WINTER WINNING DOMAIN CAMP 2024

ASSIGNMENT DAY 2

1)You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]). Find two lines that together with the x-axis form a container, such that the container contains the most water. Return the maximum amount of water a container can store. Notice that you may not slant the container.

Example 1:

Input: height = [1,8,6,2,5,4,8,3,7]

Output: 49

Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

Example 2:

Input: height = [1,1]

Output: 1

Constraints:

n == height.length

2 <= n <= 105

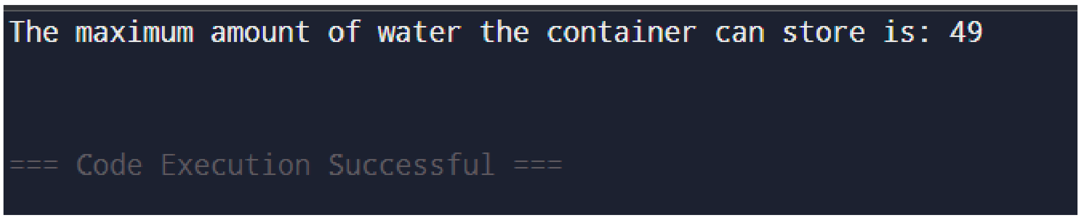
0 <= height[i] <= 104

SOLUTION:

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

class Solution {
```

```
public:
    int maxArea(vector<int>& height) {
        int left = 0;  // Start pointer
        int right = height.size() - 1;
        int maxWater = 0;

        while (left < right) {
            int width = right - left;
            int currentHeight = min(height[left], height[right]);
            int currentArea = width * currentHeight;

            // Update the maximum area found so far
            maxWater = max(maxWater, currentArea);

            // Move the pointer with the smaller height inward
            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }
        return maxWater;
    }
};
int main() {
    vector<int> height = {1, 8, 6, 2, 5, 4, 8, 3, 7};
    Solution solution;
    int result = solution.maxArea(height);
    cout << "The maximum amount of water the container can store is: " <<
result << endl;
    return 0;
}
```

OUTPUT:



```
The maximum amount of water the container can store is: 49


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

2) Given an array nums of size n, return the majority element.

The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

Example 1:

Input: nums = [3,2,3]

Output: 3

Example 2:

Input: nums = [2,2,1,1,1,2,2]

Output: 2

Constraints:

n == nums.length

1 <= n <= 5 * 104

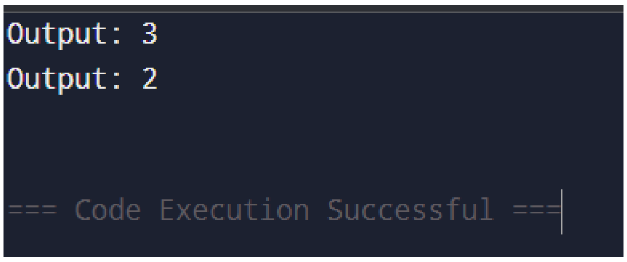
-109 <= nums[i] <= 109

SOLUTION:

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

```
int majorityElement(vector<int>& nums) {
    int candidate = nums[0], count = 0;

    for (int num : nums) {
        if (count == 0) {
            candidate = num;
        }
        count += (num == candidate) ? 1 : -1;
    }

    return candidate;
}

int main() {
    vector<int> nums1 = {3, 2, 3};
    cout << "Output: " << majorityElement(nums1) << endl;
    vector<int> nums2 = {2, 2, 1, 1, 1, 2, 2};
    cout << "Output: " << majorityElement(nums2) << endl;
    return 0;
}
```

