



DOMAIN WINTER WINNING CAMP ASSIGNMENT

Student Name: Sudhanshu Malhotra UID: 22BCS10631

Branch: BE-CSE::CS201

Section/Group: 22BCS_FL_IOT-603/A

Semester: 5th

➤ DAY-2 [20-12-2024]

1. Majority Elements

(Very Easy)

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

Implementation/Code:

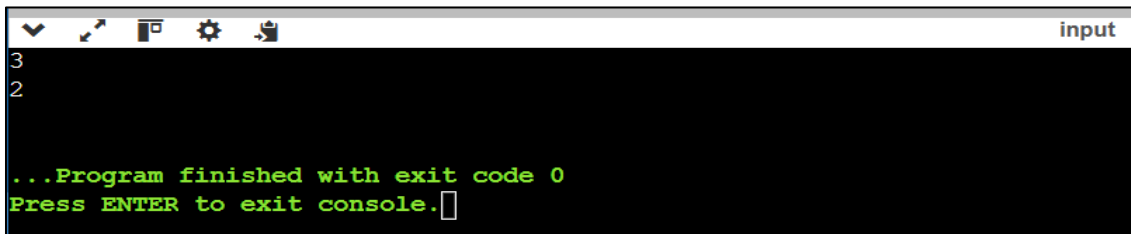
```
#include <iostream>                                     //Programming in C++
#include <vector>
using namespace std;

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

    // First pass to find the candidate
    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 << majorityElement(nums1) << endl;  
    vector<int> nums2 = {2, 2, 1, 1, 1, 2, 2};  
    cout << majorityElement(nums2) << endl;  
    return 0;  
}
```

Output:



```
input  
3  
2  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

2. Pascal's Triangle

(Easy)

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:

Example 1:

Input: numRows = 5

Output: [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]

Example 2:

Input: numRows = 1

Output: [[1]]

Constraints:

$1 \leq \text{numRows} \leq 30$

Implementation/Code:

```
#include <iostream> //Programming in C++
#include <vector>

using namespace std;

vector<vector<int>> generatePascalsTriangle(int numRows) {
    vector<vector<int>> triangle;

    // Start with the first row
    for (int i = 0; i < numRows; ++i) {
        vector<int> row(i + 1, 1); // Initialize row with 1s
        for (int j = 1; j < i; ++j) {
            // Compute the value based on the previous row
            row[j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
        }
        triangle.push_back(row);
    }

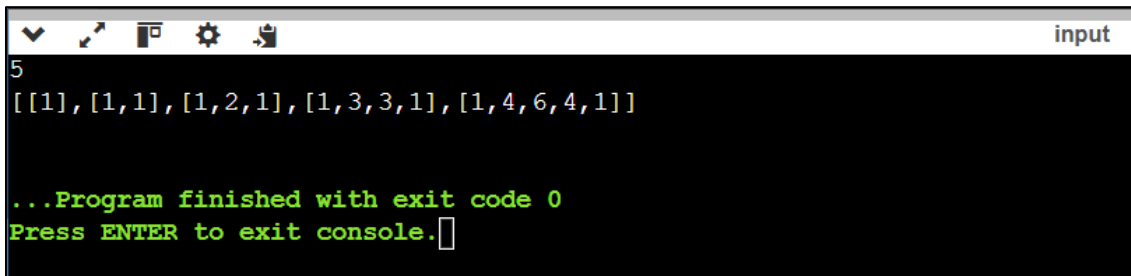
    return triangle;
}

int main() {
    int numRows;
    cin >> numRows;
    vector<vector<int>> result = generatePascalsTriangle(numRows);

    // Print the triangle in the specified format
    cout << "[";
    for (size_t i = 0; i < result.size(); ++i) {
        cout << "[";
        for (size_t j = 0; j < result[i].size(); ++j) {
            cout << result[i][j];
```

```
        if (j < result[i].size() - 1) {  
            cout << ",";  
        }  
    }  
    cout << "]  
    if (i < result.size() - 1) {  
        cout << ",";  
    }  
}  
cout << "]" << endl;  
  
return 0;  
}
```

Output:



```
5  
[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

3. Container With Most Water

(Medium)

You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the `i`th 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.

Implementation/Code:

```
#include <iostream>                                     //Programming in C++
#include <vector>
#include <algorithm>

using namespace std;

int maxArea(vector<int>& height) {
    int left = 0;           // Start pointer
    int right = height.size() - 1; // End pointer
    int max_area = 0;       // Maximum area found

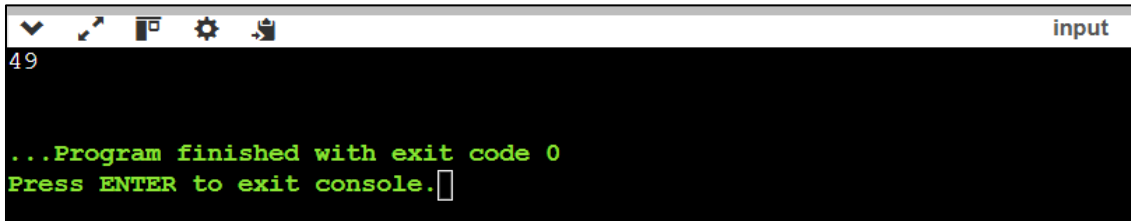
    while (left < right) {
        // Calculate the area between the two pointers
        int width = right - left;
        int current_area = min(height[left], height[right]) * width;
        max_area = max(max_area, current_area);

        // Move the pointer pointing to the shorter line
        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }

    return max_area;
}

int main() {
    vector<int> height = {1, 8, 6, 2, 5, 4, 8, 3, 7};
    cout << maxArea(height) << endl;
    return 0;
}
```

Output:



```
49  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

4. Maximum Number of Groups Getting Fresh Donuts (Hard)

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.

Example 1:

Input: `batchSize = 3, groups = [1,2,3,4,5,6]`

Output: 4

Explanation: You can arrange the groups as [6,2,4,5,1,3]. Then the 1st, 2nd, 4th, and 6th groups will be happy.

Implementation/Code:

```
#include <iostream>                                     //Programming in C++  
#include <vector>  
#include <unordered_map>  
#include <functional>  
#include <numeric>
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
using namespace std;
```

```
class Solution {  
public:
```

```
    int maxHappyGroups(int batchSize, vector<int>& groups) {  
        vector<int> remainders(batchSize, 0);  
        for (int group : groups) {  
            remainders[group % batchSize]++;  
        }  
    }
```

```
    // Handle groups that are happy by default  
    int happyGroups = remainders[0];  
    remainders[0] = 0;
```

```
    // Memoization map for DP  
    unordered_map<string, int> memo;
```

```
    // Recursive function  
    function<int(int, vector<int>&>> dfs = [&](int currentRemainder, vector<int>&  
rem) {  
        string state = to_string(currentRemainder) + "," + encode(rem);  
        if (memo.count(state)) {  
            return memo[state];  
        }  
  
        int result = 0;  
        for (int r = 1; r < batchSize; ++r) {  
            if (rem[r] > 0) {  
                rem[r]--;  
                int nextRemainder = (currentRemainder + r) % batchSize;  
                result = max(result, (nextRemainder == 0) + dfs(nextRemainder, rem));  
                rem[r]++;  
            }  
        }  
  
        return memo[state] = result;  
    }  
};
```

```
        return happyGroups + dfs(0, remainders);
    }

private:
    // Helper function to encode the state of remainders
    string encode(const vector<int>& rem) {
        string res;
        for (int r : rem) {
            res += to_string(r) + ",";
        }
        return res;
    }
};

int main() {
    Solution sol;
    vector<int> groups1 = {1, 2, 3, 4, 5, 6};
    int batchSize1 = 3;
    cout << sol.maxHappyGroups(batchSize1, groups1) << endl;
    return 0;
}
```

Output:**5. Find Minimum Time to Finish All Jobs***(Very Hard)*

You are given an integer array `jobs`, where `jobs[i]` is the amount of time it takes to complete the i^{th} 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.

Implementation/Code:

```
#include <iostream> //Programming in C++
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

class Solution {
public:
    int minimumTimeRequired(vector<int>& jobs, int k) {
        // Sort jobs in descending order to prioritize larger jobs
        sort(jobs.rbegin(), jobs.rend());

        // Initialize worker times
        vector<int> workerTime(k, 0);

        // Backtracking function
        int result = accumulate(jobs.begin(), jobs.end(), 0); // Maximum possible time
        backtrack(jobs, 0, workerTime, k, result);
        return result;
    }

private:
    void backtrack(vector<int>& jobs, int index, vector<int>& workerTime, int k, int& result) {
        if (index == jobs.size()) {
            result = min(result, *max_element(workerTime.begin(), workerTime.end()));
        }
    }
}
```

