Bi-Weekly Contest 120

2970. Count the Number of Incremovable Subarrays I

Solved

Easy · Companies · Hint

You are given a **0-indexed** array of **positive** integers `nums`.

A subarray of `nums` is called **incremovable** if `nums` becomes **strictly increasing** on removing the subarray. For example, the subarray `[3, 4]` is an incremovable subarray of `[5, 3, 4, 6, 7]` because removing this subarray changes the array `[5, 3, 4, 6, 7]` to `[5, 6, 7]` which is strictly increasing.

Return *the total number of* ***incremovable*** *subarrays of* `nums`.

Note that an empty array is considered strictly increasing.

A **subarray** is a contiguous non-empty sequence of elements within an array.

Example 1:

```
Input: nums = [1,2,3,4]
Output: 10
Explanation: The 10 incremovable subarrays are: [1], [2], [3], [4], [1,2], [2,3], [3,4], [1,2,3], [2,3,4], and [1,2,3,4], because on removing any one of these subarrays nums becomes strictly increasing. Note that you cannot select an empty subarray.
```

Example 2:

```
Input: nums = [6,5,7,8]
Output: 7
Explanation: The 7 incremovable subarrays are: [5], [6], [5,7], [6,5], [5,7,8], [6,5,7] and [6,5,7,8].
It can be shown that there are only 7 incremovable subarrays in nums.
```

Example 3:

```
Input: nums = [8,7,6,6]
Output: 3
Explanation: The 3 incremovable subarrays are: [8,7,6], [7,6,6], and [8,7,6,6]. Note that [8,7] is not an incremovable subarray because after removing [8,7] nums becomes [6,6], which is sorted in ascending order but not strictly increasing.
```

Constraints:

- `1 <= nums.length <= 50`
- `1 <= nums[i] <= 50`

Solution 1:

```
class Solution {
public:
    int incremovableSubarrayCount(vector<int>& nums) {
        int ans = 0; // Initialize the answer variable to 0.

        // Iterate through all possible subarrays defined by the indices i and j.
        for (int i = 0; i < nums.size(); ++i) {
            for (int j = i; j < nums.size(); ++j) {
                int last = -1, flag = 1; // Initialize variables to track the last element and a flag
indicating whether the subarray is incremovable.

                // Iterate through the array to check if removing the subarray defined by i and j
makes the array strictly increasing.
```

```
            for (int k = 0; k < nums.size(); ++k) {
                if (k >= i && k <= j) continue; // Skip the elements within the subarray
defined by i and j.

                if (last >= nums[k]) { // If the current element violates the increasing order,
set the flag to 0 and break the loop.
                    flag = 0;
                    break;
                }
                last = nums[k]; // Update the last element.
            }

            // If the flag is still 1, the subarray is incremovable, so increment the answer.
            if (flag) {
                ans++;
            }
        }
    }

    return ans; // Return the final answer.
  }
};
```

Explanation:

Outer Loop (`i`): This loop iterates through all possible starting indices of the subarray.
Inner Loop (`j`): This loop iterates through all possible ending indices of the subarray. It starts from the current `i` to the end of the array.
Subarray Check (`k`): The innermost loop checks whether the subarray defined by the current `i` and `j` is incremovable. It skips the elements within this subarray and compares each element to the previous one (`last`). If at any point the order is violated, the flag is set to 0, indicating that the subarray is not incremovable.
Result Update: If the flag remains 1 after the innermost loop, it means the subarray is incremovable, and the answer is incremented.
Return: The final count of incremovable subarrays is returned as the result.

Note: While this solution works, it is not efficient, and there are more optimized algorithms to solve this problem. The given code has a time complexity of O(n^3), where n is the length of the input array, making it inefficient for larger inputs.

