main.cpp

Share Run

```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;
}
```

Output

Clear

```
Majority Element: 2


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

main.cpp

Share Run

```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;
}

```

Output Clear

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


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

main.cpp

Share Run

```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;
}
```

Output Clear

```
Maximum Area: 49


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

main.cpp

Share Run

```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);
    }
```

Output

Clear

```
Maximum Happy Groups: 4


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

main.cpp

Share Run

```
                                0) +
43                                  dfs(remainderCount,
                                newRemainder, batchSize, memo));
44                  remainderCount[i]++;
45              }
46          }
47          return memo[key] = maxHappy;
48      }
49      std::string serialize(const std::vector<int>&
            remainderCount, int currentRemainder) {
50          std::string key = std::to_string(currentRemainder) +
                "|";
51          for (int count : remainderCount) {
52              key += std::to_string(count) + ",";
53          }
54          return key;
55      }
56  };
57
58  int main() {
59      Solution solution;
60      int batchSize = 3;
61      std::vector<int> groups = {1, 2, 3, 4, 5, 6};
62
63      std::cout << "Maximum Happy Groups: " << solution
            .maxHappyGroups(batchSize, groups) << std::endl;
64
65      return 0;
66  }
67
```

Output

Clear

```
Maximum Happy Groups: 4


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

main.cpp

Share Run

```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
```

Output Clear

```
Minimum Maximum Working Time: 3


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

main.cpp

Share Run

```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;
}
```

Output

Clear

```
Minimum Maximum Working Time: 3


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

main.cpp

Share Run

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

int fibonacci(int n) {
    if (n <= 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

int main() {
    int n;
    cout << "Enter a number: ";
    cin >> n;

    int result = fibonacci(n);
    cout << "F(" << n << ") = " << result << endl;

    return 0;
}

```

Output Clear

```
Enter a number: 7
F(7) = 13


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

main.cpp

Share Run

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

// Definition for singly-linked list node.
struct ListNode {
    int val;
    ListNode* next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode* next) : val(x), next(next) {}
};

// Function to reverse the list iteratively
ListNode* reverseListIterative(ListNode* head) {
    ListNode* prev = nullptr;
    ListNode* current = head;
    while (current != nullptr) {
        ListNode* nextNode = current->next; // Store the next node
        current->next = prev;               // Reverse the link
        prev = current;                     // Move prev forward
        current = nextNode;                 // Move current forward
    }
    return prev; // New head of the reversed list
}

// Function to reverse the list recursively
ListNode* reverseListRecursive(ListNode* head) {
    if (head == nullptr || head->next == nullptr) {
```

Output Clear

```
Original List: 1 2 3 4 5
Reversed List (Iterative): 5 4 3 2 1
Reversed List (Recursive): 5 4 3 2 1


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

main.cpp

Share Run

```cpp
int main() {
    // Create an example list: 1 -> 2 -> 3 -> 4 -> 5
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new ListNode(5);

    cout << "Original List: ";
    printList(head);

    // Reverse iteratively
    ListNode* reversedIterative = reverseListIterative(head);
    cout << "Reversed List (Iterative): ";
    printList(reversedIterative);

    // Reset the list for recursive reversal
    head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new ListNode(5);

    // Reverse recursively
    ListNode* reversedRecursive = reverseListRecursive(head);
    cout << "Reversed List (Recursive): ";
    printList(reversedRecursive);

    return 0;
}
```

Output

Clear

```
Original List: 1 2 3 4 5
Reversed List (Iterative): 5 4 3 2 1
Reversed List (Recursive): 5 4 3 2 1


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

main.cpp

Share Run

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

// Function to implement wildcard matching
bool isMatch(string s, string p) {
    int m = s.size(), n = p.size();

    // Create a DP table where dp[i][j] means whether s[0..i-1] matches p[0..j-1]
    vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));

    // Base case: Empty string matches with empty pattern
    dp[0][0] = true;

    // Base case: Handle patterns with '*' at the start
    for (int j = 1; j <= n; j++) {
        if (p[j - 1] == '*') {
            dp[0][j] = dp[0][j - 1];
        }
    }

    // Fill the DP table
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (p[j - 1] == s[i - 1] || p[j - 1] == '?') {
                // Characters match, or '?' matches any single character
                dp[i][j] = dp[i - 1][j - 1];
```