OUTPUT:



```
Output: 3
Output: 2


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

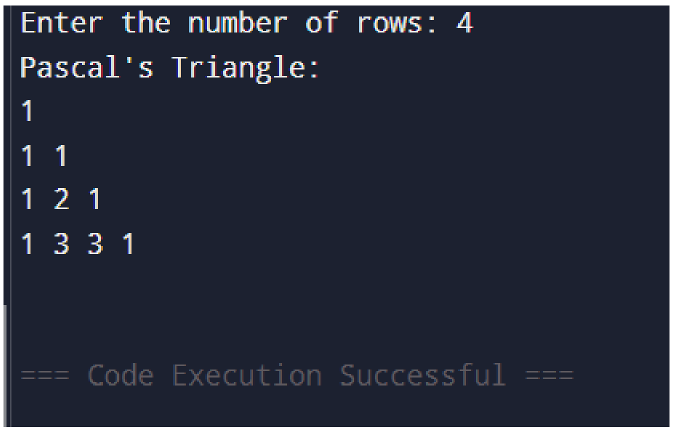
3) Given an integer numRows, return the first numRows of Pascal's triangle.

In Pascal's triangle, each number is the sum of the two numbers directly above it as shown:

SOLUTION:

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

vector<vector<int>> generate(int numRows) {
    vector<vector<int>> pascalTriangle;

    for (int i = 0; i < numRows; i++) {
        vector<int> row(i + 1, 1); // Initialize a row with all 1s
        for (int j = 1; j < i; j++) {
            row[j] = pascalTriangle[i - 1][j - 1] + pascalTriangle[i - 1][j];
        }
        pascalTriangle.push_back(row);
    }

    return pascalTriangle;
}

int main() {
    int numRows;
    cout << "Enter the number of rows: ";
    cin >> numRows;
    vector<vector<int>> result = generate(numRows);
    cout << "Pascal's Triangle:" << endl;

    for (const auto& row : result) {
        for (int num : row) {
            cout << num << " ";
        }
        cout << endl;
    }

    return 0;
}
```

OUTPUT:



```
Enter the number of rows: 4
Pascal's Triangle:
1
1 1
1 2 1
1 3 3 1


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

4) **Question: Maximum Number of Groups Getting Fresh Donuts**

There is a donuts shop that bakes donuts in batches of batchSize. They have a rule where they must serve all of the donuts of a batch before serving any donuts of the next batch. You are given an integer batchSize and an integer array groups, where groups[i] denotes that there is a group of groups[i] customers that will visit the shop. Each customer will get exactly one donut.

When a group visits the shop, all customers of the group must be served before serving any of the following groups. A group will be happy if they all get fresh donuts. That is, the first customer of the group does not receive a donut that was left over from the previous group.

You can freely rearrange the ordering of the groups. Return the maximum possible number of happy groups after rearranging the groups.

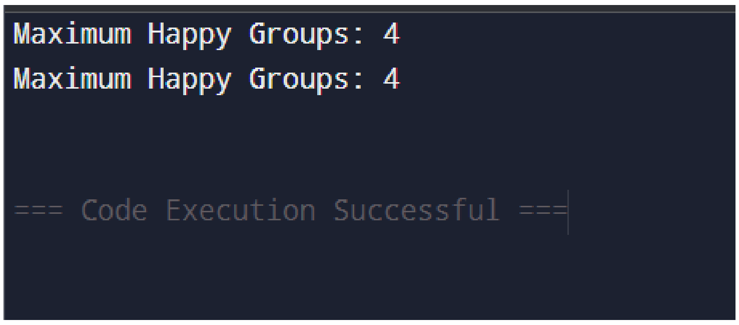
SOLUTION:

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <numeric>
using namespace std;

class Solution {
public:
    int maxHappyGroups(int batchSize, vector<int>& groups) {
        vector<int> remainders(batchSize, 0);
        int happyGroups = 0;

        // Count groups that are already happy
        for (int group : groups) {
            int remainder = group % batchSize;
            if (remainder == 0) {
                happyGroups++;
            } else {
                remainders[remainder]++;
            }
        }

        // Pair remainders to maximize happy groups
        for (int i = 1; i < batchSize; i++) {
            int complement = batchSize - i;
            if (i == complement) {
                happyGroups += remainders[i] / 2;
                remainders[i] %= 2;
            } else if (i < complement) {
                int pairs = min(remainders[i], remainders[complement]);
                happyGroups += pairs;
                remainders[i] -= pairs;
                remainders[complement] -= pairs;
            }
        }

        // Use dynamic programming to handle leftovers
        unordered_map<int, int> memo;
        happyGroups += dfs(remainders, batchSize, 0, memo);
        return happyGroups;
    }

private:
    int dfs(vector<int>& remainders, int batchSize, int currentRemainder,
unordered_map<int, int>& memo) {
        int key = hashKey(remainders, currentRemainder);
        if (memo.count(key)) return memo[key];

        int maxGroups = 0;
        for (int i = 1; i < batchSize; i++) {
            if (remainders[i] > 0) {
                remainders[i]--;
                int nextRemainder = (currentRemainder + i) % batchSize;
                maxGroups = max(maxGroups, ((nextRemainder == 0) ? 1 : 0) +
dfs(remainders, batchSize, nextRemainder, memo));
```

```
                remainders[i]++;
            }
        }

        memo[key] = maxGroups;
        return maxGroups;
    }

    int hashKey(const vector<int>& remainders, int currentRemainder) {
        int hash = currentRemainder;
        for (int rem : remainders) {
            hash = hash * 31 + rem;
        }
        return hash;
    }
};

int main() {
    Solution solution;

    // Example 1
    int batchSize1 = 3;
    vector<int> groups1 = {1, 2, 3, 4, 5, 6};
    cout << "Maximum Happy Groups: " <<
solution.maxHappyGroups(batchSize1, groups1) << endl;

    // Example 2
    int batchSize2 = 4;
    vector<int> groups2 = {4, 4, 4, 4};
    cout << "Maximum Happy Groups: " <<
solution.maxHappyGroups(batchSize2, groups2) << endl;
```

```
    return 0;
}
```

