DOMAIN WINTER WINNING CAMP ASSIGNMENT

Student Name: Gurnoor Oberoi **UID: 22BCS15716**
Branch: BE-CSE::CS201 **Section/Group: 22BCS_FL_IOT-603/B**
Semester: 5th

➢ **DAY-2 [20-12-2024]**

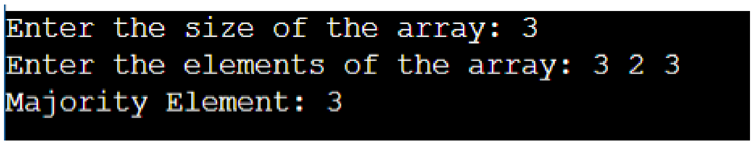
1. Majority Elements *(Very Easy)*

Given an array nums of size n, return the majority element.
The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

Implementation/Code:

```
#include <iostream>
#include <vector>
using namespace std;
int majorityElement(vector<int>& nums) {
    int candidate = 0, count = 0;
    for (int num : nums) {
        if (count == 0) {
            candidate = num;
        }
        count += (num == candidate) ? 1 : -1;
    }
    return candidate;
}
int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;
    cout << "Enter the elements of the array: ";
    vector<int> nums(n);
    for (int& num : nums) {
        cin >> num;
    }
    cout << "Majority Element: " << majorityElement(nums) << endl;
    return 0;
}
```

Output:



```
Enter the size of the array: 3
Enter the elements of the array: 3 2 3
Majority Element: 3
```

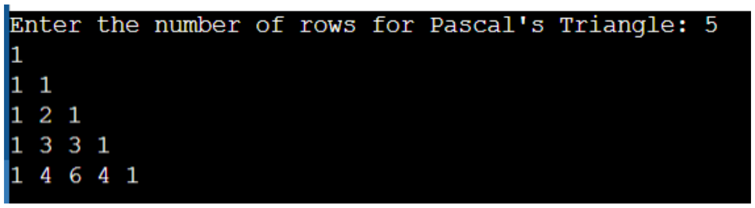
2. Pascal's Triangle *(Easy)*

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:

Implementation/Code:

```
#include <iostream>
using namespace std;
void generatePascalsTriangle(int numRows) {
    for (int i = 0; i < numRows; ++i) {
        int value = 1;
        for (int j = 0; j <= i; ++j) {
            cout << value << " ";
            value = value * (i - j) / (j + 1);
        }
        cout << endl;
    }
}
int main() {
    int numRows;
    cout << "Enter the number of rows for Pascal's Triangle: ";
    cin >> numRows;
    generatePascalsTriangle(numRows);
    return 0;
}
```

Output:



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

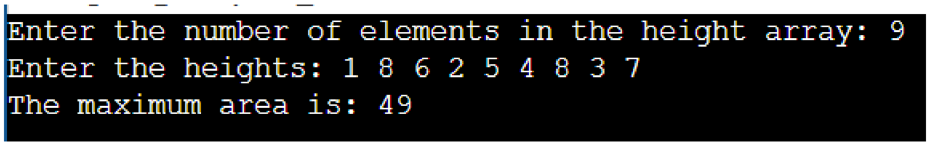
3. Container with Most Water *(Medium)*

You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).
Find two lines that together with the x-axis form a container, such that the container contains the most water.
Return the maximum amount of water a container can store.
Notice that you may not slant the container.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;
class Solution {
public:
    int maxArea(vector<int>& height) {
        int n = height.size();
        int l = 0, r = n - 1;
        int ans = INT_MIN;
        while (l < r) {
            int area = (r - l) * min(height[l], height[r]);
            ans = max(area, ans);
            if (height[l] < height[r]) {
                l++;
            } else {
                r--;
            }
        }
        return ans;
    }
};
int main() {
    int n;
    cout << "Enter the number of elements in the height array: ";
    cin >> n;
    vector<int> height(n);
    cout << "Enter the heights: ";
    for (int i = 0; i < n; ++i) {
        cin >> height[i];
    }
    Solution sol;
    int result = sol.maxArea(height);
    cout << "The maximum area is: " << result << endl;
    return 0;
}
```

Output:



```
Enter the number of elements in the height array: 9
Enter the heights: 1 8 6 2 5 4 8 3 7
The maximum area is: 49
```

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.

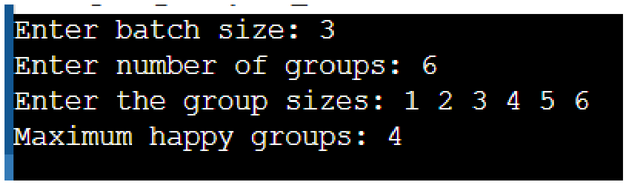
You can freely rearrange the ordering of the groups. Return the maximum possible number of happy groups after rearranging the groups.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include<functional>
```

```
#include <unordered_map>
#include <numeric>
using namespace std;
class Solution {
public:
    int maxHappyGroups(int batchSize, vector<int>& groups) {
        vector<int> count(batchSize, 0);
        for (int g : groups) {
            count[g % batchSize]++;
        }
        int happyGroups = count[0];
        count[0] = 0;
        for (int i = 1; i < batchSize; ++i) {
            if (count[i] > 0) {
                int pair = min(count[i], count[batchSize - i]);
                happyGroups += pair;
                count[i] -= pair;
                count[batchSize - i] -= pair;
            }
        }
        unordered_map<int, int> dp;
        function<int(int)> dfs = [&](int state) {
            if (dp.count(state)) return dp[state];
            int sum = 0, rem = 0;
            for (int i = 1; i < batchSize; ++i) {
                int countLeft = (state >> (i * 5)) & 31;
                sum += countLeft * i;
                rem += countLeft;
            }
            if (rem == 0) return 0;
            int maxHappy = 0;
            for (int i = 1; i < batchSize; ++i) {
                int countLeft = (state >> (i * 5)) & 31;
                if (countLeft > 0) {
                    int newState = state - (1 << (i * 5));
                    int newHappy = dfs(newState) + (sum % batchSize == 0 ? 1 : 0);
                    maxHappy = max(maxHappy, newHappy);
                }
            }
            return dp[state] = maxHappy;
        };
        int state = 0;
        for (int i = 1; i < batchSize; ++i) {
            state |= count[i] << (i * 5);
        }
        return happyGroups + dfs(state);
    }
};
int main() {
    Solution solution;
    int batchSize;
    cout << "Enter batch size: ";
    cin >> batchSize;
    int n;
    cout << "Enter number of groups: ";
    cin >> n;
    vector<int> groups(n);
    cout << "Enter the group sizes: ";
    for (int i = 0; i < n; ++i) {
        cin >> groups[i];
    }
    cout << "Maximum happy groups: " << solution.maxHappyGroups(batchSize, groups)
<< endl;
    return 0;
}
```

Output:



```
Enter batch size: 3
Enter number of groups: 6
Enter the group sizes: 1 2 3 4 5 6
Maximum happy groups: 4
```

5. Find Minimum Time to Finish all Job *(Very Hard)*

You are given an integer array jobs, where jobs[i] is the amount of time it takes to complete the ith job.There are k workers that you can assign jobs to. Each job should be assigned to exactly one worker. The working time of a worker is the sum of the time it

