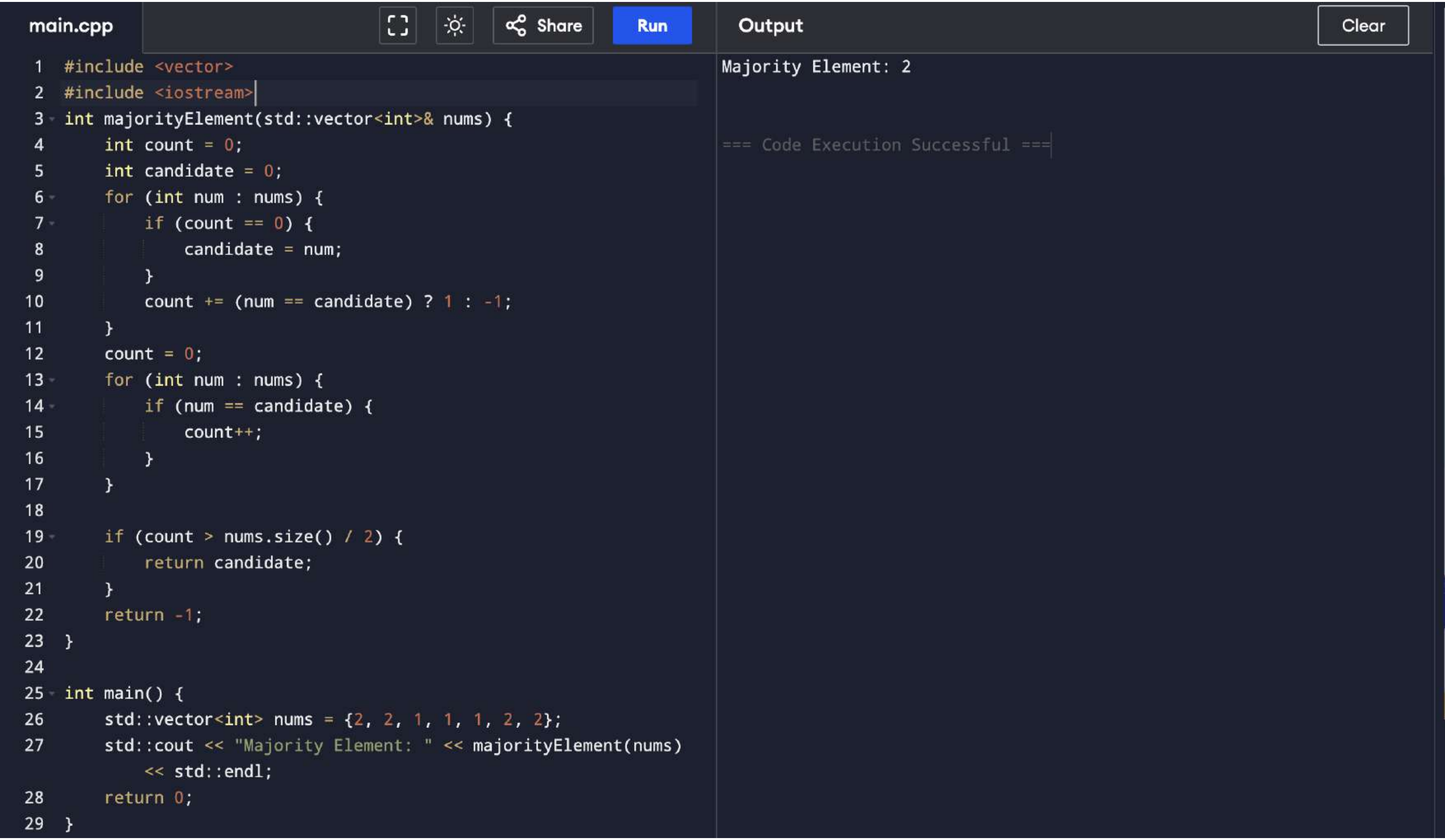
