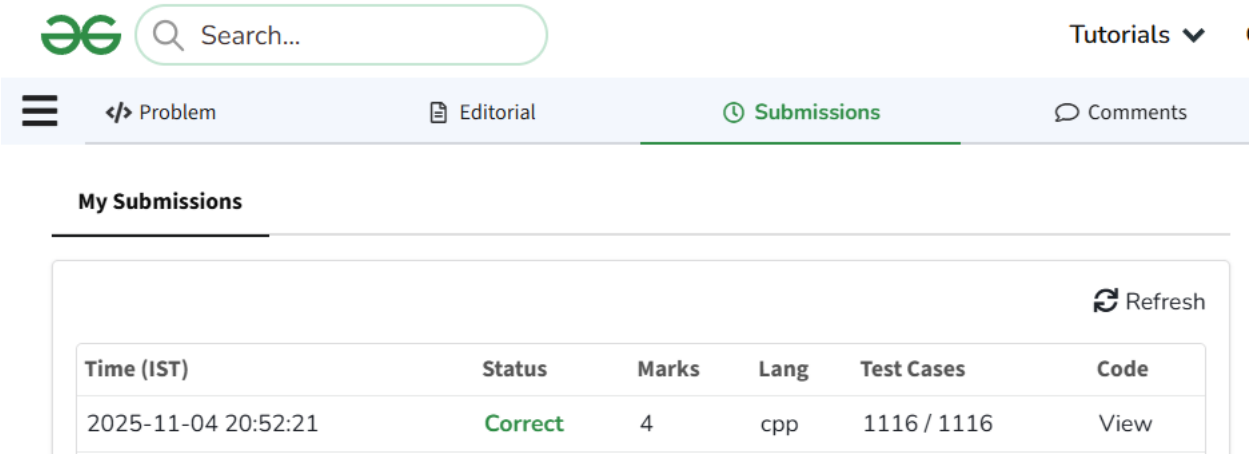
Level-wise Traversal → Explores all nodes at the current depth before moving deeper.

Shortest Path Guarantee → Finds the shortest path in unweighted graphs.

Systematic and Predictable → Uses a queue to ensure nodes are visited in order.

Memory Efficient for Shallow Graphs → Only stores the current level in the queue

10) Dijkstra's Shortest Path



C++ code:

```cpp
vector<int> dijkestra(int V, vector<vector<pair<int,int>>> &adj, int src)
{
    // Min-heap priority queue to select node with smallest distance
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;

    vector<int> dis(V, INT_MAX);      // distance vector initialized to infinity
    dis[src] = 0;                     // distance to source = 0
    pq.push({0, src});                // push source node into priority queue
    while(!pq.empty())
    {
        int u = pq.top().second;      // current node
        int d = pq.top().first;       // distance to current node
        pq.pop();

        if(d > dis[u]) continue;      // if this is not the latest distance, skip

        for(auto edge : adj[u])       // explore all neighbors
        {
            int v = edge.first;       // neighbor node
            int w = edge.second;      // edge weight
            if(dis[u] + w < dis[v])   // relaxation step
            {
                dis[v] = dis[u] + w;
                pq.push({dis[v], v}); // push updated distance
            }
        }
    }
    return dis;                       // return distances from source to all nodes
}
```

Dijkstra's algorithm finds the **shortest path from a source vertex** to all other vertices in a weighted graph with non-negative edge weights.
It maintains a set of visited vertices and repeatedly selects the vertex with the **minimum tentative distance**. For each selected vertex, it updates the distances of its adjacent vertices if a shorter path is found through it. The process continues until all vertices are visited, resulting in the shortest paths from the source to every vertex.

Why Dijkstra's Shortest Path Works Efficiently

- **Greedy Approach →** Always selects the vertex with the smallest tentative distance, ensuring optimal paths.
- **Predictable and Systematic →** Updates distances of adjacent vertices step by step until all shortest paths from the source are found.
- **Efficient for Non-Negative Weights →** Works correctly when all edge weights are non-negative, making it reliable in many real-world networks.
- **Time-Space Complexity →** Runs in **$O(V^2)$** with a simple array or **$O((V + E) \log V)$** with a priority queue (V = vertices, E = edges), and uses **$O(V)$** space for distances and tracking.