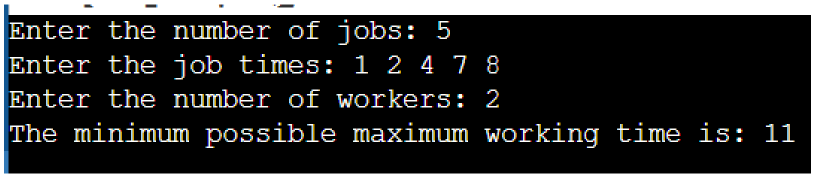
takes to complete all jobs assigned to them. Your goal is to devise an optimal assignment such that the maximum working time of any worker is minimized.
Return the minimum possible maximum working time of any assignment.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
using namespace std;
class Solution {
public:
   bool canAssign(vector<int>& jobs, int k, int maxTime, vector<int>& workers, int
jobIndex) {
      if (jobIndex == jobs.size()) return true;
      for (int i = 0; i < k; ++i) {
         if (workers[i] + jobs[jobIndex] <= maxTime) {
            workers[i] += jobs[jobIndex];
            if (canAssign(jobs, k, maxTime, workers, jobIndex + 1)) return true;
            workers[i] -= jobs[jobIndex];
            if (workers[i] == 0) break;
         }
      }
      return false;
   }
   int minimumTimeRequired(vector<int>& jobs, int k) {
      int left = *max_element(jobs.begin(), jobs.end());
      int right = accumulate(jobs.begin(), jobs.end(), 0);
      while (left < right) {
         int mid = left + (right - left) / 2;
         vector<int> workers(k, 0);
         if (canAssign(jobs, k, mid, workers, 0)) {
            right = mid;
         } else {
            left = mid + 1;
         }
      }
      return left;
```

```
    }
};
int main() {
    Solution solution;
    int n, k;
    cout << "Enter the number of jobs: ";
    cin >> n;
    vector<int> jobs(n);
    cout << "Enter the job times: ";
    for (int i = 0; i < n; ++i) {
        cin >> jobs[i];
    }
    cout << "Enter the number of workers: ";
    cin >> k;
    int result = solution.minimumTimeRequired(jobs, k);
    cout << "The minimum possible maximum working time is: " << result << endl;
    return 0;
}
```

Output:



```
Enter the number of jobs: 5
Enter the job times: 1 2 4 7 8
Enter the number of workers: 2
The minimum possible maximum working time is: 11
```

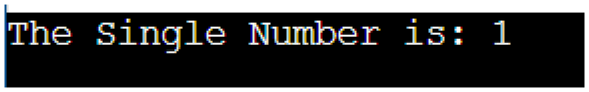
6. Single Number *(Very Easy)*

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.

Implementation/Code:

```
#include <iostream>
#include<vector>
using namespace std;
int singleNumber(vector<int>& nums ){
    int result = 0;
    for(int num : nums){
        result ^= num;
    }
    return result;
}
int main(){
    vector<int> nums = {2,2,1};
    cout<<"The Single Number is: "<<singleNumber(nums)<<endl;
    return 0;
}
```

Output:



```
The Single Number is: 1
```

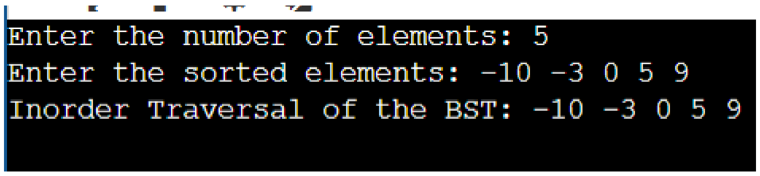
7. Convert Sorted Array to Binary Tree *(Very Easy)*

Given an integer array nums where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.

Implementation/Code:

```
#include <iostream>
#include <vector>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
TreeNode* sortedArrayToBST(const vector<int>& nums, int start, int end) {
    if (start > end) return NULL;
    int mid = (start + end) / 2;
    TreeNode* root = new TreeNode(nums[mid]);
    root->left = sortedArrayToBST(nums, start, mid - 1);
    root->right = sortedArrayToBST(nums, mid + 1, end);

    return root;
}
void inorderTraversal(TreeNode* root) {
    if (root) {
        inorderTraversal(root->left);
        cout << root->val << " ";
        inorderTraversal(root->right);
    }
}

int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    vector<int> nums(n);
    cout << "Enter the sorted elements: ";
    for (int i = 0; i < n; ++i) {
        cin >> nums[i];
    }
    TreeNode* root = sortedArrayToBST(nums, 0, n - 1);
    cout << "Inorder Traversal of the BST: ";
    inorderTraversal(root);
    cout << endl;
    return 0;
}
```

Output:



```
Enter the number of elements: 5
Enter the sorted elements: -10 -3 0 5 9
Inorder Traversal of the BST: -10 -3 0 5 9
```

8. Merge two Sorted Array *(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.

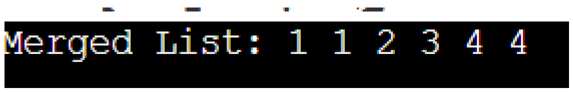
Return *the head of the merged linked list.*

Implementation/Code:

```
#include <iostream>
```

```
#include<vector>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
};
ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
    ListNode* dummy = new ListNode(0);
    ListNode* current = dummy;
    while (list1 != NULL && list2 != NULL) {
        if (list1->val <= list2->val) {
            current->next = list1;
            list1 = list1->next;
        } else {
            current->next = list2;
            list2 = list2->next;
        }
        current = current->next;
    }
    if (list1 != NULL) {
        current->next = list1;
    } else if (list2 != NULL) {
        current->next = list2;
    }
    ListNode* mergedHead = dummy->next;
    delete dummy;
    return mergedHead;
}
void printList(ListNode* head) {
    while (head != NULL) {
        cout << head->val << " ";
        head = head->next;
    }
    cout << endl;
}
ListNode* createList(const vector<int>& nums) {
    if (nums.empty()) return NULL;
    ListNode* head = new ListNode(nums[0]);
    ListNode* current = head;
    for (int i = 1; i < nums.size(); ++i) {
        current->next = new ListNode(nums[i]);
        current = current->next;
    }
    return head;
}

int main() {
    vector<int> nums1 = {1, 2, 4};
    vector<int> nums2 = {1, 3, 4};
    ListNode* list1 = createList(nums1);
    ListNode* list2 = createList(nums2);
    ListNode* mergedList = mergeTwoLists(list1, list2);
    cout << "Merged List: ";
    printList(mergedList);
    return 0;
}
```

Output:



```
Merged List: 1 1 2 3 4 4
```

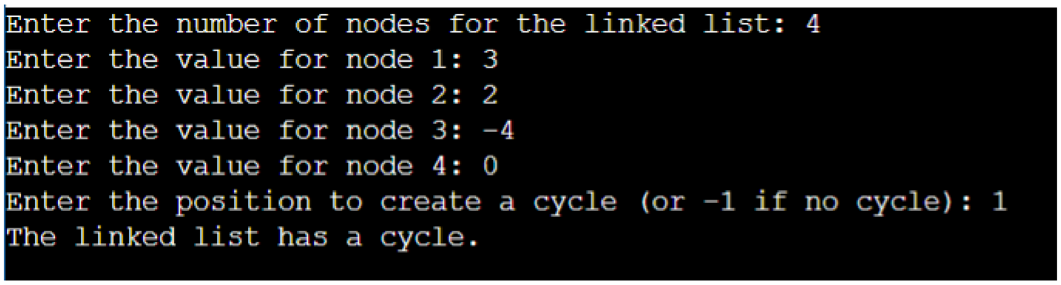
9. Linked List Cycle *(Very Easy)*

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.

Implementation/Code:

```
#include <iostream>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
};
bool hasCycle(ListNode* head) {
    if (head == NULL) return false;
    ListNode* slow = head;
    ListNode* fast = head;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) {
            return true;
        }
    }
    return false;
}
ListNode* createList(int n) {
    if (n <= 0) return NULL;
    int val;
    cout << "Enter the value for node 1: ";
    cin >> val;
    ListNode* head = new ListNode(val);
    ListNode* current = head;

    for (int i = 2; i <= n; ++i) {
        cout << "Enter the value for node " << i << ": ";
        cin >> val;
        current->next = new ListNode(val);
        current = current->next;
    }

    return head;
}
void createCycle(ListNode* head, int pos) {
    if (!head) return;
    ListNode* cycleStart = NULL;
    ListNode* current = head;
    int index = 0;
    while (current->next) {
        if (index == pos) {
            cycleStart = current;
        }
        current = current->next;
        index++;
    }
    if (cycleStart) {
        current->next = cycleStart;
    }
}
int main() {
    int n;
    cout << "Enter the number of nodes for the linked list: ";
    cin >> n;
    ListNode* head = createList(n);
    int pos;
    cout << "Enter the position to create a cycle (or -1 if no cycle): ";
    cin >> pos;

    if (pos != -1) {
        createCycle(head, pos);
    }
    if (hasCycle(head)) {
        cout << "The linked list has a cycle." << endl;
    } else {
        cout << "The linked list does not have a cycle." << endl;
    }
    return 0;
}
```

Output:



```
Enter the number of nodes for the linked list: 4
Enter the value for node 1: 3
Enter the value for node 2: 2
Enter the value for node 3: -4
Enter the value for node 4: 0
Enter the position to create a cycle (or -1 if no cycle): 1
The linked list has a cycle.
```

10. Remove Element *(Easy)*

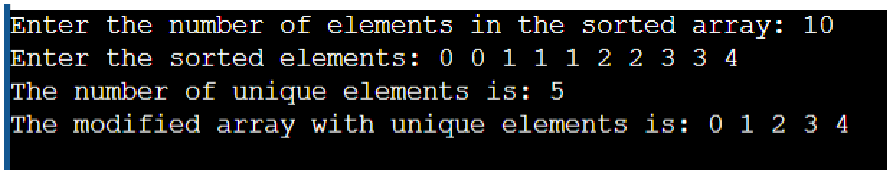
Given an integer array nums sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same. Then return the number of unique elements in nums.

Implementation/Code:

```
#include <iostream>
#include <vector>
using namespace std;
int removeDuplicates(vector<int>& nums) {
   if (nums.empty()) return 0;
   int i = 0;
   for (int j = 1; j < nums.size(); ++j) {
      if (nums[j] != nums[i]) {
         i++;
         nums[i] = nums[j];
      }
   }
   return i + 1;
}
int main() {
   int n;
   cout << "Enter the number of elements in the sorted array: ";
   cin >> n;
   vector<int> nums(n);
   cout << "Enter the sorted elements: ";
   for (int i = 0; i < n; ++i) {
      cin >> nums[i];
   }
   int uniqueCount = removeDuplicates(nums);
   cout << "The number of unique elements is: " << uniqueCount << endl;
   cout << "The modified array with unique elements is: ";
   for (int i = 0; i < uniqueCount; ++i) {
      cout << nums[i] << " ";
   }
   cout << endl;
   return 0;
```

}

Output:



```
Enter the number of elements in the sorted array: 10
Enter the sorted elements: 0 0 1 1 1 2 2 3 3 4
The number of unique elements is: 5
The modified array with unique elements is: 0 1 2 3 4
```

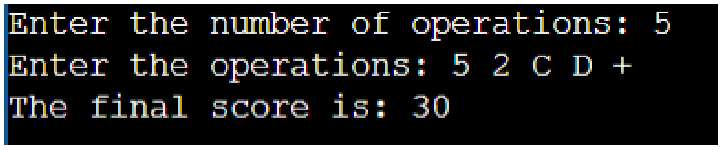
11. Baseball Game *(Easy)*

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.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
int calPoints(vector<string>& operations) {
    stack<int> scores;
    for (const string& op : operations) {
        if (op == "C") {
            scores.pop();
        } else if (op == "D") {

            int last = scores.top();
            scores.push(last * 2);
        } else if (op == "+") {
            int last = scores.top();
            scores.pop();
            int second_last = scores.top();
            scores.push(last);
            scores.push(last + second_last);
        } else {
            scores.push(stoi(op));
        }
    }
    int total = 0;
    while (!scores.empty()) {
        total += scores.top();
        scores.pop();
    }
    return total;
}
int main() {
    int n;
    cout << "Enter the number of operations: ";
    cin >> n;
    vector<string> operations(n);
    cout << "Enter the operations: ";
    for (int i = 0; i < n; ++i) {
        cin >> operations[i];
    }
    int result = calPoints(operations);
    cout << "The final score is: " << result << endl;
    return 0;
}
```

Output:



```
Enter the number of operations: 5
Enter the operations: 5 2 C D +
The final score is: 30
```

12. Remove Linked List Elements *(Easy)*

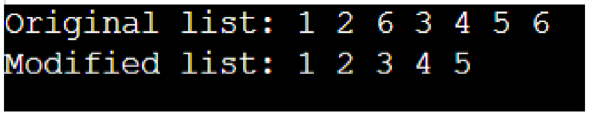
Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return *the new head*.