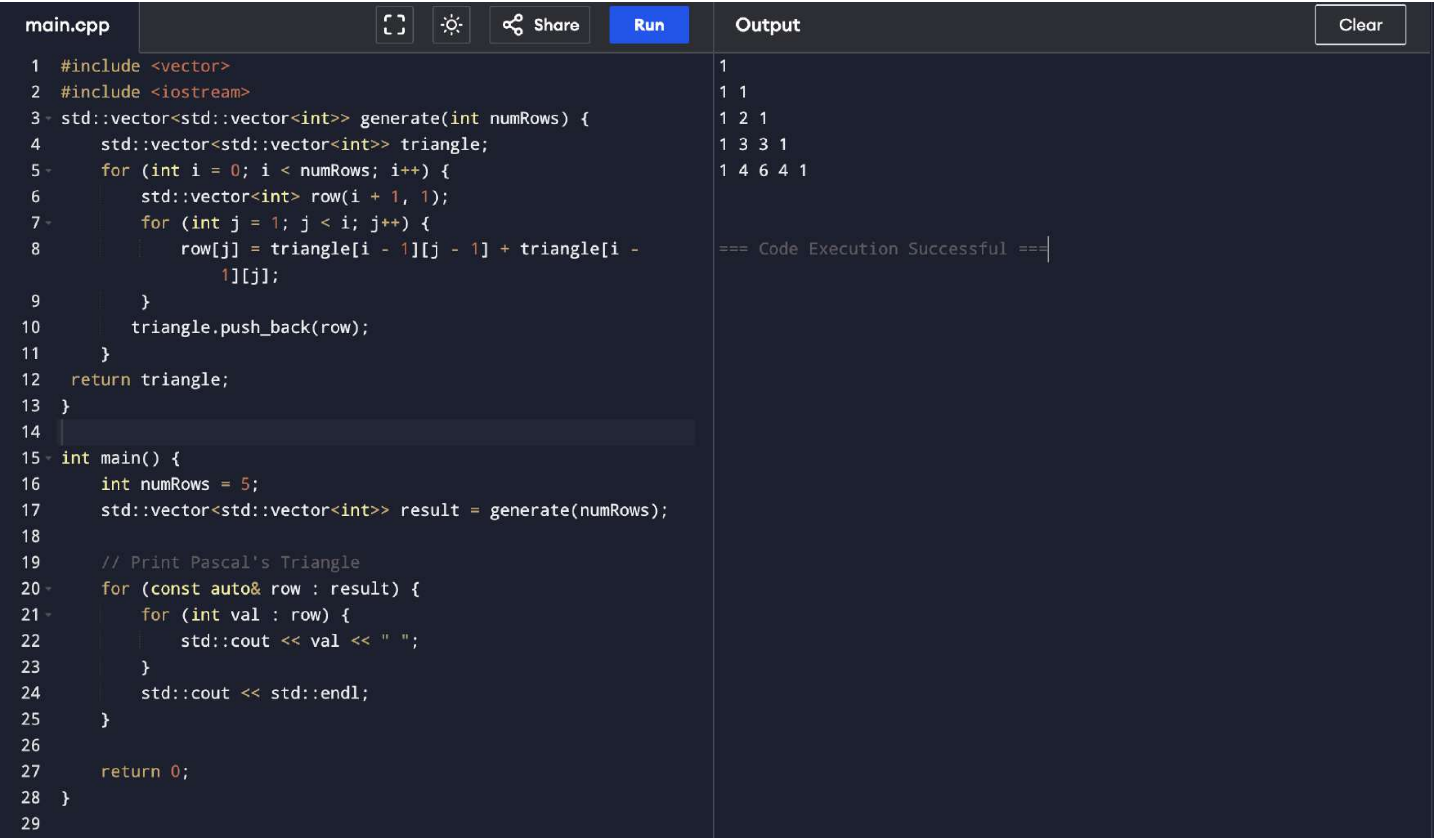


