

main.cpp



Share

Run

Output

Clear

```
1  #include <vector>
2  #include <iostream>
3  int majorityElement(std::vector<int>& nums) {
4      int count = 0;
5      int candidate = 0;
6      for (int num : nums) {
7          if (count == 0) {
8              candidate = num;
9          }
10         count += (num == candidate) ? 1 : -1;
11     }
12     count = 0;
13     for (int num : nums) {
14         if (num == candidate) {
15             count++;
16         }
17     }
18
19     if (count > nums.size() / 2) {
20         return candidate;
21     }
22     return -1;
23 }
24
25 int main() {
26     std::vector<int> nums = {2, 2, 1, 1, 1, 2, 2};
27     std::cout << "Majority Element: " << majorityElement(nums)
28               << std::endl;
29     return 0;
30 }
```

Majority Element: 2

=== Code Execution Successful ===

main.cpp



Share

Run

Output

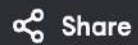
Clear

```
1  #include <vector>
2  #include <iostream>
3  std::vector<std::vector<int>> generate(int numRows) {
4      std::vector<std::vector<int>> triangle;
5      for (int i = 0; i < numRows; i++) {
6          std::vector<int> row(i + 1, 1);
7          for (int j = 1; j < i; j++) {
8              row[j] = triangle[i - 1][j - 1] + triangle[i -
9                  1][j];
10         }
11         triangle.push_back(row);
12     }
13     return triangle;
14 }
15 int main() {
16     int numRows = 5;
17     std::vector<std::vector<int>> result = generate(numRows);
18
19     // Print Pascal's Triangle
20     for (const auto& row : result) {
21         for (int val : row) {
22             std::cout << val << " ";
23         }
24         std::cout << std::endl;
25     }
26
27     return 0;
28 }
29
```

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

=== Code Execution Successful ===

main.cpp



Share

Run

Output

Clear

```
1 #include <vector>
2 #include <iostream>
3 #include <algorithm>
4
5 int maxArea(std::vector<int>& height) {
6     int left = 0;
7     int right = height.size() - 1;
8     int maxArea = 0;
9     while (left < right) {
10         int width = right - left;
11         int currentHeight = std::min(height[left], height[right]
            );
12         int currentArea = width * currentHeight;
13         maxArea = std::max(maxArea, currentArea);
14         if (height[left] < height[right]) {
15             left++;
16         } else {
17             right--;
18         }
19     }
20     return maxArea;
21 }
22 int main() {
23     std::vector<int> height = {1, 8, 6, 2, 5, 4, 8, 3, 7};
24     std::cout << "Maximum Area: " << maxArea(height) << std
        ::endl;
25     return 0;
26 }
27
```

Maximum Area: 49

=== Code Execution Successful ===

main.cpp



Run

Output

Clear

```
1  #include <vector>
2  #include <unordered_map>
3  #include <iostream>
4  #include <numeric>
5  #include <algorithm>
6  class Solution {
7  public:
8      int maxHappyGroups(int batchSize, std::vector<int>& groups)
9      {
10         std::vector<int> remainderCount(batchSize, 0);
11         for (int group : groups) {
12             remainderCount[group % batchSize]++;
13         }
14         int happyGroups = remainderCount[0];
15         for (int i = 1; i <= batchSize / 2; i++) {
16             if (i == batchSize - i) {
17                 happyGroups += remainderCount[i] / 2;
18                 remainderCount[i] %= 2;
19             } else {
20                 int pairs = std::min(remainderCount[i],
21                                     remainderCount[batchSize - i]);
22                 happyGroups += pairs;
23                 remainderCount[i] -= pairs;
24                 remainderCount[batchSize - i] -= pairs;
25             }
26         }
27         std::unordered_map<std::string, int> memo;
28         return happyGroups + dfs(remainderCount, 0, batchSize,
29                                 memo);
30     }
31 }
```

Maximum Happy Groups: 4

=== Code Execution Successful ===

main.cpp



Share

Run

Output

Clear

```
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
64               .maxHappyGroups(batchSize, groups) << std::endl;
65
66     return 0;
67 }
```

Maximum Happy Groups: 4

=== Code Execution Successful ===



main.cpp



Share

Run

Output

Clear

```
1 #include <vector>
2 #include <algorithm>
3 #include <numeric>
4 #include <iostream>
5
6 class Solution {
7 public:
8     int minimumTimeRequired(std::vector<int>& jobs, int k) {
9         int left = *std::max_element(jobs.begin(), jobs.end());
10         int right = std::accumulate(jobs.begin(), jobs.end(), 0);
11         while (left < right) {
12             int mid = left + (right - left) / 2;
13             if (canDistribute(jobs, k, mid)) {
14                 right = mid; // Try for a smaller maxTime
15             } else {
16                 left = mid + 1; // Increase maxTime
17             }
18         }
19         return left;
20     }
21 private:
22     bool canDistribute(const std::vector<int>& jobs, int k, int
maxTime) {
23         std::vector<int> workers(k, 0);
24         return backtrack(jobs, workers, 0, maxTime);
25     }
26     bool backtrack(const std::vector<int>& jobs, std::vector
```

