



Department of Computer Science and Engineering

Name:- Vikash Ranjan Kumar UID:- 22BCS11322 Section:- 22KPIT-901-B

DAY-2

Q.1. Given an array nums of size n, return the majority element.

The majority element is the element that appears more than $\lfloor n / 2 \rfloor$ 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`

Program Code:-

```
#include <iostream>

#include <vector>

using namespace std;

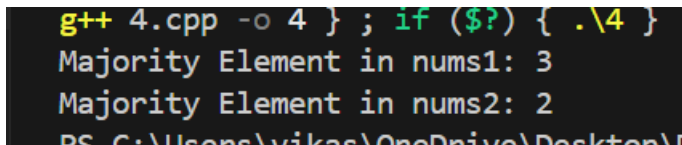
int majorityElement(vector<int>& nums) {
    int count = 0;
    int candidate = 0;
    for (int num : nums) {
        if (count == 0) {
            candidate = num;
        }
        count += (num == candidate) ? 1 : -1;
    }
    return candidate;
}
```

```

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

```

Output:-



```

g++ 4.cpp -o 4 } ; if ($?) { .\4 }
Majority Element in nums1: 3
Majority Element in nums2: 2
PS C:\Users\vikas\OneDrive\Desktop\B

```

Q.2 Given an integer numRows, return the first numRows of Pascal's triangle.

Program Code:-

```

#include <iostream>

#include <vector>

using namespace std;

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

    for (int i = 0; i < numRows; ++i) {
        vector<int> row(i + 1, 1);
    }
}

```

```

        for (int j = 1; j < i; ++j) {
            row[j] = pascalTriangle[i - 1][j - 1] + pascalTriangle[i - 1][j];
        }
        pascalTriangle.push_back(row);
    }
    return pascalTriangle;
}

void printPascalTriangle(const vector<vector<int>>& triangle) {
    for (const auto& row : triangle) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
}

int main() {
    int numRows;

    cout << "Enter the number of rows for Pascal's Triangle: ";

    cin >> numRows;

    vector<vector<int>> pascalTriangle = generatePascalTriangle(numRows);

    cout << "Pascal's Triangle:\n";

    printPascalTriangle(pascalTriangle);
}

```

```
    return 0;  
}
```

Output:-

```
Enter the number of rows for Pascal's Triangle: 5  
Pascal's Triangle:  
1  
1 1  
1 2 1  
1 3 3 1  
1 4 6 4 1
```

Q.3. 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.

Program Code:-

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

```
using namespace std;

int maxArea(vector<int>& height) {
    int left = 0, right = height.size() - 1;

    int maxWater = 0;

    while (left < right) {
        int width = right - left;

        int currentHeight = min(height[left], height[right]);

        int area = width * currentHeight;

        maxWater = max(maxWater, area);

        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }

    return maxWater;
}

int main() {
    vector<int> height = {1, 8, 6, 2, 5, 4, 8, 3, 7};

    cout << "Maximum area of water the container can store: " << maxArea(height)
    << endl;
```

```
    return 0;  
}
```

Output:-

```
Maximum area of water the container  
can store: 49
```

Q.4. 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.

Program Code:-

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

```

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

        for (int g : groups) {
            modCount[g % batchSize]++;
        }

        int happyGroups = modCount[0];
        for (int i = 1; i <= batchSize / 2; ++i) {
            if (i == batchSize - i) {
                happyGroups += modCount[i] / 2;
            } else {
                int pairs = min(modCount[i], modCount[batchSize - i]);
                happyGroups += pairs;
                modCount[i] -= pairs;
                modCount[batchSize - i] -= pairs;
            }
        }

        return happyGroups + dfs(modCount, batchSize, 0);
    }

private:
    unordered_map<string, int> memo;

```



```

int dfs(vector<int>& modCount, int batchSize, int remainder) {
    string key = to_string(remainder) + "," + to_string_vector(modCount);
    if (memo.find(key) != memo.end()) return memo[key];
    int result = 0;
    bool hasRemaining = false;

    for (int i = 1; i < modCount.size(); ++i) {
        if (modCount[i] > 0) {
            hasRemaining = true;
            modCount[i]--;
            result = max(result, (remainder == 0 ? 1 : 0) + dfs(modCount, batchSize,
(remainder + i) % batchSize));
            modCount[i]++;
        }
    }

    if (!hasRemaining) {
        result = 0; // No remaining groups
    }

    memo[key] = result;
    return result;
}

string to_string_vector(const vector<int>& vec) {

```

```

        string result;

        for (int val : vec) {

            result += to_string(val) + ",";

        }

        return result;

    }

};

int main() {

    Solution solution;

    int batchSize = 3;

    vector<int> groups = {1, 2, 3, 4, 5, 6};

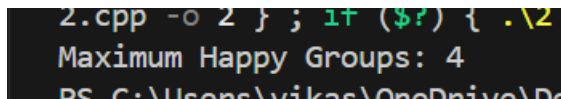
    cout << "Maximum Happy Groups: " << solution.maxHappyGroups(batchSize,
groups) << endl;

    return 0;

}

```

Output:-



```

2.cpp -o 2 } ; 1+ ($?) { .\2
Maximum Happy Groups: 4
PS C:\Users\vikas\OneDrive\De

```

Q.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.

Example 1:

Input: jobs = [3,2,3], k = 3

Output: 3

Explanation: By assigning each person one job, the maximum time is 3.

```
#include <iostream>
```

```
#include <vector>
```

```
#include <algorithm>
```

```
using namespace std;
```

```
class Solution {
```

```
public:
```

```
    int minimumWorkingTime(vector<int>& jobs, int k) {
```

```
        vector<int> workloads(k, 0);
```

```
        int result ;
```

```
        sort(jobs.rbegin(), jobs.rend());
```

```
        backtrack(jobs, 0, workloads, k, result);
```

```
        return result;
```

```
    }
```

```
private:
```

```
    void backtrack(vector<int>& jobs, int index, vector<int>& workloads, int k,  
int& result) {
```

```
        if (index == jobs.size()) {
```

```

        result = min(result, *max_element(workloads.begin(), workloads.end()));

        return;
    }

    int currentJob = jobs[index];

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

        if (workloads[i] + currentJob >= result) continue;

        if (i > 0 && workloads[i] == workloads[i - 1]) continue;

        workloads[i] += currentJob;

        backtrack(jobs, index + 1, workloads, k, result);

        workloads[i] -= currentJob;

        if (workloads[i] == 0) break;

    }

}

};

int main() {

    Solution solution;

    vector<int> jobs = {3, 2, 3};

    int k = 3;

    cout << "Minimum possible maximum working time: " <<
    solution.minimumWorkingTime(jobs, k) << endl;

    return 0;

}

```

Output:-

```
o 1 } ; if ($?) { .\1 }  
Minimum possible maximum working time: 3  
PS C:\Users\vikas\OneDrive\Desktop\Database
```