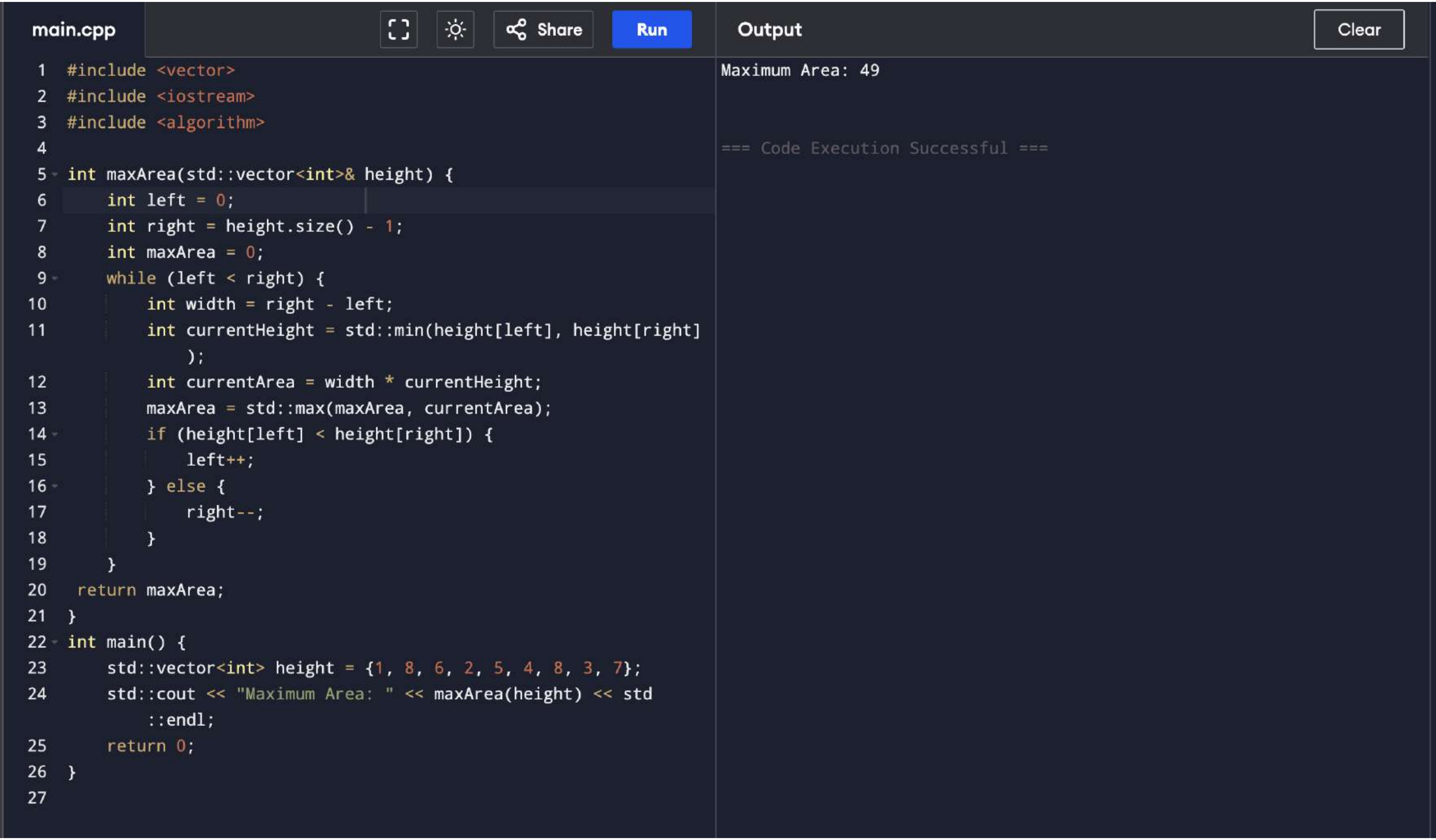
```cpp
#include <vector>
#include <iostream>
int majorityElement(std::vector<int>& nums) {
    int count = 0;
    int candidate = 0;
    for (int num : nums) {
        if (count == 0) {
            candidate = num;
        }
        count += (num == candidate) ? 1 : -1;
    }
    count = 0;
    for (int num : nums) {
        if (num == candidate) {
            count++;
        }
    }

    if (count > nums.size() / 2) {
        return candidate;
    }
    return -1;
}

int main() {
    std::vector<int> nums = {2, 2, 1, 1, 1, 2, 2};
    std::cout << "Majority Element: " << majorityElement(nums)
        << std::endl;
    return 0;
}
```

```
Majority Element: 2


=== Code Execution Successful ===
```