```
        return;
    }

    for (int i = 0; i < k; ++i) {
        // Assign the current job to worker i
        workerTime[i] += jobs[index];

        // Prune if this assignment exceeds the current best result
        if (workerTime[i] < result) {
            backtrack(jobs, index + 1, workerTime, k, result);
        }

        // Backtrack
        workerTime[i] -= jobs[index];

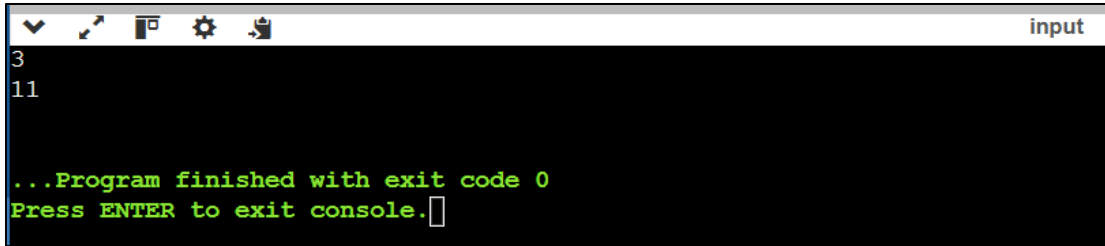
        // Optimization: If a worker has no jobs, don't assign the same job to another
        worker
        if (workerTime[i] == 0) break;
    }
};

int main()
{
    Solution sol;
    vector<int> jobs1 = {3, 2, 3};
    int k1 = 3;
    cout << sol.minimumTimeRequired(jobs1, k1) << endl;

    vector<int> jobs2 = {1, 2, 4, 7, 8};
    int k2 = 2;
    cout << sol.minimumTimeRequired(jobs2, k2) << endl;

    return 0;
}
```

Output:



```
3
11

...Program finished with exit code 0
Press ENTER to exit console.
```

6. Single Number

(*Very Easy*)

Given a non-empty array of integers `nums`, every element appears twice except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant extra space.

Example 1:

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

Output: 1

Example 2:

Input: `nums = [4,1,2,1,2]`

Output: 4

Example 3:

Input: `nums = [1]`

Output: 1

Constraints:

$1 \leq \text{nums.length} \leq 3 * 10^4$

$-3 * 10^4 \leq \text{nums}[i] \leq 3 * 10^4$

Each element in the array appears twice except for one element which appears only once.

Implementation/Code:

```
#include <iostream>
```

```
//Programming in C++
```

```
#include <vector>
```

```
using namespace std;
```

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int result = 0;
        for (int num : nums) {
            result ^= num; // XOR each number with the result
        }
        return result;
    }
};

int main() {
    Solution sol;
    vector<int> nums1 = {2, 2, 1};
    cout << sol.singleNumber(nums1) << endl;

    vector<int> nums2 = {4, 1, 2, 1, 2};
    cout << sol.singleNumber(nums2) << endl;

    vector<int> nums3 = {1};
    cout << sol.singleNumber(nums3) << endl;

    return 0;
}
```

Output:



```
input
1
4
1
...Program finished with exit code 0
Press ENTER to exit console.
```

7. Maximum Number of Darts Inside of a Circular Dartboard *(Hard)*

Alice is throwing n darts on a very large wall. You are given an array darts where $\text{darts}[i] = [x_i, y_i]$ is the position of the i th dart that Alice threw on the wall.

Bob knows the positions of the n darts on the wall. He wants to place a dartboard of radius r on the wall so that the maximum number of darts that Alice throws lie on the dartboard.

Given the integer r , return the maximum number of darts that can lie on the dartboard.

Example 1:

Input: darts = $[[-2, 0], [2, 0], [0, 2], [0, -2]]$, $r = 2$

Output: 4

Explanation: Circle dartboard with center in $(0,0)$ and radius = 2 contain all points.

Example 2:

Input: darts = $[[-3,0],[3,0],[2,6],[5,4],[0,9],[7,8]]$, $r = 5$

Output: 5

Explanation: Circle dartboard with center in $(0,4)$ and radius = 5 contain all points except the point $(7,8)$.

Constraints:

$1 \leq \text{darts.length} \leq 100$

$\text{darts}[i].\text{length} == 2$

$-104 \leq x_i, y_i \leq 104$

All the darts are unique

$1 \leq r \leq 5000$

Implementation/Code:

```
#include <iostream>                                     //Programming in C++
#include <vector>
#include <cmath>
#include <algorithm>

using namespace std;

class Solution {
public:
```

```
int numDartsInsideCircle(vector<vector<int>>& darts, int r) {
    int n = darts.size();
    int maxDarts = 0;

    // Function to calculate squared distance between two points
    auto distanceSquared = [](const vector<int>& p1, const vector<int>& p2) {
        return (p1[0] - p2[0]) * (p1[0] - p2[0]) + (p1[1] - p2[1]) * (p1[1] - p2[1]);
    };

    // Iterate over each pair of darts
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            // Calculate the midpoint of the two darts
            int midX = (darts[i][0] + darts[j][0]) / 2;
            int midY = (darts[i][1] + darts[j][1]) / 2;

            // Calculate the distance between the two darts
            int distSq = distanceSquared(darts[i], darts[j]);

            // If the distance between the two darts is greater than 2 * r, they can't both be
            // in the same circle
            if (distSq > 4 * r * r) continue;

            // Count how many darts lie within a circle centered at the midpoint
            int count = 0;
            for (int k = 0; k < n; ++k) {
                if (distanceSquared(darts[k], {midX, midY}) <= r * r) {
                    count++;
                }
            }

            // Update the maximum darts inside the circle
            maxDarts = max(maxDarts, count);
        }
    }

    // Additionally, consider the case where a single dart is the center of the circle
    for (int i = 0; i < n; ++i) {
```

```
        int count = 0;
        for (int j = 0; j < n; ++j) {
            if (distanceSquared(darts[i], darts[j]) <= r * r) {
                count++;
            }
        }
        maxDarts = max(maxDarts, count);
    }

    return maxDarts;
}

};

int main() {
    Solution sol;

    // Example 1
    vector<vector<int>> darts1 = {{-2, 0}, {2, 0}, {0, 2}, {0, -2}};
    int r1 = 2;
    cout << sol.numDartsInsideCircle(darts1, r1) << endl; // Output: 4

    // Example 2
    vector<vector<int>> darts2 = {{-3, 0}, {3, 0}, {2, 6}, {5, 4}, {0, 9}, {7, 8}};
    int r2 = 5;
    cout << sol.numDartsInsideCircle(darts2, r2) << endl; // Output: 5

    return 0;
}
```

Output:



```
input
4
5

...Program finished with exit code 0
Press ENTER to exit console.
```

8. Merge Two Sorted Lists

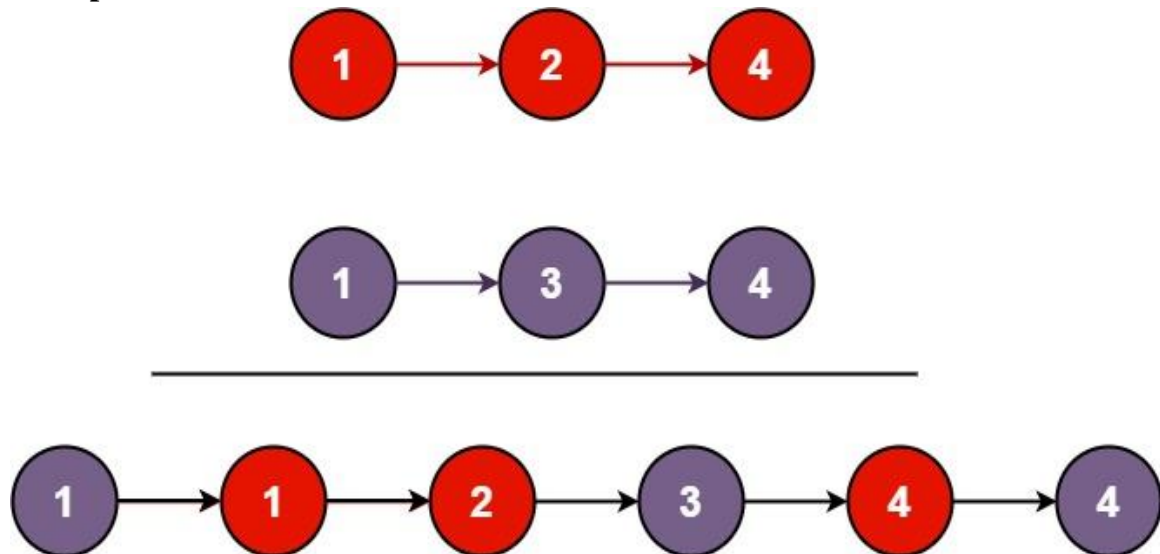
(*Very Easy*)

You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list*.