OUTPUT:



```
Maximum Happy Groups: 4
Maximum Happy Groups: 4


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

5) You are given an integer array jobs, where jobs[i] is the amount of time it takes to complete the ith job.There are k workers that you can assign jobs to. Each job should be assigned to exactly one worker. The working time of a worker is the sum of the time it takes to complete all jobs assigned to them. Your goal is to devise an optimal assignment such that the maximum working time of any worker is minimized.

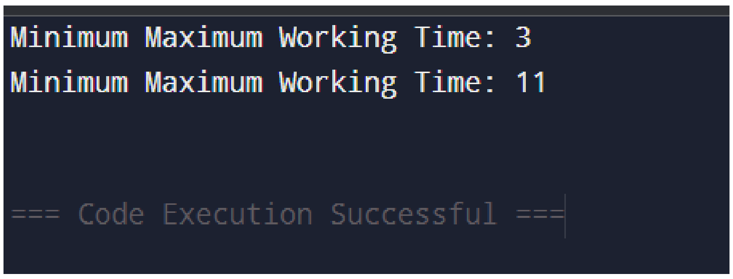
Return the minimum possible maximum working time of any assignment.

SOLUTION:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include<climits>
using namespace std;

class Solution {
public:
    int minimumTimeRequired(vector<int>& jobs, int k) {
        int n = jobs.size();
        vector<int> workers(k, 0);
        int result = INT_MAX;
        sort(jobs.rbegin(), jobs.rend());
        backtrack(jobs, workers, 0, k, result);
        return result;
    }
private:
    void backtrack(vector<int>& jobs, vector<int>& workers, int jobIndex, int k,
int& result) {
        if (jobIndex == jobs.size()) {
            result = min(result, *max_element(workers.begin(), workers.end()));
            return;
        }
        for (int i = 0; i < k; i++) {
            workers[i] += jobs[jobIndex];
            if (workers[i] < result) {
                backtrack(jobs, workers, jobIndex + 1, k, result);
            }
            workers[i] -= jobs[jobIndex];
            if (workers[i] == 0) break;
        }
    }
};
int main() {
    Solution solution;
    vector<int> jobs1 = {3, 2, 3};
    int k1 = 3;
    cout << "Minimum Maximum Working Time: " <<
solution.minimumTimeRequired(jobs1, k1) << endl;
    vector<int> jobs2 = {1, 2, 4, 7, 8};
    int k2 = 2;
    cout << "Minimum Maximum Working Time: " <<
solution.minimumTimeRequired(jobs2, k2) << endl;
    return 0;
```

```
}
```

OUTPUT:



```
Minimum Maximum Working Time: 3
Minimum Maximum Working Time: 11


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