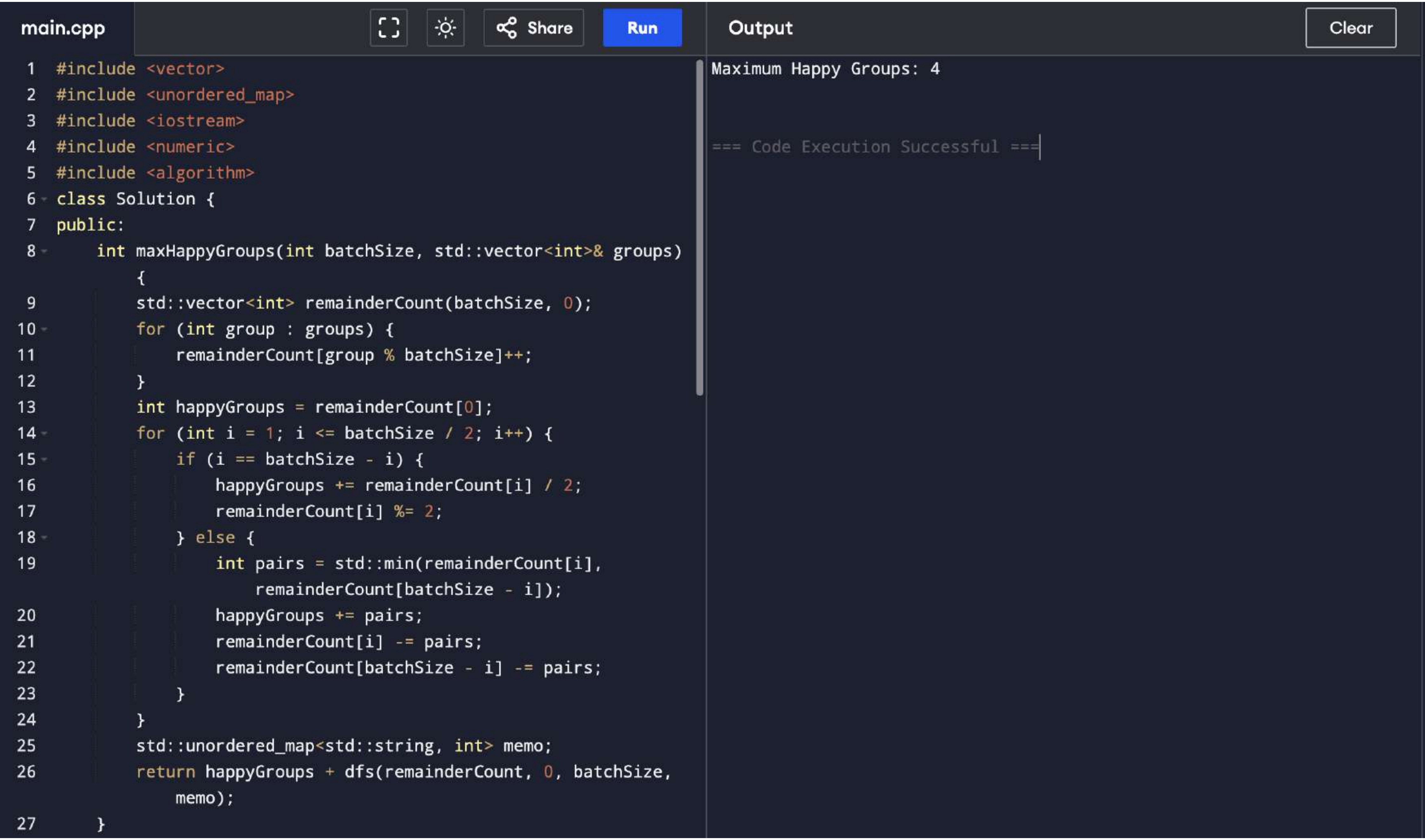
```cpp
#include <vector>
#include <iostream>
std::vector<std::vector<int>> generate(int numRows) {
    std::vector<std::vector<int>> triangle;
    for (int i = 0; i < numRows; i++) {
        std::vector<int> row(i + 1, 1);
        for (int j = 1; j < i; j++) {
            row[j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
        }
       triangle.push_back(row);
    }
 return triangle;
}

int main() {
    int numRows = 5;
    std::vector<std::vector<int>> result = generate(numRows);

    // Print Pascal's Triangle
    for (const auto& row : result) {
        for (int val : row) {
            std::cout << val << " ";
        }
        std::cout << std::endl;
    }

    return 0;
}