Example 1:



Input: list1 = [1,2,4], list2 = [1,3,4]

Output: [1,1,2,3,4,4]

Example 2:

Input: list1 = [], list2 = []

Output: []

Example 3:

Input: list1 = [], list2 = [0]

Output: [0]

Constraints:

- The number of nodes in both lists is in the range [0, 50].
- $-100 \leq \text{Node.val} \leq 100$
- Both list1 and list2 are sorted in **non-decreasing** order.

Implementation/Code:

```
#include <iostream>                                     //Programming in C++
#include <vector>
using namespace std;

// Definition for singly-linked list.
struct ListNode {
    int val;
    ListNode* next;

    // Constructor for a node
    ListNode(int x) : val(x), next(nullptr) {}
};

class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        // Dummy node to simplify list operations
        ListNode dummy(0);
        ListNode* current = &dummy;

        // Traverse both lists
        while (list1 != nullptr && list2 != nullptr) {
            if (list1->val <= list2->val) {
                current->next = list1;
                list1 = list1->next;
            } else {
                current->next = list2;
                list2 = list2->next;
            }
            current = current->next;
        }

        // Append the remaining nodes from either list
        if (list1 != nullptr) {
            current->next = list1;
        } else if (list2 != nullptr) {
```

```
        current->next = list2;
    }

    return dummy.next; // Return the merged list
}
};
```

```
// Helper functions to create and print linked lists
ListNode* createList(const vector<int>& values) {
    ListNode* head = nullptr;
    ListNode* tail = nullptr;

    for (int val : values) {
        ListNode* newNode = new ListNode(val);
        if (!head) {
            head = newNode;
            tail = newNode;
        } else {
            tail->next = newNode;
            tail = tail->next;
        }
    }

    return head;
}
```

```
void printList(ListNode* head) {
    cout<<"[";
    while (head != nullptr) {
        cout << head->val;
        if (head->next) cout << ",";
        head = head->next;
    }
    cout << "]"<< endl;
}
```

```
int main() {
    Solution solution;
```

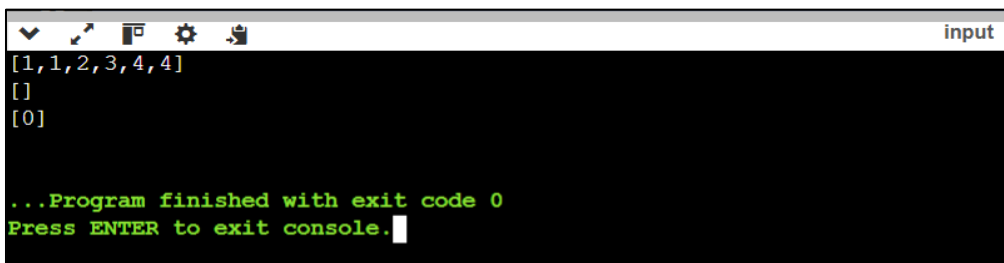
```
// Example 1
vector<int> values1 = {1, 2, 4};
vector<int> values2 = {1, 3, 4};
ListNode* list1 = createList(values1);
ListNode* list2 = createList(values2);
ListNode* mergedList = solution.mergeTwoLists(list1, list2);
printList(mergedList); // Output: 1 -> 1 -> 2 -> 3 -> 4 -> 4

// Example 2
list1 = nullptr;
list2 = nullptr;
mergedList = solution.mergeTwoLists(list1, list2);
printList(mergedList); // Output: (empty list)

// Example 3
list1 = nullptr;
vector<int> values3 = {0};
list2 = createList(values3);
mergedList = solution.mergeTwoLists(list1, list2);
printList(mergedList); // Output: 0

return 0;
}
```

Output:



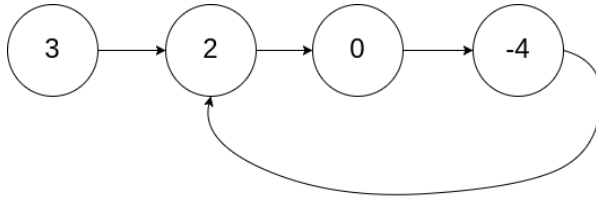
9. Linked List Cycle

(Very Easy)

Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. **Note that pos is not passed as a parameter.** Return true *if there is a cycle in the linked list*. Otherwise, return false.

Example 1:

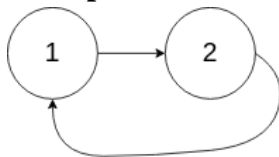


Input: head = [3,2,0,-4], pos = 1

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

Example 2:



Input: head = [1,2], pos = 0

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

Example 3:



Input: head = [1], pos = -1

Output: false

Explanation: There is no cycle in the linked list.

Constraints:

- The number of the nodes in the list is in the range [0, 104].
- $-105 \leq \text{Node.val} \leq 105$
- pos is -1 or a **valid index** in the linked-list.

Follow up: Can you solve it using O(1) (i.e. constant) memory?

Implementation/Code:

```
#include <iostream>                                     //Programming in C++
#include <vector>
using namespace std;

// Definition for singly-linked list.
struct ListNode {
    int val;
    ListNode* next;

    // Constructor for a node
    ListNode(int x) : val(x), next(nullptr) {}
};

class Solution {
public:
    bool hasCycle(ListNode* head) {
        if (!head || !head->next) {
            return false; // Empty list or single node without a cycle
        }

        ListNode* slow = head; // Slow pointer moves one step at a time
        ListNode* fast = head; // Fast pointer moves two steps at a time

        while (fast && fast->next) {
            slow = slow->next;    // Move slow pointer one step
            fast = fast->next->next; // Move fast pointer two steps

            if (slow == fast) {
                return true; // Cycle detected
            }
        }

        return false; // No cycle found
    }
};
```

```
// Helper function to create a cycle in the linked list
void createCycle(ListNode* head, int pos) {
    if (pos == -1) return;

    ListNode* cycleNode = nullptr;
    ListNode* tail = head;
    int index = 0;

    while (tail->next) {
        if (index == pos) {
            cycleNode = tail;
        }
        tail = tail->next;
        index++;
    }
    tail->next = cycleNode; // Create the cycle
}

// Helper function to create a linked list from a vector
ListNode* createList(const vector<int>& values) {
    if (values.empty()) return nullptr;

    ListNode* head = new ListNode(values[0]);
    ListNode* current = head;

    for (size_t i = 1; i < values.size(); ++i) {
        current->next = new ListNode(values[i]);
        current = current->next;
    }

    return head;
}

int main() {
    Solution solution;

    // Example 1
    vector<int> values1 = {3, 2, 0, -4};
```

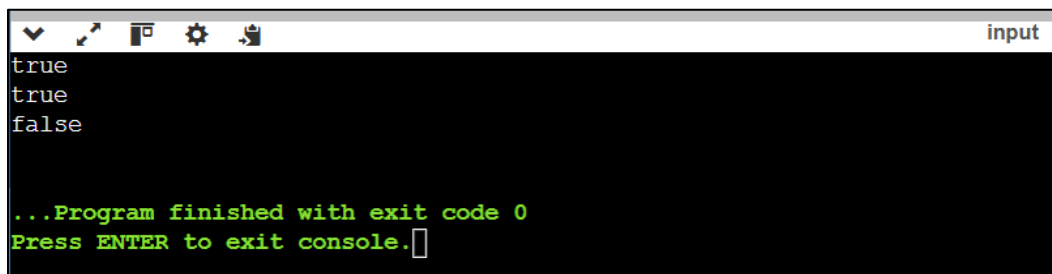
```
ListNode* head1 = createList(values1);
createCycle(head1, 1); // Create a cycle at position 1
cout << (solution.hasCycle(head1) ? "true" : "false") << endl; // Output: true

// Example 2
vector<int> values2 = {1, 2};
ListNode* head2 = createList(values2);
createCycle(head2, 0); // Create a cycle at position 0
cout << (solution.hasCycle(head2) ? "true" : "false") << endl; // Output: true

// Example 3
vector<int> values3 = {1};
ListNode* head3 = createList(values3);
createCycle(head3, -1); // No cycle
cout << (solution.hasCycle(head3) ? "true" : "false") << endl; // Output: false

return 0;
}
```

Output:



```
input
true
true
false

...Program finished with exit code 0
Press ENTER to exit console.
```

10. Remove Element

(Easy)

Given an integer array `nums` sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same. Then return the number of unique elements in `nums`.

Consider the number of unique elements of `nums` to be `k`, to get accepted, you need to do the following things:



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Change the array `nums` such that the first `k` elements of `nums` contain the unique elements in the order they were present in `nums` initially. The remaining elements of `nums` are not important as well as the size of `nums`. Return `k`.

Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int[] expectedNums = [...]; // The expected answer with correct length
```

```
int k = removeDuplicates(nums); // Calls your implementation
```

```
assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
    assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be accepted.

Example 1:

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

Output: 2, `nums = [1,2,_]`

Explanation: Your function should return `k = 2`, with the first two elements of `nums` being 1 and 2 respectively.

It does not matter what you leave beyond the returned `k` (hence they are underscores).

Example 2:

Input: `nums = [0,0,1,1,1,2,2,3,3,4]`

Output: 5, `nums = [0,1,2,3,4,_,_,_,_,_]`

Explanation: Your function should return `k = 5`, with the first five elements of `nums` being 0, 1, 2, 3, and 4 respectively.

It does not matter what you leave beyond the returned `k` (hence they are underscores).

Constraints:

$1 \leq \text{nums.length} \leq 3 \times 10^4$

$-100 \leq \text{nums}[i] \leq 100$

`nums` is sorted in non-decreasing order.

Implementation/Code:

```
#include <iostream> //Programming in C++
#include <vector>
using namespace std;

class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if (nums.empty()) return 0; // If the array is empty, return 0

        int k = 1; // Pointer to place the next unique element

        for (int i = 1; i < nums.size(); ++i) {
            if (nums[i] != nums[i - 1]) { // Check for a new unique element
                nums[k] = nums[i];      // Place the unique element at index k
                k++;                    // Increment the count of unique elements
            }
        }

        // Replace the remaining elements with underscores (not necessary for the algorithm
        // but for display purposes)
        for (int i = k; i < nums.size(); ++i) {
            nums[i] = '_';
        }

        return k; // Return the number of unique elements
    }
};

int main() {
    Solution solution;

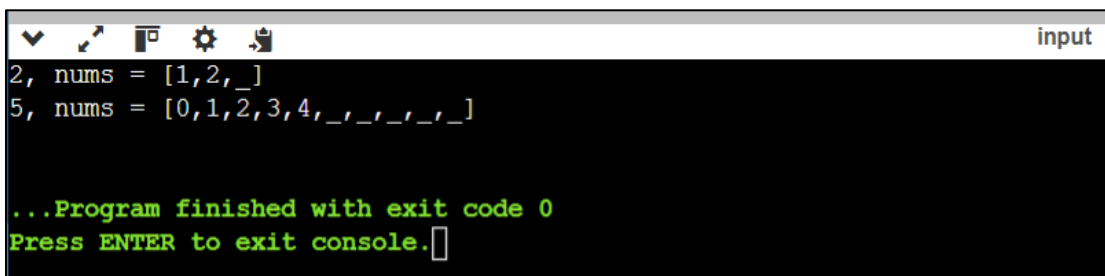
    // Example 1
    vector<int> nums1 = {1, 1, 2};
    int k1 = solution.removeDuplicates(nums1);
    cout << k1 << ", nums = [";
```

```
for (int i = 0; i < nums1.size(); ++i) {
    if (nums1[i] == '_') {
        cout << "_ ";
    } else {
        cout << nums1[i];
    }
    if (i < nums1.size() - 1) cout << ",";
}
cout << "]" << endl;

// Example 2
vector<int> nums2 = {0, 0, 1, 1, 1, 2, 2, 3, 3, 4};
int k2 = solution.removeDuplicates(nums2);
cout << k2 << ", nums = [";
for (int i = 0; i < nums2.size(); ++i) {
    if (nums2[i] == '_') {
        cout << "_ ";
    } else {
        cout << nums2[i];
    }
    if (i < nums2.size() - 1) cout << ",";
}
cout << "]" << endl;

return 0;
}
```

Output:



The screenshot shows a console window with a title bar containing standard Windows icons and the word "input". The output text is as follows:

```
2, nums = [1,2,_]
5, nums = [0,1,2,3,4,_,_,_,_,_]

...Program finished with exit code 0
Press ENTER to exit console.
```

11. Baseball Game

(Easy)

You are keeping the scores for a baseball game with strange rules. At the beginning of the game, you start with an empty record.

You are given a list of strings operations, where operations[i] is the ith operation you must apply to the record and is one of the following:

An integer x.

Record a new score of x.

'+'.

Record a new score that is the sum of the previous two scores.

'D'.

Record a new score that is the double of the previous score.

'C'.

Invalidate the previous score, removing it from the record.

Return the sum of all the scores on the record after applying all the operations.

The test cases are generated such that the answer and all intermediate calculations fit in a 32-bit integer and that all operations are valid.

Example 1:

Input: ops = ["5","2","C","D","+"]

Output: 30

Explanation:

"5" - Add 5 to the record, record is now [5].

"2" - Add 2 to the record, record is now [5, 2].

"C" - Invalidate and remove the previous score, record is now [5].

"D" - Add $2 * 5 = 10$ to the record, record is now [5, 10].

"+" - Add $5 + 10 = 15$ to the record, record is now [5, 10, 15].

The total sum is $5 + 10 + 15 = 30$.

Example 2:

Input: ops = ["5","-2","4","C","D","9","+","+"]

Output: 27

Explanation:



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

"5" - Add 5 to the record, record is now [5].
"-2" - Add -2 to the record, record is now [5, -2].
"4" - Add 4 to the record, record is now [5, -2, 4].
"C" - Invalidate and remove the previous score, record is now [5, -2].
"D" - Add $2 * -2 = -4$ to the record, record is now [5, -2, -4].
"9" - Add 9 to the record, record is now [5, -2, -4, 9].
"+" - Add $-4 + 9 = 5$ to the record, record is now [5, -2, -4, 9, 5].
"+" - Add $9 + 5 = 14$ to the record, record is now [5, -2, -4, 9, 5, 14].
The total sum is $5 + -2 + -4 + 9 + 5 + 14 = 27$.

Example 3:

Input: ops = ["1","C"]

Output: 0

Explanation:

"1" - Add 1 to the record, record is now [1].

"C" - Invalidate and remove the previous score, record is now [].

Since the record is empty, the total sum is 0.

Constraints:

$1 \leq \text{operations.length} \leq 1000$

operations[i] is "C", "D", "+", or a string representing an integer in the range $[-3 * 10^4, 3 * 10^4]$.

For operation "+", there will always be at least two previous scores on the record.

For operations "C" and "D", there will always be at least one previous score on the record.

Implementation/Code:

```
#include <iostream> //Programming in C++
#include <vector>
#include <string>
using namespace std;

class Solution {
public:
    int calPoints(vector<string>& ops) {
```

```
vector<int> record; // To store the scores

for (const string& op : ops) {
    if (op == "C") {
        // Remove the last score
        record.pop_back();
    } else if (op == "D") {
        // Double the last score and add it to the record
        record.push_back(2 * record.back());
    } else if (op == "+") {
        // Add the sum of the last two scores
        int n = record.size();
        record.push_back(record[n - 1] + record[n - 2]);
    } else {
        // Add the integer score
        record.push_back(stoi(op));
    }
}

// Calculate the total sum of scores
int totalSum = 0;
for (int score : record) {
    totalSum += score;
}

return totalSum;
};

int main() {
    Solution solution;

    // Example 1
    vector<string> ops1 = {"5", "2", "C", "D", "+"};
    cout << solution.calPoints(ops1) << endl;

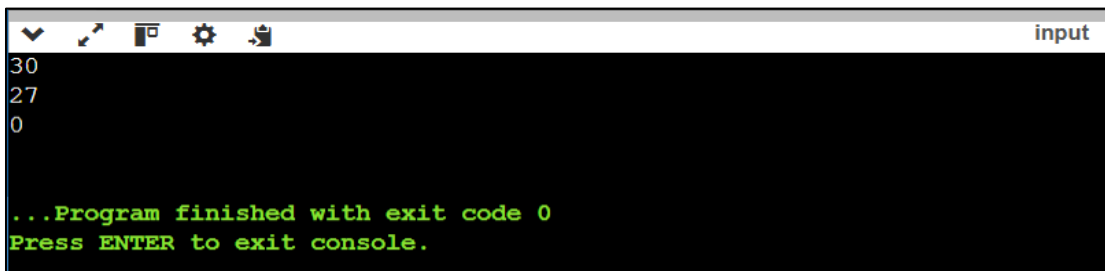
    // Example 2
    vector<string> ops2 = {"5", "-2", "4", "C", "D", "9", "+", "+"};
```

```
cout << solution.calPoints(ops2) << endl;

// Example 3
vector<string> ops3 = {"1", "C"};
cout << solution.calPoints(ops3) << endl;

return 0;
}
```

Output:



```
input
30
27
0

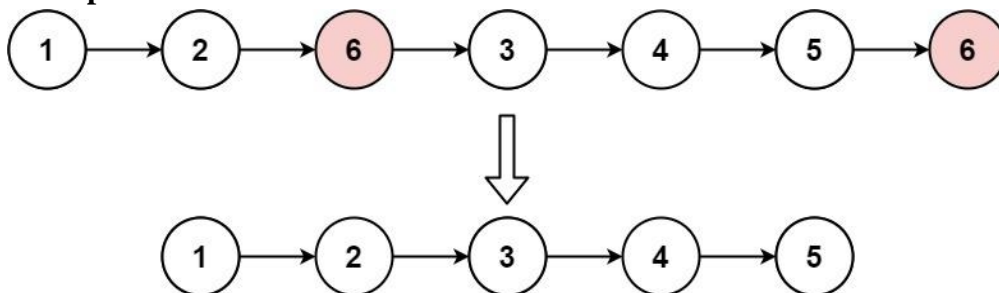
...Program finished with exit code 0
Press ENTER to exit console.
```

12. Remove Linked List Elements

(Easy)

Given the head of a linked list and an integer val, remove all the nodes of the linked list that has `Node.val == val`, and return *the new head*.

