WINTER DOMAIN CAMP

NAME: Vikram Kumar **UID:22BCS12220**

SECTION: KPIT-901-B **DATE: 19/12/2024**

Q1. Return majority elements(Easy)
Sol.

```cpp
//return the majority elements that appears more than n/2 times.time complexity O(1).easy
#include <iostream>
#include <vector>
using namespace std;

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

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

OUTPUT:

```
Majority Element: 2


...Program finished
```

Q2. Pascal triangle(medium)
Sol.

```cpp
//pascal's triangle time coplexity On^2 medium
#include <iostream>
#include <vector>
using namespace std;
vector<vector<int>> generatePascalsTriangle(int numRows) {
    vector<vector<int>> triangle;
    for (int i = 0; i < numRows; ++i) {
        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;}
void printTriangle(const vector<vector<int>>& triangle) {
    for (const auto& row : triangle) {
        for (int num : row) {
            cout << num << " ";
        }
        cout << endl;
    }
}
int main() {
    int numRows = 6;
    vector<vector<int>> result = generatePascalsTriangle(numRows);
    cout << "Pascal's Triangle with " << numRows << " rows:" <<endl;
    printTriangle(result);
    return 0;}

```

OUTPUT:

```
Pascal's Triangle with 6 rows:
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

Q3. Container with most water(medium)

Sol.

```cpp
//container with most water //medium.time com o(n).
#include <iostream>
#include <vector>
using namespace std;
int maxArea(vector<int>& height) {
    int left = 0, right = height.size() - 1;
    int maxWater = 0;
    while (left < right) {
        // Calculate the area between the two lines
        int width = right - left;
        int currentArea = min(height[left], height[right]) * width;
        maxWater = max(maxWater, currentArea);
        // Move the pointer pointing to the shorter line
        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 Water Container: " << maxArea(height) << endl;
    return 0;
}
```

OUTPUT:

```
Maximum Water Container: 49


=== Code Execution Successfu
```

Q4.Maximum no. of groups getting fresh donuts(hard)

Sol.

```cpp
// maximum no. of groups getting fresh donuts. hard.time complexity
//O(batchSize×(n/batchSize)^batchSize)
#include <iostream>
#include <vector>
#include <cstring>
using namespace std;

// Function to calculate the maximum happy groups
int dfs(int batchSize, vector<int>& remainders, int remainder, int
    dp[]) {
    int state = 0;
    for (int i = 0; i < remainders.size(); ++i) {
        state = state * 10 + remainders[i]; // Compress the state
            into an integer
    }
    state = state * batchSize + remainder; // Include the current
        remainder in the state
    if (dp[state] != -1) return dp[state];

    int maxHappy = 0;
    for (int i = 0; i < remainders.size(); ++i) {
        if (remainders[i] > 0) {
            remainders[i]--;
            int newRemainder = (remainder + i) % batchSize;
            maxHappy = max(maxHappy, dfs(batchSize, remainders,
                newRemainder, dp) + (newRemainder == 0 ? 1 : 0));
            remainders[i]++;
        }
    }
```

```
    return dp[state] = maxHappy;
}

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

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

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

        }
    }

    // Dynamic Programming Array
    int dp[1000000];
    memset(dp, -1, sizeof(dp));

    happyGroups += dfs(batchSize, remainders, 0, dp);
    return happyGroups;
}

int main() {
    int batchSize = 3;
    vector<int> groups = {1, 2, 3, 4, 5, 6};
    cout << "Maximum Happy Groups: " << maxHappyGroups(batchSize,
        groups) << endl;
    return 0;
}
```

OUTPUT:

```
Maximum Happy Groups: 4


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

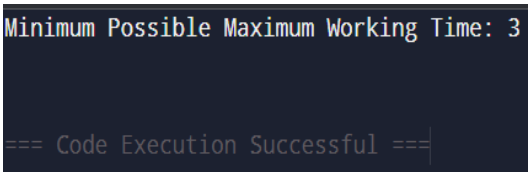
Q5. Find Minimum Time to Finish All Jobs.(very hard)

Sol.

```cpp
//Question 1. Find Minimum Time to Finish All Jobs.very hard.
//time complexity O(k^n)
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

bool canDistribute(vector<int>& jobs, vector<int>& workers, int idx
    int limit) {
    if (idx == jobs.size()) return true; // All jobs assigned

    for (int i = 0; i < workers.size(); ++i) {
        if (workers[i] + jobs[idx] > limit) continue; // Exceeds
            current limit
        workers[i] += jobs[idx]; // Assign the job to the worker
        if (canDistribute(jobs, workers, idx + 1, limit)) return
            true; // Recurse
        workers[i] -= jobs[idx]; // Backtrack

        // If a worker remains idle, no need to try assigning jobs
            to other idle workers
        if (workers[i] == 0) break;
    }
    return false;
}

int minimumTimeRequired(vector<int>& jobs, int k) {
    sort(jobs.rbegin(), jobs.rend()); // Sort jobs in descending
        order for better pruning
```

```cpp

int minimumTimeRequired(vector<int>& jobs, int k) {
    sort(jobs.rbegin(), jobs.rend()); // Sort jobs in descending order for better pruning
    int low = jobs[0]; // Minimum possible time (largest job)
    int high = accumulate(jobs.begin(), jobs.end(), 0); // Maximum possible time (all jobs assigned to one worker)

    while (low < high) {
        int mid = low + (high - low) / 2;
        vector<int> workers(k, 0); // Array to store worker loads
        if (canDistribute(jobs, workers, 0, mid)) {
            high = mid; // Try a smaller limit
        } else {
            low = mid + 1; // Increase the limit
        }
    }
    return low;
}

int main() {
    vector<int> jobs = {3, 2, 3};
    int k = 3;
    cout << "Minimum Possible Maximum Working Time: " << minimumTimeRequired(jobs, k) << endl;
    return 0;
}

```

OUTPUT:


Minimum Possible Maximum Working Time: 3
=== Code Execution Successful ===