```

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1


=== Code Execution Successful ===
```



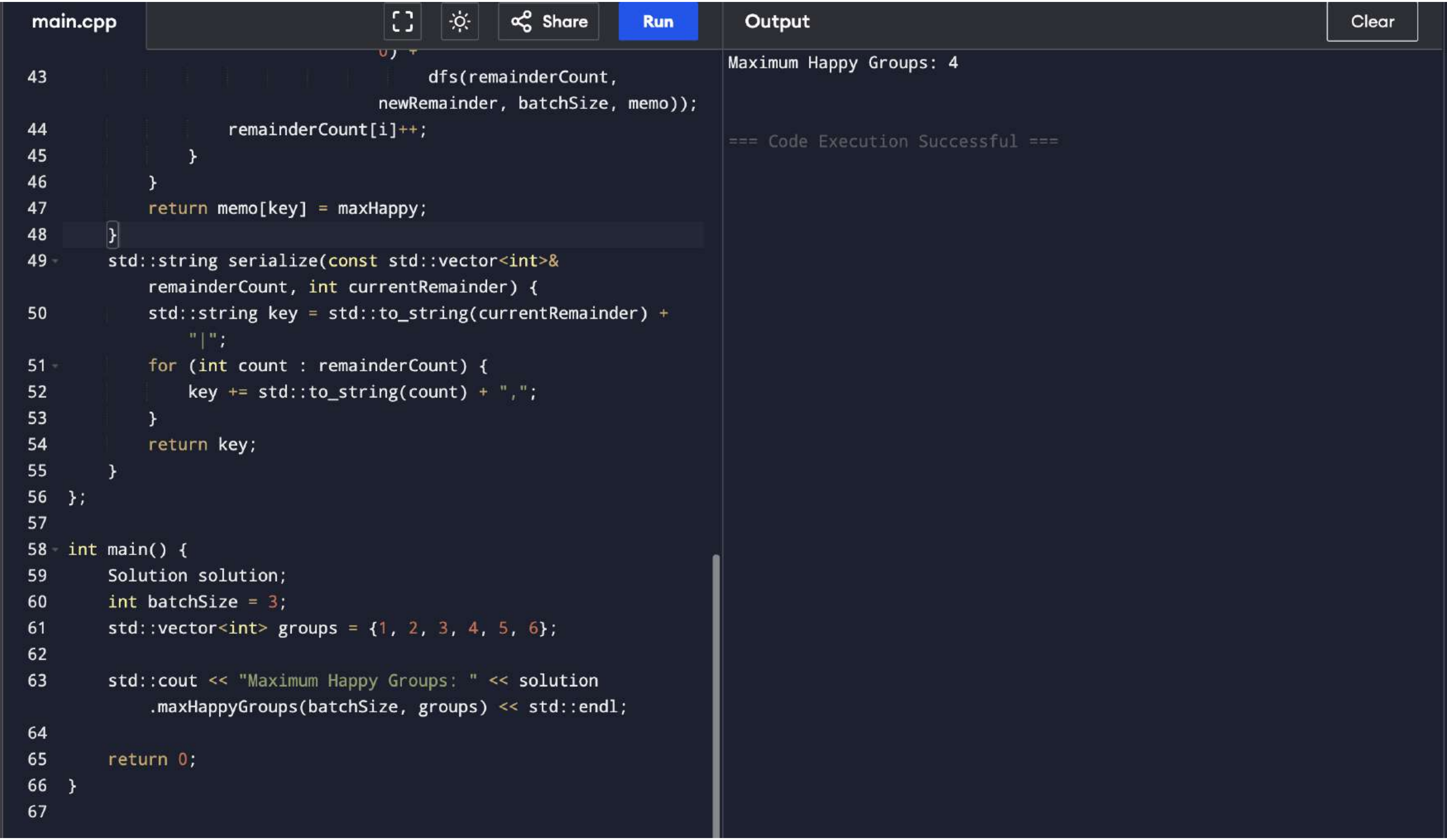
```cpp
#include <vector>
#include <iostream>
#include <algorithm>