Example 1:



Input: head = [1,2,6,3,4,5,6], val = 6

Output: [1,2,3,4,5]

Example 2:

Input: head = [], val = 1

Output: []

Example 3:

Input: head = [7,7,7,7], val = 7

Output: []

Constraints:

- The number of nodes in the list is in the range [0, 104].
- $1 \leq \text{Node.val} \leq 50$
- $0 \leq \text{val} \leq 50$

Implementation/Code:

```
#include <iostream>                                     //Programming in C++
using namespace std;

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

// Function to remove elements from the linked list
ListNode* removeElements(ListNode* head, int val) {
    // Create a dummy node to handle edge cases
    ListNode* dummy = new ListNode(-1);
    dummy->next = head;

    // Pointer to traverse the list
    ListNode* current = dummy;
    while (current->next != nullptr) {
        if (current->next->val == val) {
            // Remove the node
            ListNode* temp = current->next;
            current->next = current->next->next;
            delete temp; // Free memory
        } else {
            current = current->next;
        }
    }
}
```

```
    }
}

// Save the new head and delete the dummy node
ListNode* newHead = dummy->next;
delete dummy;
return newHead;
}

// Helper function to print the linked list
void printList(ListNode* head) {
    cout<<"[";
    while (head != nullptr) {
        cout << head->val << ", ";
        head = head->next;
    }
    cout << "]" << endl;
}

// Main function to test the solution
int main() {
    // Example 1: Create the linked list [1, 2, 6, 3, 4, 5, 6]
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(6);
    head->next->next->next = new ListNode(3);
    head->next->next->next->next = new ListNode(4);
    head->next->next->next->next->next = new ListNode(5);
    head->next->next->next->next->next->next = new ListNode(6);

    int val = 6;
    ListNode* newHead = removeElements(head, val);

    printList(newHead);

    // Example 2: Empty list
    ListNode* emptyList = nullptr;
    printList(removeElements(emptyList, 1));
}
```



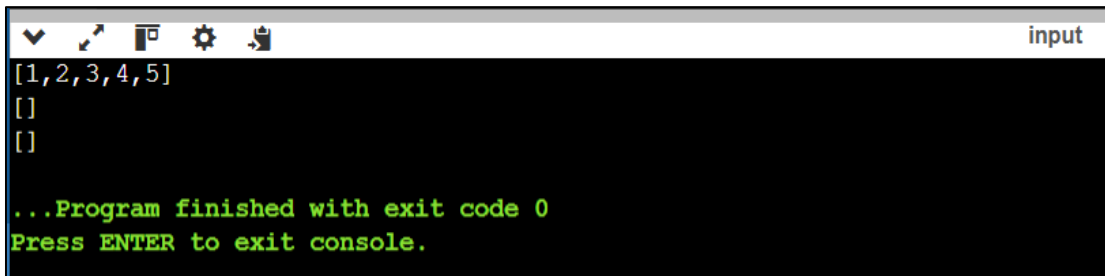
```
// Example 3: All nodes have the same value [7, 7, 7, 7]
ListNode* allSame = new ListNode(7);
allSame->next = new ListNode(7);
allSame->next->next = new ListNode(7);
allSame->next->next->next = new ListNode(7);

ListNode* result = removeElements(allSame, 7);

printList(result);

return 0;
}
```

Output:



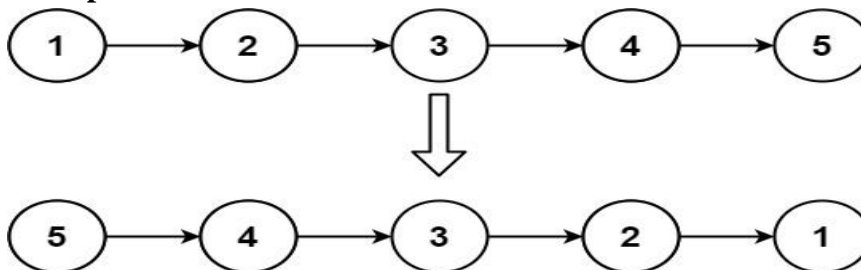
```
input
[1,2,3,4,5]
[]
[]
...Program finished with exit code 0
Press ENTER to exit console.
```

13. Reverse Linked List

(Easy)

Given the head of a singly linked list, reverse the list, and return *the reversed list*.

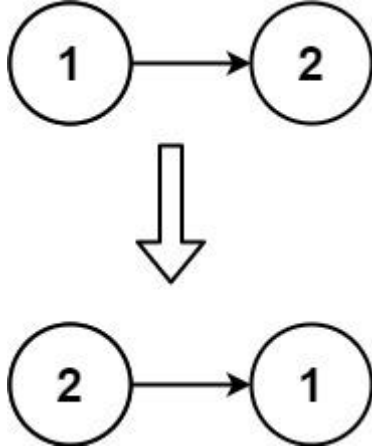
Example 1:



Input: head = [1,2,3,4,5]

Output: [5,4,3,2,1]

Example 2:



Input: head = [1,2]

Output: [2,1]

Example 3:

Input: head = []

Output: []

Constraints:

- The number of nodes in the list is the range [0, 5000].
- $-5000 \leq \text{Node.val} \leq 5000$

Follow up: A linked list can be reversed either iteratively or recursively. Could you implement both?

Implementation/Code:

```
#include <iostream>                                     //Programming in C++
using namespace std;

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

```
};

// Iterative approach to reverse the linked list
ListNode* reverseListIterative(ListNode* head) {
    ListNode* prev = nullptr;
    ListNode* current = head;

    while (current != nullptr) {
        ListNode* nextNode = current->next; // Save the next node
        current->next = prev;                // Reverse the link
        prev = current;                      // Move prev to current
        current = nextNode;                  // Move current to next node
    }

    return prev; // New head of the reversed list
}

// Recursive approach to reverse the linked list
ListNode* reverseListRecursive(ListNode* head) {
    // Base case: If the list is empty or has one node
    if (head == nullptr || head->next == nullptr) {
        return head;
    }

    // Reverse the rest of the list
    ListNode* newHead = reverseListRecursive(head->next);

    // Adjust the pointers
    head->next->next = head;
    head->next = nullptr;

    return newHead;
}

// Helper function to print the linked list
void printList(ListNode* head) {
    cout<<"[";
    while (head != nullptr) {
```

```
        cout << head->val << " ";
        head = head->next;
    }
    cout << "]" << endl;
}

// Main function to test the solution
int main() {
    // Example 1: Create the linked 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);

    // Test iterative reversal
    ListNode* reversedIterative = reverseListIterative(head);
    cout << "Reversed List (Iterative): ";
    printList(reversedIterative);

    // Reset the list for recursive test
    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);

    // Test recursive reversal
    ListNode* reversedRecursive = reverseListRecursive(head);
    cout << "Reversed List (Recursive): ";
    printList(reversedRecursive);

    return 0;
}
```

Output:

```
input
Original List: [1, 2, 3, 4, 5, ]
Reversed List (Iterative): [5, 4, 3, 2, 1, ]
Reversed List (Recursive): [5, 4, 3, 2, 1, ]

...Program finished with exit code 0
Press ENTER to exit console.
```

14. Valid Sudoku

(Medium)

Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated according to the following rules:

Each row must contain the digits 1-9 without repetition.

Each column must contain the digits 1-9 without repetition.

Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition.

Note:

A Sudoku board (partially filled) could be valid but is not necessarily solvable. Only the filled cells need to be validated according to the mentioned rules.

Example 1:

Input: board =

```
[["5","3",".",".","7",".",".",".","."],
["6",".",".","1","9","5",".",".","."],
[".","9","8",".",".",".","6","."],
["8",".",".","6",".",".","3"],
["4",".","8",".","3",".","1"],
["7",".","2",".","6"],
[".","6",".","2","8","."],
[".","4","1","9",".","5"],
[".","8",".","7","9"]]
```

Output: true

Example 2:

Input: board =