Output

Clear

```
Enter input string: Ankita
Enter pattern: An
The pattern does not match the string.


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

main.cpp

Share Run

```
                character
                dp[i][j] = dp[i - 1][j - 1];
            } else if (p[j - 1] == '*') {
                // '*' matches any sequence: empty sequence or
                    one or more characters
                dp[i][j] = dp[i][j - 1] || dp[i - 1][j];
            }
        }
    }

    return dp[m][n];
}

int main() {
    string s, p;
    cout << "Enter input string: ";
    cin >> s;
    cout << "Enter pattern: ";
    cin >> p;

    if (isMatch(s, p)) {
        cout << "The pattern matches the string." << endl;
    } else {
        cout << "The pattern does not match the string." <<
            endl;
    }

    return 0;
}
```

Output Clear

```
Enter input string: Ankita
Enter pattern: An
The pattern does not match the string.


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

main.cpp

Share Run

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
string makeLargestSpecial(string s) {
    if (s.empty()) return "";

    vector<string> substrings;
    int balance = 0, start = 0;
    for (int i = 0; i < s.size(); ++i) {
        balance += (s[i] == '1') ? 1 : -1;
        if (balance == 0) {
            string substring = s.substr(start + 1, i - start -
                1);
            substrings.push_back('1' + makeLargestSpecial
                (substring) + '0');
            start = i + 1;
        }
    }
    sort(substrings.begin(), substrings.end(), greater<string
        >());
    string result = "";
    for (const string& sub : substrings) {
        result += sub;
    }

    return result;
}

```

Output Clear

```
Enter a special binary string: 110011000
Lexicographically largest special binary string: 11001100


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

main.cpp

Share Run

```
        if (balance == 0) {
            string substring = s.substr(start + 1, i - start - 1);
            substrings.push_back('1' + makeLargestSpecial(substring) + '0');
            start = i + 1;
        }
    }
    sort(substrings.begin(), substrings.end(), greater<string>());
    string result = "";
    for (const string& sub : substrings) {
        result += sub;
    }

    return result;
}

int main() {
    string s;
    cout << "Enter a special binary string: ";
    cin >> s;

    string result = makeLargestSpecial(s);
    cout << "Lexicographically largest special binary string: " << result << endl;

    return 0;
}

```

Output Clear

```
Enter a special binary string: 110011000
Lexicographically largest special binary string: 11001100


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

main.cpp

Share Run

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

struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(nullptr) {}
};

ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    ListNode* dummy = new ListNode(0);
    ListNode* curr = dummy;
    int carry = 0;

    while (l1 || l2 || carry) {
        int sum = carry;
        if (l1) {
            sum += l1->val;
            l1 = l1->next;
        }
        if (l2) {
            sum += l2->val;
            l2 = l2->next;
        }
        curr->next = new ListNode(sum % 10);
        carry = sum / 10;
        curr = curr->next;
    }

    return dummy->next;
```

Output

Clear

```
Sum: 7 0 8


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

main.cpp

Share Run

```
        carry = sum / 10;
        curr = curr->next;
    }

    return dummy->next;
}

void printList(ListNode* head) {
    while (head) {
        cout << head->val << " ";
        head = head->next;
    }
    cout << endl;
}

int main() {
    ListNode* l1 = new ListNode(2);
    l1->next = new ListNode(4);
    l1->next->next = new ListNode(3);

    ListNode* l2 = new ListNode(5);
    l2->next = new ListNode(6);
    l2->next->next = new ListNode(4);

    ListNode* result = addTwoNumbers(l1, l2);
    cout << "Sum: ";
    printList(result);

    return 0;
}
```

Output

Clear

```
Sum: 7 0 8


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