int maxArea(std::vector<int>& height) {
    int left = 0;
    int right = height.size() - 1;
    int maxArea = 0;
    while (left < right) {
        int width = right - left;
        int currentHeight = std::min(height[left], height[right]);
        int currentArea = width * currentHeight;
        maxArea = std::max(maxArea, currentArea);
        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }
 return maxArea;
}
int main() {
    std::vector<int> height = {1, 8, 6, 2, 5, 4, 8, 3, 7};
    std::cout << "Maximum Area: " << maxArea(height) << std::endl;
    return 0;
}
```

```
Maximum Area: 49


=== Code Execution Successful ===
```



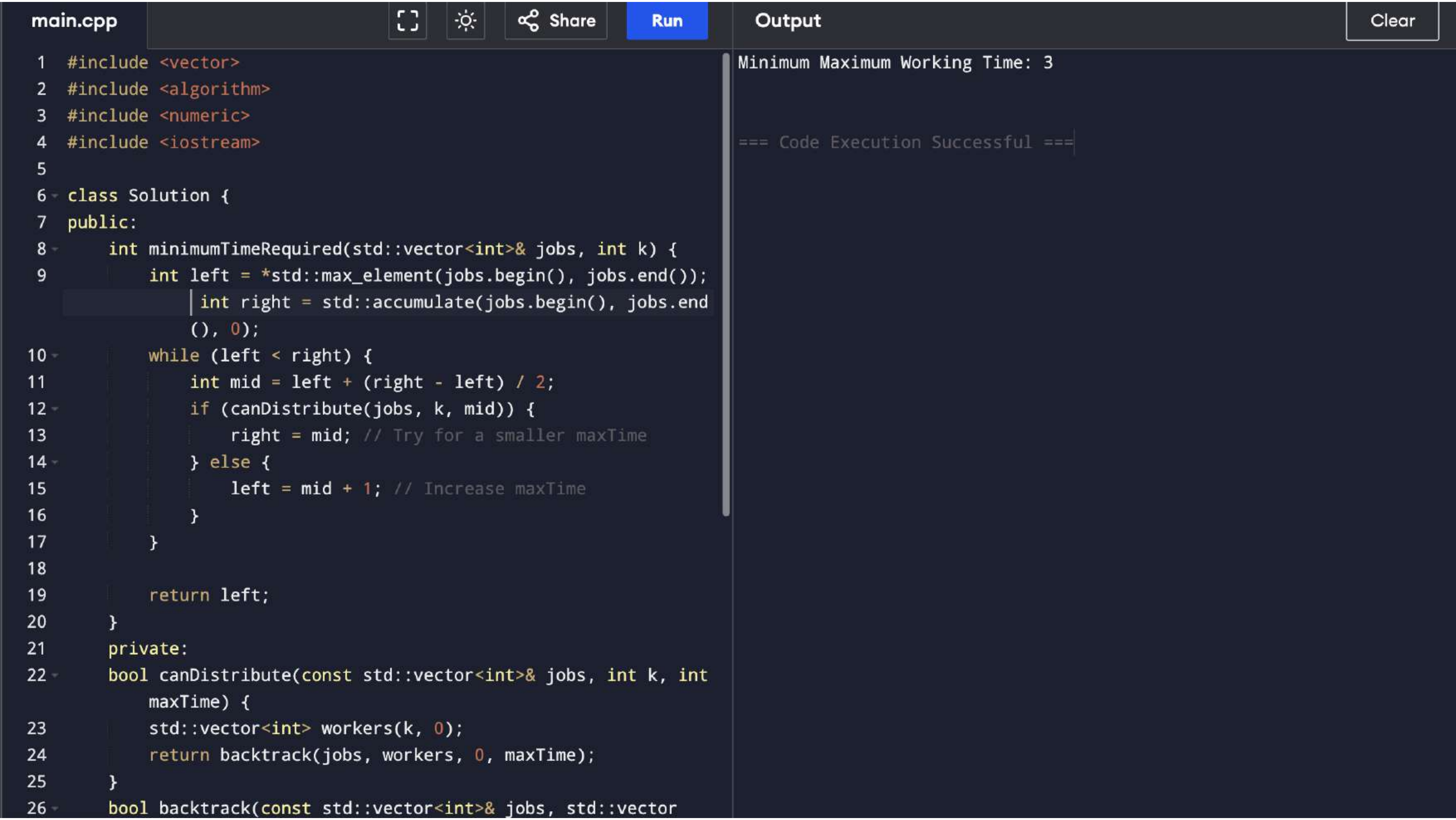
```cpp
#include <vector>
#include <unordered_map>
#include <iostream>
#include <numeric>
#include <algorithm>
class Solution {
public:
    int maxHappyGroups(int batchSize, std::vector<int>& groups) {
        std::vector<int> remainderCount(batchSize, 0);
        for (int group : groups) {
            remainderCount[group % batchSize]++;
        }
        int happyGroups = remainderCount[0];
        for (int i = 1; i <= batchSize / 2; i++) {
            if (i == batchSize - i) {
                happyGroups += remainderCount[i] / 2;
                remainderCount[i] %= 2;
            } else {
                int pairs = std::min(remainderCount[i], remainderCount[batchSize - i]);
                happyGroups += pairs;
                remainderCount[i] -= pairs;
                remainderCount[batchSize - i] -= pairs;
            }
        }
        std::unordered_map<std::string, int> memo;
        return happyGroups + dfs(remainderCount, 0, batchSize, memo);
    }
```