```
[["8","3",".",".","7",".",".",".","."]  
,"6",".",".","1","9","5",".",".","."]  
,".","9","8",".",".",".","6","."]  
,"8",".",".","6",".",".","3"]  
,"4",".","8",".","3",".","1"]  
,"7",".","2",".","6"]  
,"6",".","2","8","."]  
,".","4","1","9",".","5"]  
,".","8",".","7","9"]]
```

Output: false

Explanation: Same as Example 1, except with the 5 in the top left corner being modified to 8. Since there are two 8's in the top left 3x3 sub-box, it is invalid.

Constraints:

```
board.length == 9  
board[i].length == 9  
board[i][j] is a digit 1-9 or '.'.
```

Implementation/Code:

```
#include <iostream> //Programming in C++  
#include <vector>  
#include <unordered_set>  
using namespace std;  
  
bool isValidSudoku(vector<vector<char>>& board) {  
    // Use hash sets to track rows, columns, and 3x3 sub-boxes  
    vector<unordered_set<char>> rows(9), cols(9), boxes(9);  
  
    // Iterate through each cell in the board  
    for (int i = 0; i < 9; ++i) {  
        for (int j = 0; j < 9; ++j) {  
            char current = board[i][j];
```

```

        // Skip empty cells
        if (current == '.') continue;

        // Calculate the index of the 3x3 sub-box
        int boxIndex = (i / 3) * 3 + (j / 3);

        // Check if the current number already exists in the row, column, or box
        if (rows[i].count(current) || cols[j].count(current) ||
            boxes[boxIndex].count(current)) {
            return false;
        }

        // Add the current number to the corresponding row, column, and box
        rows[i].insert(current);
        cols[j].insert(current);
        boxes[boxIndex].insert(current);
    }
}

return true;
}

// Helper function to test the solution
int main() {
    vector<vector<char>> board1 = {
        {'5', '3', '.', '.', '7', '.', '.', '.', '.'},
        {'6', '.', '.', '1', '9', '5', '.', '.', '.'},
        {'.', '9', '8', '.', '.', '.', '.', '6', '.'},
        {'8', '.', '.', '.', '6', '.', '.', '.', '3'},
        {'4', '.', '.', '8', '.', '3', '.', '.', '1'},
        {'7', '.', '.', '.', '2', '.', '.', '.', '6'},
        {'.', '6', '.', '.', '.', '.', '2', '8', '.'},
        {'.', '.', '.', '4', '1', '9', '.', '.', '5'},
        {'.', '.', '.', '.', '8', '.', '.', '7', '9'}
    };

    vector<vector<char>> board2 = {
        {'8', '3', '.', '.', '7', '.', '.', '.', '.'},

```

```

        {'6', '.', '.', '1', '9', '5', '.', '.', '.'},
        {'.', '9', '8', '.', '.', '.', '.', '6', '.'},
        {'8', '.', '.', '.', '6', '.', '.', '.', '3'},
        {'4', '.', '.', '8', '.', '3', '.', '.', '1'},
        {'7', '.', '.', '.', '2', '.', '.', '.', '6'},
        {'.', '6', '.', '.', '.', '.', '2', '8', '.'},
        {'.', '.', '.', '4', '1', '9', '.', '.', '5'},
        {'.', '.', '.', '.', '8', '.', '.', '7', '9'}
    };

    cout << (isValidSudoku(board1) ? "true" : "false") << endl;
    cout << (isValidSudoku(board2) ? "true" : "false") << endl;

    return 0;
}

```

Output:



```

input
true
false

...Program finished with exit code 0
Press ENTER to exit console.

```

15. Jump Game II

(Medium)

You are given a 0-indexed array of integers `nums` of length `n`. You are initially positioned at `nums[0]`.

Each element `nums[i]` represents the maximum length of a forward jump from index `i`. In other words, if you are at `nums[i]`, you can jump to any `nums[i + j]` where:

$0 \leq j \leq \text{nums}[i]$ and

$i + j < n$

Return the minimum number of jumps to reach `nums[n - 1]`. The test cases are generated such that you can reach `nums[n - 1]`.

Example 1:

Input: nums = [2,3,1,1,4]

Output: 2

Explanation: The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.

Example 2:

Input: nums = [2,3,0,1,4]

Output: 2

Constraints:

$1 \leq \text{nums.length} \leq 10^4$

$0 \leq \text{nums}[i] \leq 1000$

It's guaranteed that you can reach $\text{nums}[n - 1]$.

Implementation/Code:

```
#include <iostream>                                     //Programming in C++
#include <vector>
#include <climits>
using namespace std;

int jump(vector<int>& nums) {
    int n = nums.size();
    if (n == 1) return 0; // If there's only one element, no jumps are needed.

    int jumps = 0;      // Count of jumps
    int current_end = 0; // The farthest index reachable in the current jump
    int farthest = 0;   // The farthest index reachable overall

    for (int i = 0; i < n - 1; ++i) {
        // Update the farthest reachable index
        farthest = max(farthest, i + nums[i]);
    }
```

```
// If we reach the end of the current jump's range

if (i == current_end) {
    jumps++;          // Increment the jump count
    current_end = farthest; // Update the range to the farthest reachable index

    // If we can reach or exceed the last index, stop
    if (current_end >= n - 1) break;
}

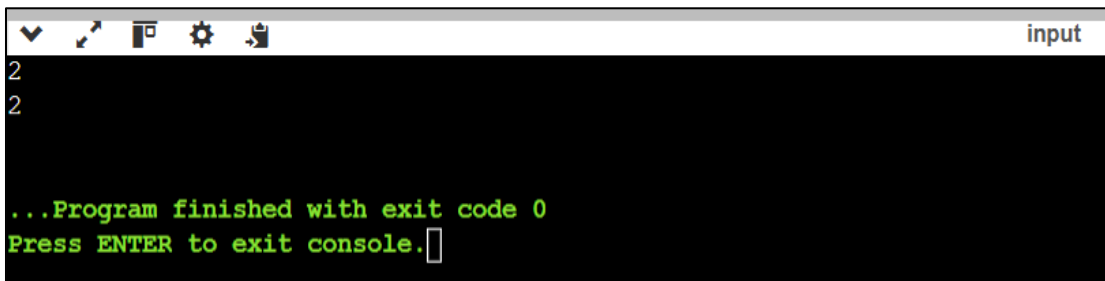
return jumps;
}

// Helper function to test the solution

int main()
{
    vector<int> nums1 = {2, 3, 1, 1, 4};
    vector<int> nums2 = {2, 3, 0, 1, 4};

    cout << jump(nums1) << endl; // Output: 2
    cout << jump(nums2) << endl; // Output: 2

    return 0;
}
```

Output:

```
input
2
2

...Program finished with exit code 0
Press ENTER to exit console.
```

16. Populating Next Right Pointers in Each Node

(Medium)

You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
    int val;
    Node *left;
    Node *right;
    Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

Example 1:

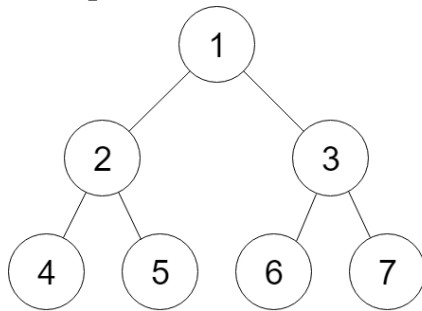


Figure A

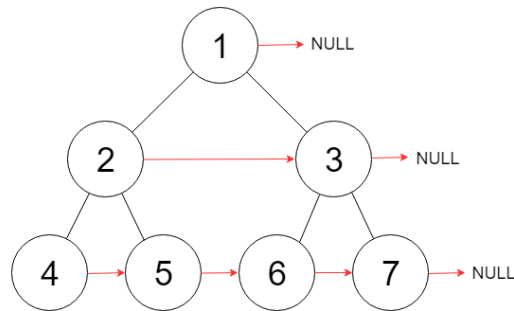


Figure B

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

Output: [1,#,2,3,#,4,5,6,7,#]

Explanation: Given the above perfect binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.

Example 2:

Input: root = []

Output: []

Constraints:

- The number of nodes in the tree is in the range [0, 2¹² - 1].
- -1000 ≤ Node.val ≤ 1000

Follow-up:

- You may only use constant extra space.
- The recursive approach is fine. You may assume implicit stack space does not count as extra space for this problem.

Implementation/Code:

```
#include <iostream> //Programming in C++
using namespace std;

// Definition for a Node.
class Node {
public:
    int val;
    Node* left;
    Node* right;
    Node* next;

    Node() : val(0), left(NULL), right(NULL), next(NULL) {}

    Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}

    Node(int _val, Node* _left, Node* _right, Node* _next)
        : val(_val), left(_left), right(_right), next(_next) {}
};

class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return nullptr;