Minimum Maximum Working Time: 3

=== Code Execution Successful ===

main.cpp



Share

Run

Output

Clear

```
26- bool backtrack(const std::vector<int>& jobs, std::vector<int>& workers, int jobIndex, int maxTime) {
27-     if (jobIndex == jobs.size()) {
28-         return true;
29-     } for (int i = 0; i < workers.size(); i++) {
30-         if (workers[i] + jobs[jobIndex] > maxTime) continue
31-         ;
32-         workers[i] += jobs[jobIndex];
33-         if (backtrack(jobs, workers, jobIndex + 1, maxTime
34-             )) {
35-             return true;
36-         }
37-         workers[i] -= jobs[jobIndex];
38-         if (workers[i] == 0) break;
39-     }
40-     return false;
41- };
42-
43- int main() {
44-     Solution solution;
45-     std::vector<int> jobs = {3, 2, 3};
46-     int k = 3;
47-
48-     std::cout << "Minimum Maximum Working Time: " << solution
49-         .minimumTimeRequired(jobs, k) << std::endl;
50-
51-     return 0;
52- }
```

Minimum Maximum Working Time: 3

=== Code Execution Successful ===

main.cpp



Share

Run

Output

Clear

```
1 #include <iostream>
2 using namespace std;
3
4 int fibonacci(int n) {
5     if (n <= 0) {
6         return 0;
7     } else if (n == 1) {
8         return 1;
9     } else {
10        return fibonacci(n - 1) + fibonacci(n - 2);
11    }
12 }
13
14 int main() {
15     int n;
16     cout << "Enter a number: ";
17     cin >> n;
18
19     int result = fibonacci(n);
20     cout << "F(" << n << ") = " << result << endl;
21
22     return 0;
23 }
24
```

Enter a number: 7  
F(7) = 13

=== Code Execution Successful ===



main.cpp



Share

Run

Output

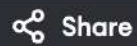
Clear

```
1  #include <iostream>
2  using namespace std;
3
4  // Definition for singly-linked list node.
5  struct ListNode {
6      int val;
7      ListNode* next;
8      ListNode() : val(0), next(nullptr) {}
9      ListNode(int x) : val(x), next(nullptr) {}
10     ListNode(int x, ListNode* next) : val(x), next(next) {}
11 };
12
13 // Function to reverse the list iteratively
14 ListNode* reverseListIterative(ListNode* head) {
15     ListNode* prev = nullptr;
16     ListNode* current = head;
17     while (current != nullptr) {
18         ListNode* nextNode = current->next; // Store the next
19         // node
20         current->next = prev;                // Reverse the link
21         prev = current;                     // Move prev forward
22         current = nextNode;                 // Move current
23         // forward
24     }
25     return prev; // New head of the reversed list
26 }
27
28 // Function to reverse the list recursively
29 ListNode* reverseListRecursive(ListNode* head) {
30     if (head == nullptr || head->next == nullptr) {
```

```
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

Output

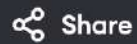
Clear

```
42 - int main() {
43     // Create an example list: 1 -> 2 -> 3 -> 4 -> 5
44     ListNode* head = new ListNode(1);
45     head->next = new ListNode(2);
46     head->next->next = new ListNode(3);
47     head->next->next->next = new ListNode(4);
48     head->next->next->next->next = new ListNode(5);
49
50     cout << "Original List: ";
51     printList(head);
52
53     // Reverse iteratively
54     ListNode* reversedIterative = reverseListIterative(head);
55     cout << "Reversed List (Iterative): ";
56     printList(reversedIterative);
57
58     // Reset the list for recursive reversal
59     head = new ListNode(1);
60     head->next = new ListNode(2);
61     head->next->next = new ListNode(3);
62     head->next->next->next = new ListNode(4);
63     head->next->next->next->next = new ListNode(5);
64
65     // Reverse recursively
66     ListNode* reversedRecursive = reverseListRecursive(head);
67     cout << "Reversed List (Recursive): ";
68     printList(reversedRecursive);
69
70     return 0;
71 }
```