Solution 2:

```
class Solution {
public:
    // Function to check if a subarray is strictly increasing
    bool check(vector<int> cnums) {
        for (int i = 1; i < cnums.size(); i++) {
            if (cnums[i] <= cnums[i - 1])
                return false;
        }
        return true;
    }

    // Main function to find the total number of incremovable subarrays
    int incremovableSubarrayCount(vector<int>& nums) {
        int n = nums.size();
        int ans = 0;

        // Iterate through all possible subarrays defined by the indices i and j
        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                vector<int> cnums;

                // Construct a new array excluding elements between i and j
                for (int k = 0; k < i; k++) {
                    cnums.push_back(nums[k]);
                }

                for (int k = j + 1; k < n; k++) {
                    cnums.push_back(nums[k]);
                }

                // Check if the constructed subarray is strictly increasing
                if (check(cnums))
                    ans++;
            }
        }
```

```
        return ans; // Return the final answer
    }
};
```

Explanation:

- `check` Function: This function takes a vector `cnums` as input and checks if it is strictly increasing. It does so by iterating through the elements of `cnums` and returning `false` if any adjacent pair violates the increasing order.
- Main Function (`incremovableSubarrayCount`):
  - Outer Loop (`i`): Iterates through all possible starting indices of the subarray.
  - Inner Loop (`j`): Iterates through all possible ending indices of the subarray, starting from the current `i`.
  - Subarray Construction (`cnums`): Constructs a new vector `cnums` by excluding elements between the current `i` and `j`.
  - Check if Subarray is Incremovable: Calls the `check` function to verify if the constructed subarray is strictly increasing. If it is, increments the answer.
- Return: Returns the final count of incremovable subarrays as the answer.

While this code is correct, it is also not efficient, with a time complexity of O(n^3) due to the three nested loops. There are more optimized algorithms to solve this problem with better time complexity.

Solution 3 :

```
class Solution {
public:
    int incremovableSubarrayCount(vector<int>& nums) {
        int n = nums.size();
        int count = 0;

        // Handle the case of a single-element array
        if (n == 1) return 1;

        // Iterate through all possible subarrays defined by the indices i and j
        for (int i = 0; i < n; i++) {
            unordered_map<int, int> mp; // Map to store indices to be excluded
```

```
        for (int j = i; j < n; j++) {
            vector<int> temp; // Temporary vector to construct the subarray

            // Mark the index j to be excluded
            mp[j]++;
            bool helpme = false;

            // Construct a new array excluding elements marked in the map
            for (int k = 0; k < n; k++) {
                if (mp.find(k) == mp.end()) {
                    temp.push_back(nums[k]);
                }
            }

            // Check if the constructed subarray is incremovable
            if (temp.size() == 1 || temp.size() == 0) {
                count += 1;
            } else {
                for (int m = 0; m < temp.size() - 1; m++) {
                    if (temp[m] >= temp[m + 1]) {
                        helpme = true;
                    }
                }

                if (helpme == false) {
                    count += 1;
                }
            }
        }
    }

    return count; // Return the final answer
  }
};
```
Explanation:

Initialization: Initialize variables, including `count` to keep track of the number of incremovable subarrays.
Handling Single Element Array: If the array has only one element, return 1 because a single element is considered strictly increasing.

Main Loop (`i`): Iterate through all possible starting indices of the subarray.
Nested Loop (`j`): Iterate through all possible ending indices of the subarray, starting from the current `i`.
Map (`mp`): Use an unordered map to mark the indices that should be excluded from the subarray.
Subarray Construction (`temp`): Construct a new vector `temp` by excluding elements marked in the map.
Incremovable Check: Check if the constructed subarray is incremovable. If it has only one or no elements, increment the count. Otherwise, check if the elements are strictly increasing. If so, increment the count.
Return: Return the final count of incremovable subarrays as the answer.

While this code is correct, it is not efficient, with a time complexity of O(n^4) due to the four nested loops. There are more optimized algorithms to solve this problem with better time complexity.