        // Start with the leftmost node of the current level
        Node* leftmost = root;

        while (leftmost->left) { // While there are levels to process
            Node* current = leftmost;
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
while (current) { // Process nodes in the current level
    // Connect left child to right child
    current->left->next = current->right;

    // Connect right child to the left child of the next node
    if (current->next) {
        current->right->next = current->next->left;
    }

    // Move to the next node in the current level
    current = current->next;
}

// Move to the next level
leftmost = leftmost->left;
}

return root;
}
};

// Helper function to test the solution
void printTree(Node* root) {
    Node* level = root;
    cout<<"[";
    while (level) {
        Node* node = level;
        while (node) {
            cout << node->val << ", ";
            node = node->next;
        }
        cout << "#, "; // End of level
        level = level->left;
    }
    cout<<"]";
}

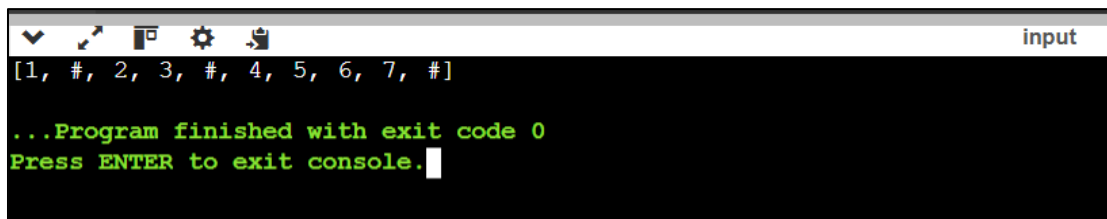
int main() {
```

```
// Create a perfect binary tree
Node* root = new Node(1);
root->left = new Node(2);
root->right = new Node(3);
root->left->left = new Node(4);
root->left->right = new Node(5);
root->right->left = new Node(6);
root->right->right = new Node(7);

Solution sol;
root = sol.connect(root);

// Print the tree level by level using next pointers
printTree(root); // Output: 1 # 2 3 # 4 5 6 7 #
return 0;
}
```

Output:



17. Design Circular Queue

(Medium)

Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle, and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.

Implement the MyCircularQueue class:

- `MyCircularQueue(k)` Initializes the object with the size of the queue to be `k`.
- `int Front()` Gets the front item from the queue. If the queue is empty, return `-1`.
- `int Rear()` Gets the last item from the queue. If the queue is empty, return `-1`.
- `boolean enqueue(int value)` Inserts an element into the circular queue. Return `true` if the operation is successful.
- `boolean dequeue()` Deletes an element from the circular queue. Return `true` if the operation is successful.
- `boolean isEmpty()` Checks whether the circular queue is empty or not.
- `boolean isFull()` Checks whether the circular queue is full or not.

You must solve the problem without using the built-in queue data structure in your programming language.

Example 1:

Input

```
["MyCircularQueue", "enqueue", "enqueue", "enqueue", "enqueue", "Rear", "isFull",  
"dequeue", "enqueue", "Rear"]
```

```
[[3], [1], [2], [3], [4], [], [], [], [4], []]
```

Output

```
[null, true, true, true, false, 3, true, true, true, 4]
```

Explanation

```
MyCircularQueue myCircularQueue = new MyCircularQueue(3);  
myCircularQueue.enqueue(1); // return True  
myCircularQueue.enqueue(2); // return True  
myCircularQueue.enqueue(3); // return True  
myCircularQueue.enqueue(4); // return False  
myCircularQueue.Rear();    // return 3  
myCircularQueue.isFull();  // return True
```

```
myCircularQueue.deQueue(); // return True  
myCircularQueue.enQueue(4); // return True  
myCircularQueue.Rear();    // return 4
```

Constraints:

- $1 \leq k \leq 1000$
- $0 \leq \text{value} \leq 1000$
- At most 3000 calls will be made to enQueue, deQueue, Front, Rear, isEmpty, and isFull.

Implementation/Code:

```
#include <iostream>                                     //Programming in C++  
#include <vector>  
using namespace std;  
  
class MyCircularQueue {  
private:  
    vector<int> queue;  
    int head, tail, size, capacity;  
  
public:  
    // Constructor to initialize the queue with size k  
    MyCircularQueue(int k) {  
        capacity = k;  
        queue.resize(k);  
        head = -1;  
        tail = -1;  
        size = 0;  
    }  
  
    // Inserts an element into the circular queue  
    bool enQueue(int value) {  
        if (isFull()) return false;
```



```
    if (isEmpty()) {
        head = 0;
    }
    tail = (tail + 1) % capacity;
    queue[tail] = value;
    size++;
    return true;
}

// Deletes an element from the circular queue
bool deQueue() {
    if (isEmpty()) return false;

    if (head == tail) { // Only one element
        head = -1;
        tail = -1;
    } else {
        head = (head + 1) % capacity;
    }
    size--;
    return true;
}

// Gets the front item from the queue
int Front() {
    if (isEmpty()) return -1;
    return queue[head];
}

// Gets the last item from the queue
int Rear() {
    if (isEmpty()) return -1;
    return queue[tail];
}

// Checks whether the circular queue is empty
bool isEmpty() {
    return size == 0;
}
```

```
    }

    // Checks whether the circular queue is full
    bool isFull() {
        return size == capacity;
    }
};

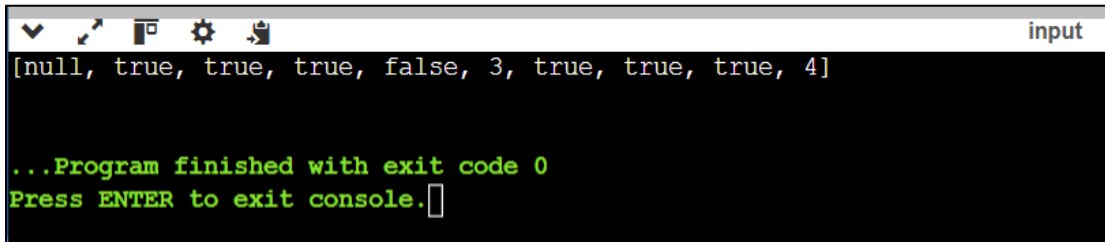
int main()
{
    // Initialize MyCircularQueue with size 3
    MyCircularQueue myCircularQueue(3);

    // Perform the sequence of operations and display results
    cout << "[null, ";

    // Enqueue 1
    cout << (myCircularQueue.enqueue(1) ? "true" : "false") << ", ";
    // Enqueue 2
    cout << (myCircularQueue.enqueue(2) ? "true" : "false") << ", ";
    // Enqueue 3
    cout << (myCircularQueue.enqueue(3) ? "true" : "false") << ", ";
    // Enqueue 4 (should return false as the queue is full)
    cout << (myCircularQueue.enqueue(4) ? "true" : "false") << ", ";
    // Get the rear element (should be 3)
    cout << myCircularQueue.Rear() << ", ";
    // Check if the queue is full (should return true)
    cout << (myCircularQueue.isFull() ? "true" : "false") << ", ";
    // Dequeue (should return true)
    cout << (myCircularQueue.dequeue() ? "true" : "false") << ", ";
    // Enqueue 4 (should return true)
    cout << (myCircularQueue.enqueue(4) ? "true" : "false") << ", ";
    // Get the rear element (should be 4)
    cout << myCircularQueue.Rear() << "]" << endl;

    return 0;
}
```

Output:



```
input
[null, true, true, true, false, 3, true, true, true, 4]

...Program finished with exit code 0
Press ENTER to exit console.
```

18. Cherry Pickup II

(Hard)

You are given a rows x cols matrix grid representing a field of cherries where grid[i][j] represents the number of cherries that you can collect from the (i, j) cell.

You have two robots that can collect cherries for you:

Robot #1 is located at the top-left corner (0, 0), and

Robot #2 is located at the top-right corner (0, cols - 1).

Return the maximum number of cherries collection using both robots by following the rules below:

From a cell (i, j), robots can move to cell (i + 1, j - 1), (i + 1, j), or (i + 1, j + 1).

When any robot passes through a cell, It picks up all cherries, and the cell becomes an empty cell.

When both robots stay in the same cell, only one takes the cherries.

Both robots cannot move outside of the grid at any moment.

Both robots should reach the bottom row in grid.

Example 1:

Input: grid = [[3,1,1],[2,5,1],[1,5,5],[2,1,1]]

Output: 24

Explanation: Path of robot #1 and #2 are described in color green and blue respectively.



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Cherries taken by Robot #1, $(3 + 2 + 5 + 2) = 12$.

Cherries taken by Robot #2, $(1 + 5 + 5 + 1) = 12$.

Total of cherries: $12 + 12 = 24$.

Example 2:

Input: grid = $[[1,0,0,0,0,0,1],[2,0,0,0,0,3,0],[2,0,9,0,0,0,0],[0,3,0,5,4,0,0],[1,0,2,3,0,0,6]]$

Output: 28

Explanation: Path of robot #1 and #2 are described in color green and blue respectively.

Cherries taken by Robot #1, $(1 + 9 + 5 + 2) = 17$.

Cherries taken by Robot #2, $(1 + 3 + 4 + 3) = 11$.

Total of cherries: $17 + 11 = 28$.

Constraints:

rows == grid.length

cols == grid[i].length

$2 \leq \text{rows}, \text{cols} \leq 70$

$0 \leq \text{grid}[i][j] \leq 100$

Implementation/Code:

```
#include <iostream>                                     //Programming in C++
#include <vector>
#include <algorithm>
using namespace std;

int cherryPickup(vector<vector<int>>& grid) {
    int rows = grid.size();
    int cols = grid[0].size();

    // DP table to store maximum cherries collected up to each point
```

```
vector<vector<vector<int>>> dp(rows, vector<vector<int>>(cols, vector<int>(cols, -1)));
```

```
// Initialize the first row
```

```
dp[0][0][cols-1] = grid[0][0] + grid[0][cols-1]; // Robots start at (0, 0) and (0, cols-1)
```

```
// Iterate through all rows
```

```
for (int r = 1; r < rows; r++) {
```

```
    for (int c1 = 0; c1 < cols; c1++) {
```

```
        for (int c2 = 0; c2 < cols; c2++) {
```

```
            if (dp[r-1][c1][c2] == -1) continue; // If no valid state in the previous row, skip
```

```
            // Try all combinations of moves for both robots
```

```
            for (int nc1 = max(0, c1-1); nc1 <= min(cols-1, c1+1); nc1++) {
```

```
                for (int nc2 = max(0, c2-1); nc2 <= min(cols-1, c2+1); nc2++) {
```

```
                    int cherries = grid[r][nc1] + grid[r][nc2];
```

```
                    if (nc1 == nc2) cherries -= grid[r][nc1]; // Avoid double counting if both robots are in the same cell
```

```
                    dp[r][nc1][nc2] = max(dp[r][nc1][nc2], dp[r-1][c1][c2] + cherries);
```

```
                }
```

```
            }
```

```
        }
```

```
    }
```

```
}
```

```
// Find the maximum cherries collected in the last row
```

```
int result = 0;
```

```
for (int c1 = 0; c1 < cols; c1++) {
```

```
    for (int c2 = 0; c2 < cols; c2++) {
```

```
        result = max(result, dp[rows-1][c1][c2]);
```

```
    }
```

```
}
```

```
return result;
```

```
}
```

```
int main() {
```

```
    vector<vector<int>> grid1 = {{3,1,1},{2,5,1},{1,5,5},{2,1,1}};
```

```
vector<vector<int>> grid2 =  
{ {1,0,0,0,0,0,1},{2,0,0,0,0,3,0},{2,0,9,0,0,0,0},{0,3,0,5,4,0,0},{1,0,2,3,0,0,6}};  
  
cout << cherryPickup(grid1) << endl;  
cout << cherryPickup(grid2) << endl;  
  
return 0;  
}
```

Output:



```
input  
24  
28  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

19. Minimum Number of People to Teach

(Very Hard)

On a social network consisting of m users and some friendships between users, two users can communicate with each other if they know a common language.

You are given an integer n , an array `languages`, and an array `friendships` where:

There are n languages numbered 1 through n ,

`languages[i]` is the set of languages the i th user knows, and

`friendships[i] = [ui, vi]` denotes a friendship between the users ui and vi .

You can choose one language and teach it to some users so that all friends can communicate with each other. Return the minimum number of users you need to teach.

Note that friendships are not transitive, meaning if x is a friend of y and y is a friend of z , this doesn't guarantee that x is a friend of z .

Example 1:

Input: $n = 2$, `languages = [[1],[2],[1,2]]`, `friendships = [[1,2],[1,3],[2,3]]`

Output: 1

Explanation: You can either teach user 1 the second language or user 2 the first language.

Example 2:

Input: $n = 3$, languages = [[2],[1,3],[1,2],[3]], friendships = [[1,4],[1,2],[3,4],[2,3]]

Output: 2

Explanation: Teach the third language to users 1 and 3, yielding two users to teach.

Constraints:

$2 \leq n \leq 500$

languages.length == m

$1 \leq m \leq 500$

$1 \leq \text{languages}[i].\text{length} \leq n$

$1 \leq \text{languages}[i][j] \leq n$

$1 \leq u_i < v_i \leq \text{languages.length}$

$1 \leq \text{friendships.length} \leq 500$

All tuples (u_i, v_i) are unique

languages[i] contains only unique values

Implementation/Code:

```
#include <iostream> //Programming in C++
#include <vector>
#include <unordered_set>
#include <unordered_map>
#include <algorithm>
using namespace std;

// Union-Find (Disjoint Set Union) structure to find connected components
```

```
class DSU {
public:
    vector<int> parent;
    vector<int> size;

    DSU(int n) {
        parent.resize(n);
        size.resize(n, 1);
        for (int i = 0; i < n; i++) parent[i] = i;
    }

    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]); // Path compression
        }
        return parent[x];
    }

    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            if (size[rootX] > size[rootY]) swap(rootX, rootY);
            parent[rootX] = rootY;
            size[rootY] += size[rootX];
        }
    }
};

int minimumTeachings(int n, vector<vector<int>>& languages, vector<vector<int>>&
friendships) {
    int m = languages.size();

    // Step 1: Initialize DSU to find connected components
    DSU dsu(m);
    for (const auto& f : friendships) {
        int u = f[0] - 1; // Convert to 0-based index
        int v = f[1] - 1; // Convert to 0-based index
```



```
        dsu.unite(u, v);
    }

    // Step 2: Group users by their connected components
    unordered_map<int, unordered_set<int>> componentLanguages;
    for (int i = 0; i < m; i++) {
        int root = dsu.find(i);
        for (int lang : languages[i]) {
            componentLanguages[root].insert(lang);
        }
    }

    // Step 3: Calculate the minimum number of users to teach
    int result = 0;
    for (auto& entry : componentLanguages) {
        int component = entry.first;
        unordered_set<int> knownLanguages = entry.second;
        unordered_map<int, int> languageCount;

        // Count how many users know each language in this component
        for (int i = 0; i < m; i++) {
            if (dsu.find(i) == component) {
                for (int lang : languages[i]) {
                    languageCount[lang]++;
                }
            }
        }

        // Find the language with the maximum number of users knowing it
        int maxKnownUsers = 0;
        for (const auto& langCount : languageCount) {
            maxKnownUsers = max(maxKnownUsers, langCount.second);
        }

        // If no language is known by all users in this component, teach one user a new
        language
        result += (knownLanguages.size() == 0 ? 1 : maxKnownUsers);
    }
```

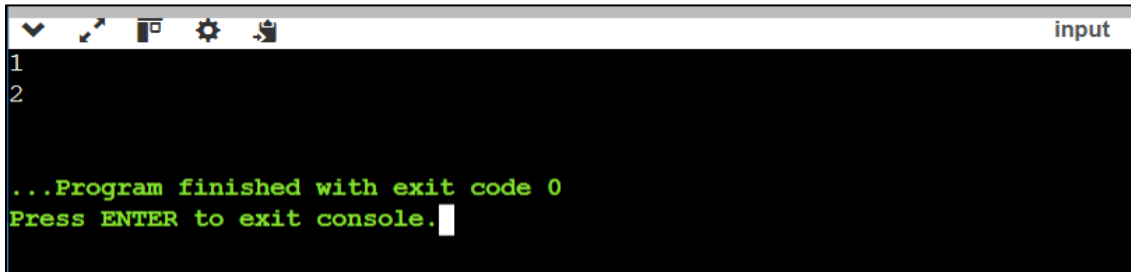
```
        return result;
    }

    int main() {
        int n1 = 2;
        vector<vector<int>> languages1 = {{1}, {2}, {1, 2}};
        vector<vector<int>> friendships1 = {{1, 2}, {1, 3}, {2, 3}};
        cout << minimumTeachings(n1, languages1, friendships1) << endl; // Output: 1

        int n2 = 3;
        vector<vector<int>> languages2 = {{2}, {1, 3}, {1, 2}, {3}};
        vector<vector<int>> friendships2 = {{1, 4}, {1, 2}, {3, 4}, {2, 3}};
        cout << minimumTeachings(n2, languages2, friendships2) << endl; // Output: 2

        return 0;
    }
```

Output:



```
input
1
2

...Program finished with exit code 0
Press ENTER to exit console.
```