```
Maximum Happy Groups: 4


=== Code Execution Successful ===
```



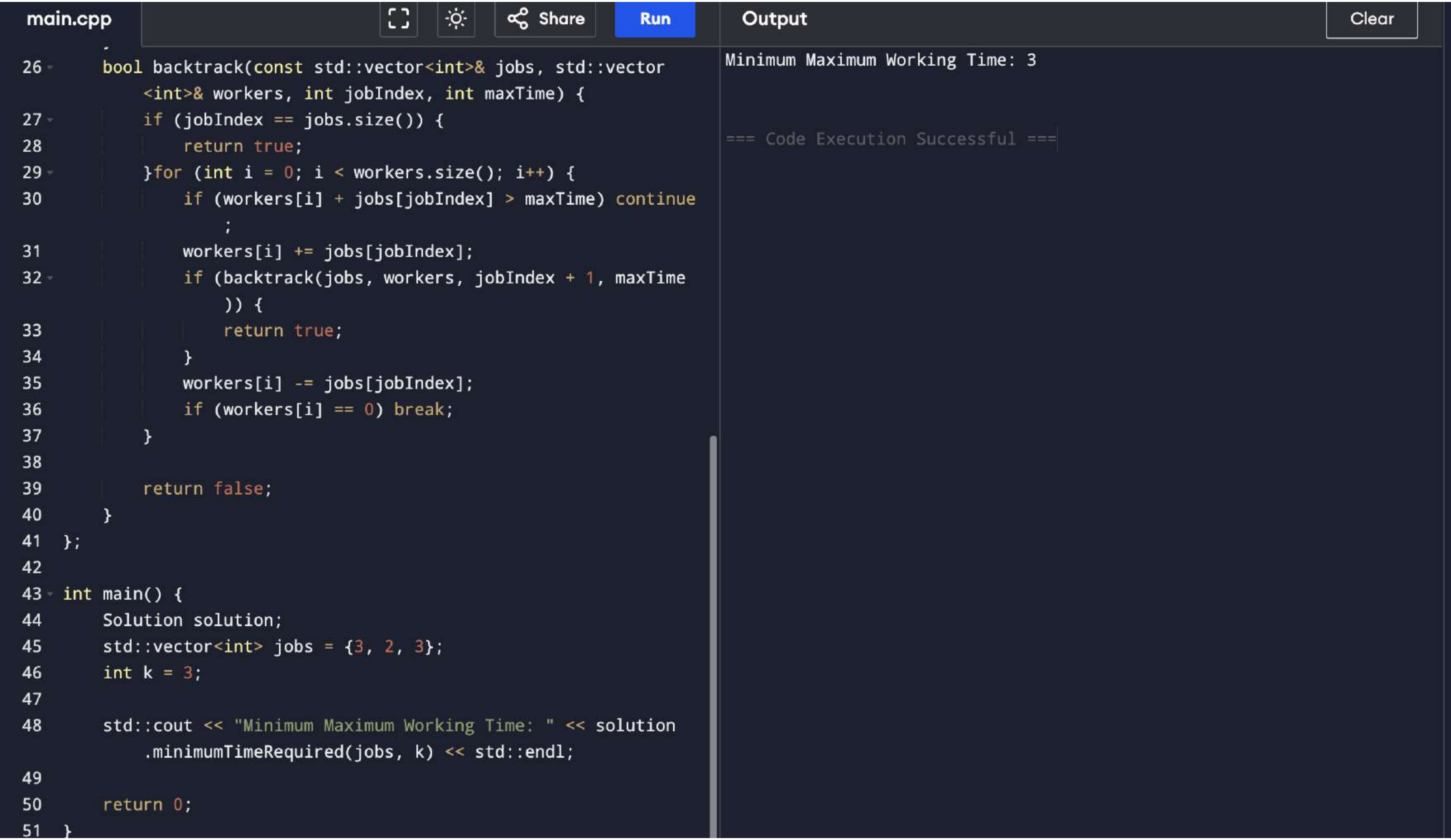
```cpp
                                0) +
                                    dfs(remainderCount,
                                newRemainder, batchSize, memo));
                remainderCount[i]++;
            }
        }
        return memo[key] = maxHappy;
    }
    std::string serialize(const std::vector<int>&
        remainderCount, int currentRemainder) {
        std::string key = std::to_string(currentRemainder) +
            "|";
        for (int count : remainderCount) {
            key += std::to_string(count) + ",";
        }
        return key;
    }
};