Solution 4-

```
class Solution {

public:

    long long incremovableSubarrayCount(vector<int>& nums) {

        int n = nums.size();

        vector<int> start, end;



        // Construct the increasing subarray from the beginning

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

            if (start.empty() || start.back() < nums[i]) {

                start.push_back(nums[i]);

            } else {
```

```
            break;
        }
    }

    // Construct the increasing subarray from the end
    for (int i = n - 1; i >= 0; --i) {
        if (end.empty() || end.back() > nums[i]) {
            end.push_back(nums[i]);
        } else {
            break;
        }
    }

    // If the combined size of start and end is greater than n, return the total possible
subarrays
    if (start.size() + end.size() > n) {
        return 1LL * n * (n + 1) / 2;
    }

    long long ans = start.size() + end.size();
    int i = 0, j = 0;
```

```
        // Reverse the end vector to compare elements easily
        reverse(end.begin(), end.end());

        // Merge the increasing subarrays and count the incremovable subarrays
        while (i < start.size() && j < end.size()) {
            if (start[i] < end[j]) {
                ans += end.size() - j;
                ++i;
            } else {
                ++j;
            }
        }

        return ans + 1; // Add 1 to include the case of an empty subarray
    }
};
```

Explanation:

Constructing Increasing Subarrays (`start` and `end`):

- The first loop (`i` from 0 to n-1) constructs an increasing subarray (`start`) from the beginning of the array.

- The second loop (`i` from n-1 to 0) constructs an increasing subarray (`end`) from the end of the array.

Total Possible Subarrays Check:

- If the combined size of `start` and `end` is greater than `n`, it means that the array itself is strictly increasing, so the total number of subarrays is returned using the formula for the sum of the first `n` natural numbers.

Merging and Counting:

- The code then merges the increasing subarrays (`start` and `end`) and counts the incremovable subarrays.
- It uses two pointers (`i` and `j`) to iterate through `start` and `end` vectors.
- It compares elements at the pointers, and if an element from `start` is less than an element from `end`, it increments the answer by the remaining size of `end` (`end.size() - j`).
- This is because if `start[i] < end[j]`, then all subarrays with `start[i]` and elements in `end` from `end[j]` to the end are incremovable.
- The pointers are then appropriately incremented.

Return:

- The final answer is returned by adding 1 to include the case of an empty subarray.

This solution has a time complexity of O(n), making it more efficient compared to the previous solutions.

2971. Find Polygon With the Largest Perimeter

Solved

Medium · Companies · Hint

You are given an array of **positive** integers `nums` of length `n`.

A **polygon** is a closed plane figure that has at least `3` sides. The **longest side** of a polygon is **smaller** than the sum of its other sides.

Conversely, if you have `k` (`k >= 3`) **positive** real numbers a_1, a_2, a_3, ..., a_k where $a_1 <= a_2 <= a_3 <= ... <= a_k$ **and** $a_1 + a_2 + a_3 + ... + a_{k-1} > a_k$, then there **always** exists a polygon with `k` sides whose lengths are a_1, a_2, a_3, ..., a_k.

The **perimeter** of a polygon is the sum of lengths of its sides.

*Return the **largest** possible **perimeter** of a **polygon** whose sides can be formed from* `nums`, *or* `-1` *if it is not possible to create a polygon.*

Example 1:

```
Input: nums = [5,5,5]
Output: 15
Explanation: The only possible polygon that can be made from nums has 3 sides: 5, 5, and 5. The perimeter is 5 + 5 + 5 = 15.
```

Example 2:

```
Input: nums = [1,12,1,2,5,50,3]
Output: 12
Explanation: The polygon with the largest perimeter which can be made from nums has 5 sides: 1, 1, 2, 3, and 5. The
perimeter is 1 + 1 + 2 + 3 + 5 = 12.
We cannot have a polygon with either 12 or 50 as the longest side because it is not possible to include 2 or more smaller
sides that have a greater sum than either of them.
It can be shown that the largest possible perimeter is 12.
```

Example 3:

```
Input: nums = [5,5,50]
Output: -1
Explanation: There is no possible way to form a polygon from nums, as a polygon
has at least 3 sides and 50 > 5 + 5.
```

Constraints:

- $3 <= n <= 10^5$
- $1 <= nums[i] <= 10^9$

Solution-1

```
class Solution {

public:

    long long largestPerimeter(vector<int>& nums) {
```

```
long long sum = 0; // Initialize a variable to store the sum of all elements

sort(nums.begin(), nums.end()); // Sort the elements in non-decreasing order



// Calculate the sum of all elements in the input vector

for (auto i : nums) {

    sum += i;

}



int n = nums.size(); // Get the size of the input vector



// Loop through the sorted vector from the end to find the largest perimeter

for (int i = n - 1; i >= 2; i--) {

    sum -= nums[i]; // Remove the largest element from the sum

    // Check if the sum of two smaller sides is greater than the largest side

    if (sum > nums[i]) {

        return sum + nums[i]; // If true, return the largest perimeter
```

```
        }
      }

      return -1; // If no valid triangle is possible, return -1
    }
};
```

Explanation:

Sorting the Array: The input array `nums` is sorted in non-decreasing order. Sorting helps in identifying the largest sides of the potential polygon towards the end of the array.

Calculating the Sum: The sum of all elements in the array is calculated.

Finding the Largest Perimeter:

- The code then loops through the sorted array from the end (`i = n - 1`) to the beginning (`i >= 2`). It starts from the end because we are looking for the largest sides.
- In each iteration, the largest element (`nums[i]`) is removed from the total sum (`sum -= nums[i]`).
- The code then checks if the sum of the two remaining smaller sides is greater than the largest side (`if (sum > nums[i])`).
- If the condition is true, it means a valid triangle is possible, and the function returns the largest perimeter (`return sum + nums[i]`).

Returning -1 if No Valid Triangle: If no valid triangle is found after the loop, the function returns -1.

The time complexity of this solution is O(n log n) due to the sorting operation. The loop has a time complexity of O(n), resulting in an overall time complexity of O(n log n).

Solution-2

```
# define ll long long


class Solution {

public:

    long long largestPerimeter(vector<int>& nums) {

        sort(nums.begin(), nums.end()); // Sort the elements in non-decreasing order

        ll prefSum = nums[0] + nums[1]; // Initialize a variable to store the sum of the two smallest elements

        ll ans = 0; // Initialize the variable to store the largest perimeter


        // Iterate through the sorted array from the third element

        for(int i = 2; i < nums.size(); i++) {
```

```
            if(prefSum > nums[i]) {
                ans = max(ans, prefSum + nums[i]); // Check if the sum of two smaller sides is greater than the third side
            }


            prefSum += nums[i]; // Update the sum by adding the current element
        }


        return (ans == 0) ? -1 : ans; // If no valid triangle is possible, return -1; otherwise, return the largest perimeter
    }
};
```

Explanation:

Sorting the Array: The input array `nums` is sorted in non-decreasing order.

Initializing Variables:

- `prefSum`: This variable is initialized with the sum of the two smallest elements in the sorted array.
- `ans`: This variable is initialized to 0 and will be used to store the largest perimeter.

Iterating Through the Sorted Array:

- The code iterates through the sorted array starting from the third element (`i = 2`).
- In each iteration, it checks if the sum of the two smallest elements (`prefSum`) is greater than the current element (`nums[i]`).
- If the condition is true, it means a valid triangle is possible, and the code updates the `ans` with the maximum perimeter.

Updating the Sum:

- The `prefSum` is updated by adding the current element (`prefSum += nums[i]`).

Returning the Result:

- If no valid triangle is found (i.e., `ans` remains 0), the function returns -1.
- Otherwise, it returns the largest perimeter stored in the `ans` variable.

The time complexity of this solution is O(n log n) due to the sorting operation. The loop has a time complexity of O(n), resulting in an overall time complexity of O(n log n). The use of `ll` (long long) suggests that the code accounts for potential large values, ensuring correct results for edge cases.