```
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



Run

Output

Clear

```
1  #include <iostream>
2  #include <vector>
3  #include <string>
4  using namespace std;
5
6  // Function to implement wildcard matching
7  bool isMatch(string s, string p) {
8      int m = s.size(), n = p.size();
9
10     // Create a DP table where dp[i][j] means whether s[0..i-1]
        matches p[0..j-1]
11     vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));
12
13     // Base case: Empty string matches with empty pattern
14     dp[0][0] = true;
15
16     // Base case: Handle patterns with '*' at the start
17     for (int j = 1; j <= n; j++) {
18         if (p[j - 1] == '*') {
19             dp[0][j] = dp[0][j - 1];
20         }
21     }
22
23     // Fill the DP table
24     for (int i = 1; i <= m; i++) {
25         for (int j = 1; j <= n; j++) {
26             if (p[j - 1] == s[i - 1] || p[j - 1] == '?') {
27                 // Characters match, or '?' matches any single
                character
28                 dp[i][j] = dp[i - 1][j - 1];
```

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

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

main.cpp



Share

Run

Output

Clear

```
28         dp[i][j] = dp[i - 1][j - 1];
29     } else if (p[j - 1] == '*') {
30         // '*' matches any sequence: empty sequence or
           one or more characters
31         dp[i][j] = dp[i][j - 1] || dp[i - 1][j];
32     }
33 }
34 }
35
36 return dp[m][n];
37 }
38
39 int main() {
40     string s, p;
41     cout << "Enter input string: ";
42     cin >> s;
43     cout << "Enter pattern: ";
44     cin >> p;
45
46     if (isMatch(s, p)) {
47         cout << "The pattern matches the string." << endl;
48     } else {
49         cout << "The pattern does not match the string." <<
           endl;
50     }
51
52     return 0;
53 }
54
```

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

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

main.cpp



Share

Run

Output

Clear

```
1 #include <iostream>
2 #include <vector>
3 #include <string>
4 #include <algorithm>
5 using namespace std;
6 string makeLargestSpecial(string s) {
7     if (s.empty()) return "";
8
9     vector<string> substrings;
10    int balance = 0, start = 0;
11    for (int i = 0; i < s.size(); ++i) {
12        balance += (s[i] == '1') ? 1 : -1;
13        if (balance == 0) {
14            string substring = s.substr(start + 1, i - start -
15                                     1);
16            substrings.push_back('1' + makeLargestSpecial
17                                (substring) + '0');
18            start = i + 1;
19        }
20    }
21    sort(substrings.begin(), substrings.end(), greater<string>
22          >());
23    string result = "";
24    for (const string& sub : substrings) {
25        result += sub;
26    }
27    return result;
28 }
```

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

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



main.cpp



Share

Run

Output

Clear

```
13     if (balance -- 0) {
14         string substring = s.substr(start + 1, i - start -
            1);
15         substrings.push_back('1' + makeLargestSpecial
            (substring) + '0');
16         start = i + 1;
17     }
18 }
19 sort(substrings.begin(), substrings.end(), greater<string
    >());
20 string result = "";
21 for (const string& sub : substrings) {
22     result += sub;
23 }
24
25 return result;
26 }
27
28 int main() {
29     string s;
30     cout << "Enter a special binary string: ";
31     cin >> s;
32
33     string result = makeLargestSpecial(s);
34     cout << "Lexicographically largest special binary string: "
        << result << endl;
35
36     return 0;
37 }
38
```

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

=== Code Execution Successful ===



main.cpp



Share

Run

Output

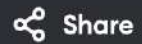
Clear

```
1  #include <iostream>
2  using namespace std;
3
4  struct ListNode {
5      int val;
6      ListNode* next;
7      ListNode(int x) : val(x), next(nullptr) {}
8  };
9
10 ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
11     ListNode* dummy = new ListNode(0);
12     ListNode* curr = dummy;
13     int carry = 0;
14
15     while (l1 || l2 || carry) {
16         int sum = carry;
17         if (l1) {
18             sum += l1->val;
19             l1 = l1->next;
20         }
21         if (l2) {
22             sum += l2->val;
23             l2 = l2->next;
24         }
25         curr->next = new ListNode(sum % 10);
26         carry = sum / 10;
27         curr = curr->next;
28     }
29
30     return dummy->next;
```

Sum: 7 0 8

=== Code Execution Successful ===

main.cpp



Share

Run

Output

Clear

```
26     carry = sum / 10;
27     curr = curr->next;
28 }
29
30     return dummy->next;
31 }
32
33 void printList(ListNode* head) {
34     while (head) {
35         cout << head->val << " ";
36         head = head->next;
37     }
38     cout << endl;
39 }
40
41 int main() {
42     ListNode* l1 = new ListNode(2);
43     l1->next = new ListNode(4);
44     l1->next->next = new ListNode(3);
45
46     ListNode* l2 = new ListNode(5);
47     l2->next = new ListNode(6);
48     l2->next->next = new ListNode(4);
49
50     ListNode* result = addTwoNumbers(l1, l2);
51     cout << "Sum: ";
52     printList(result);
53
54     return 0;
55 }
```

Sum: 7 0 8

=== Code Execution Successful ===