int main() {
    Solution solution;
    int batchSize = 3;
    std::vector<int> groups = {1, 2, 3, 4, 5, 6};

    std::cout << "Maximum Happy Groups: " << solution
        .maxHappyGroups(batchSize, groups) << std::endl;

    return 0;
}
```

```
Maximum Happy Groups: 4


=== Code Execution Successful ===
```



```cpp
#include <vector>
#include <algorithm>
#include <numeric>
#include <iostream>

class Solution {
public:
    int minimumTimeRequired(std::vector<int>& jobs, int k) {
        int left = *std::max_element(jobs.begin(), jobs.end());
            int right = std::accumulate(jobs.begin(), jobs.end(), 0);
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (canDistribute(jobs, k, mid)) {
                right = mid; // Try for a smaller maxTime
            } else {
                left = mid + 1; // Increase maxTime
            }
        }

        return left;
    }
    private:
    bool canDistribute(const std::vector<int>& jobs, int k, int maxTime) {
        std::vector<int> workers(k, 0);
        return backtrack(jobs, workers, 0, maxTime);
    }
    bool backtrack(const std::vector<int>& jobs, std::vector
```

```
Minimum Maximum Working Time: 3


=== Code Execution Successful ===
```



```cpp
    bool backtrack(const std::vector<int>& jobs, std::vector<int>& workers, int jobIndex, int maxTime) {
        if (jobIndex == jobs.size()) {
            return true;
        }for (int i = 0; i < workers.size(); i++) {
            if (workers[i] + jobs[jobIndex] > maxTime) continue;
            workers[i] += jobs[jobIndex];
            if (backtrack(jobs, workers, jobIndex + 1, maxTime)) {
                return true;
            }
            workers[i] -= jobs[jobIndex];
            if (workers[i] == 0) break;
        }

        return false;
    }
};

int main() {
    Solution solution;
    std::vector<int> jobs = {3, 2, 3};
    int k = 3;

    std::cout << "Minimum Maximum Working Time: " << solution.minimumTimeRequired(jobs, k) << std::endl;

    return 0;
}
```

```
Minimum Maximum Working Time: 3


=== Code Execution Successful ===
```