Solution-3

```
class Solution {
    typedef long long ll; // Define a type alias for long long
public:
    long long largestPerimeter(vector<int>& nums) {
        ll sum = 0; // Initialize a variable to store the sum of all elements

        // Calculate the sum of all elements in the input vector
        for(auto & it: nums) {
            sum += it;
        }

        priority_queue<ll> pq; // Initialize a max heap (priority queue) to efficiently get the
largest elements

        // Push all elements into the priority queue
```

```
for(auto & it: nums) {
    pq.push(it);
}

// Loop until the priority queue is empty
while(!pq.empty()) {
    ll val = pq.top(); // Get the largest element from the priority queue
    sum -= val; // Remove the largest element from the sum

    // Check if the sum of two smaller sides is greater than the largest side
    if(sum > val) {
        return sum + val; // If true, return the largest perimeter
    } else {
        pq.pop(); // If not true, try another value by removing the largest element
    }
}
```

```
        return -1; // If no valid triangle is possible, return -1
    }
};
```

Explanation:

Calculating the Sum: The code calculates the sum of all elements in the input array `nums`.

Initializing the Priority Queue (Max Heap): It initializes a priority queue (`pq`) as a max heap, which ensures that the largest elements are at the top.

Pushing Elements into the Priority Queue: All elements from the input array are pushed into the priority queue.

Main Loop:

- The code then enters a loop that continues until the priority queue is empty.
- In each iteration, it gets the largest element from the priority queue (`val`) and removes it from the sum.
- It checks if the sum of the two remaining smaller sides is greater than the largest side.
- If the condition is true, it means a valid triangle is possible, and the function returns the largest perimeter.
- If not true, it tries another value by removing the largest element from the priority queue.

Returning -1 if No Valid Triangle: If no valid triangle is found after the loop, the function returns -1.

The use of `ll` (long long) suggests that the code accounts for potential large values, ensuring correct results for edge cases. The priority queue efficiently handles the process of selecting the largest elements from the array. The overall time complexity of this solution is O(n log n), where n is the size of the input array.

2973. Find Number of Coins to Place in Tree Nodes

Solve

Hard · Companies · Hint

You are given an **undirected** tree with `n` nodes labeled from `0` to `n - 1`, and rooted at node `0`. You are given a 2D integer array `edges` of length `n - 1`, where `edges[i] = [`a_i`, `b_i`]` indicates that there is an edge between nodes a_i and b_i in the tree.

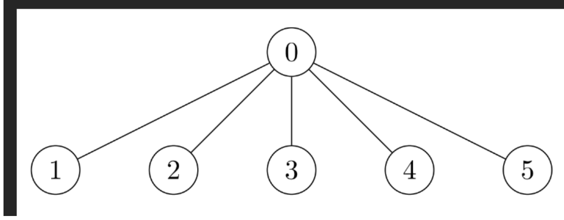
You are also given a **0-indexed** integer array `cost` of length `n`, where `cost[i]` is the **cost** assigned to the i^{th} node.

You need to place some coins on every node of the tree. The number of coins to be placed at node `i` can be calculated as:

- If size of the subtree of node `i` is less than `3`, place `1` coin.
- Otherwise, place an amount of coins equal to the **maximum** product of cost values assigned to `3` distinct nodes in the subtree of node `i`. If this product is **negative**, place `0` coins.

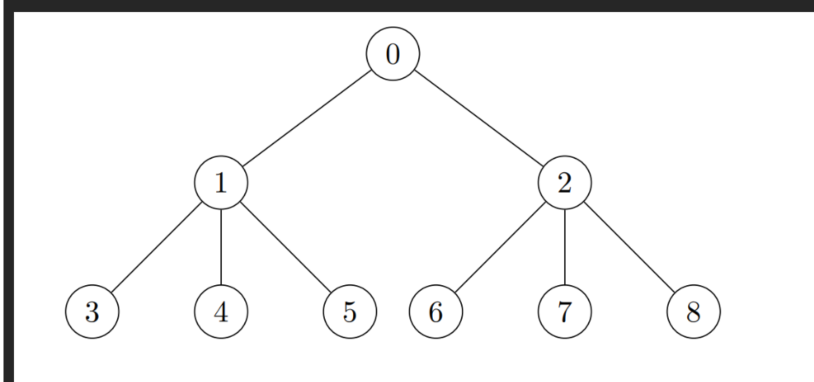
Return *an array* `coin` *of size* `n` *such that* `coin[i]` *is the number of coins placed at node* `i`.

Example 1:




```
Input: edges = [[0,1],[0,2],[0,3],[0,4],[0,5]], cost = [1,2,3,4,5,6]
Output: [120,1,1,1,1,1]
Explanation: For node 0 place 6 * 5 * 4 = 120 coins. All other nodes are leaves with subtree of size 1, place 1 coin on each of them.
```

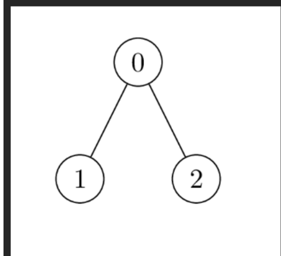
Example 2:




```
Input: edges = [[0,1],[0,2],[1,3],[1,4],[1,5],[2,6],[2,7],[2,8]], cost = [1,4,2,3,5,7,8,-4,2]
Output: [280,140,32,1,1,1,1,1,1]
Explanation: The coins placed on each node are:
- Place 8 * 7 * 5 = 280 coins on node 0.
- Place 7 * 5 * 4 = 140 coins on node 1.
- Place 8 * 2 * 2 = 32 coins on node 2.
```

– All other nodes are leaves with subtree of size 1, place 1 coin on each of them.

Example 3:



> **Input:** edges = [[0,1],[0,2]], cost = [1,2,-2]
> **Output:** [0,1,1]
> **Explanation:** Node 1 and 2 are leaves with subtree of size 1, place 1 coin on each of them. For node 0 the only possible product of cost is 2 * 1 * -2 = -4. Hence place 0 coins on node 0.

Constraints:

- `2 <= n <= 2 * 10^4`
- `edges.length == n - 1`
- `edges[i].length == 2`
- $0 <= a_i, b_i < n$
- `cost.length == n`
- `1 <= |cost[i]| <= 10^4`
- The input is generated such that `edges` represents a valid tree.

Solution-1-

```
class Solution {

public:

    vector<long long> ans;



    vector<long long> dfs(vector<vector<int>>& t, vector<int>& cost, int root, int par){
```

```
        vector<long long> usefulCost = {cost[root]}; // Initialize the vector with the cost of
the current node

        // Traverse through the children of the current node
        for(auto n: t[root]){
            if(n == par) continue; // Skip the parent node
            vector<long long> v = dfs(t, cost, n, root);  // Recursively go deep into the leaf
nodes first
            for(auto e: v) usefulCost.push_back(e);  // Accumulate all the costs from all its
child nodes at the root level
        }

        // After traversing all subtrees, sort the accumulated cost
        sort(usefulCost.begin(), usefulCost.end(), greater<long long>());
        long long sz = usefulCost.size();

        if(usefulCost.size() < 3) {
```

```
    ans[root] = 1;

    return usefulCost;

  }  // Check if the size of the subtree is less than 3, then set the cost to 1 and return
from here.



  if(usefulCost[1] * usefulCost[2] > usefulCost[sz - 1] * usefulCost[sz -2]) {

    ans[root] = usefulCost[0] * usefulCost[1] * usefulCost[2];

  }

  else {

    ans[root] = usefulCost[0] * usefulCost[sz - 1] * usefulCost[sz - 2];

  }

  if(ans[root] < 0) ans[root] = 0; // If the product is negative, set cost to 0



  // If the size of the subtree is less than or equal to 5, return the useful costs

  if(usefulCost.size() <= 5) return usefulCost;
```

```
        // Return the largest 3 and smallest two items, only those can be useful in later
steps and discard others
        return {usefulCost[0], usefulCost[1], usefulCost[2], usefulCost[sz-2],
usefulCost[sz-1]};
    }

    vector<long long> placedCoins(vector<vector<int>>& edges, vector<int>& cost) {
        ans.resize(cost.size(), 0);
        vector<vector<int>> t(cost.size());

        // Build the tree from the given edges
        for(auto e : edges){
            t[e[0]].push_back(e[1]);
            t[e[1]].push_back(e[0]);
        }

        // Perform DFS to calculate the number of coins for each node
```

```
        dfs(t, cost, 0, -1);


        return ans; // Return the vector containing the number of coins for each node
    }
};
```

Explanation:

Depth-First Search (DFS):

- The `dfs` function performs a depth-first search on the tree.
- It recursively traverses through the children of the current node and accumulates the costs from its child nodes.
- After traversing all subtrees, it sorts the accumulated costs in descending order.

Calculating Coins:

- If the size of the subtree is less than 3, the cost for the current node is set to 1.
- Otherwise, the cost is calculated as the maximum product of the cost values assigned to three distinct nodes in the subtree.
- If the product is negative, the cost is set to 0.

Returning Useful Costs:

- The function returns the largest 3 and smallest two items, as only those can be useful in later steps, and discards the others.

Building the Tree:

- The `placedCoins` function builds the tree from the given edges.

Overall Process:

- The DFS is performed to calculate the number of coins for each node, and the result is stored in the `ans` vector.

Returning the Result:

- The `ans` vector containing the number of coins for each node is returned as the final result.

The code ensures that the cost is calculated according to the specified conditions, and the time complexity is determined by the DFS traversal of the tree.

Solution2-

```
typedef long long int ll;

const ll NO_VALUE = 1e9;



class Solution {

    int n;

    vector<vector<int>> g;

    vector<int> cost;
```

```
vector<vector<ll>> val;

vector<ll> coins;



void CalculateCoins (int src, int par) {

    vector<ll> child_val = {cost[src]};



    // Traverse through the children of the current node

    for (auto i : g[src]) {

        if (i == par) continue;



        CalculateCoins (i, src);

        for (auto i : val[i]) child_val.push_back(i);

    }

    sort (child_val.begin(), child_val.end());
```

```
        if (child_val.size() < 3) {

            coins[src] = 1;

            val[src] = child_val;

        }

        else {

            int n = child_val.size();

            ll max_product = -1e18;

            max_product = max (max_product, child_val[0]*child_val[1]*child_val[n-1]);

            max_product = max (max_product, child_val[n-1]*child_val[n-2]*child_val[n-3]);



            coins[src] = max_product < 0? 0 : max_product;



            // If the size of the subtree is less than 6, return all values; otherwise, return the
largest 3 and smallest two items

            if (child_val.size() < 6) val[src] = child_val;

            else val[src] = {child_val[0], child_val[1], child_val[n-3], child_val[n-2],
child_val[n-1]};
```

```
        }
    }

public:
    vector<long long> placedCoins(vector<vector<int>>& edges, vector<int>& _cost) {
        n = _cost.size();


        g.clear(), val.clear(), cost.clear(), coins.clear();
        val.resize(n), g.resize(n), cost.resize(n), coins.resize(n);
        cost = _cost;


        for (auto e: edges) {
            g[e[0]].push_back(e[1]);
            g[e[1]].push_back(e[0]);
        }
```

```
        CalculateCoins(0, -1);

        return coins; // Return the vector containing the number of coins for each node

    }

};
```

Explanation:

CalculateCoins Function:

- The `CalculateCoins` function is a recursive function that calculates the number of coins for each node based on the given conditions.
- It traverses through the children of the current node, accumulates the costs from its child nodes, and sorts them.
- If the size of the subtree is less than 3, the cost for the current node is set to 1.
- Otherwise, the cost is calculated as the maximum product of the cost values assigned to three distinct nodes in the subtree.
- If the product is negative, the cost is set to 0.
- If the size of the subtree is less than 6, it returns all values; otherwise, it returns the largest 3 and smallest two items.

placedCoins Function:

- The `placedCoins` function initializes the necessary vectors and calls the `CalculateCoins` function to calculate the number of coins for each node.

Overall Process:

- The code ensures that the cost is calculated according to the specified conditions, and the time complexity is determined by the recursive traversal of the tree.

Returning the Result:

- The `coins` vector containing the number of coins for each node is returned as the final result.