Implementation/Code:

```
#include <iostream>
#include<vector>
using namespace std;
struct ListNode {
   int val;
   ListNode* next;
   ListNode(int x) : val(x), next(NULL) {}
};
ListNode* removeElements(ListNode* head, int val) {

   ListNode* dummy = new ListNode(0);
   dummy->next = head;
   ListNode* current = dummy;
   while (current->next != NULL) {
      if (current->next->val == val) {
         ListNode* temp = current->next;
         current->next = current->next->next;
         delete temp;
      } else {

         current = current->next;
      }
   }
   ListNode* newHead = dummy->next;
   delete dummy;
   return newHead;
}
void printList(ListNode* head) {
   while (head != NULL) {
      cout << head->val << " ";
      head = head->next;
   }
   cout << endl;
}
```

```
ListNode* createList(const vector<int>& values) {
    if (values.empty()) return NULL;
    ListNode* head = new ListNode(values[0]);
    ListNode* current = head;
    for (int i = 1; i < values.size(); ++i) {
        current->next = new ListNode(values[i]);
        current = current->next;
    }
    return head;
}
int main() {
    vector<int> values = {1, 2, 6, 3, 4, 5, 6};
    int val = 6;
    ListNode* head = createList(values);
    cout << "Original list: ";
    printList(head);
    head = removeElements(head, val);
    cout << "Modified list: ";
    printList(head);
    return 0;
}
```

Output:



```
Original list: 1 2 6 3 4 5 6
Modified list: 1 2 3 4 5
```

13. Reverse Linked List *(Easy)*

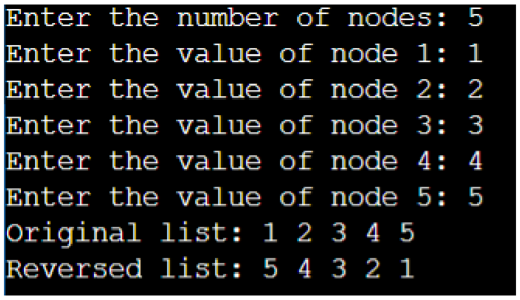
Given the head of a singly linked list, reverse the list, and return *the reversed list*.

Implementation/Code:

```
#include <iostream>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
```

```
};
ListNode* reverseList(ListNode* head) {
    ListNode* prev = NULL;
    ListNode* current = head;
    ListNode* next = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    return prev;
}
void printList(ListNode* head) {
    while (head != NULL) {
        cout << head->val << " ";
        head = head->next;
    }
    cout << endl;
}
ListNode* createList(int n) {
    if (n == 0) return NULL;
    int val;
    cout << "Enter the value of node 1: ";
    cin >> val;
    ListNode* head = new ListNode(val);
    ListNode* current = head;
    for (int i = 2; i <= n; ++i) {
        cout << "Enter the value of node " << i << ": ";
        cin >> val;
        current->next = new ListNode(val);
        current = current->next;
    }
    return head;
}
int main() {
    int n;
    cout << "Enter the number of nodes: ";
    cin >> n;
    ListNode* head = createList(n);
    cout << "Original list: ";
    printList(head);
    head = reverseList(head);
    cout << "Reversed list: ";
    printList(head);
    return 0;
}
```

Output:



```
Enter the number of nodes: 5
Enter the value of node 1: 1
Enter the value of node 2: 2
Enter the value of node 3: 3
Enter the value of node 4: 4
Enter the value of node 5: 5
Original list: 1 2 3 4 5
Reversed list: 5 4 3 2 1
```

14. Valid Sudoku *(Medium)*

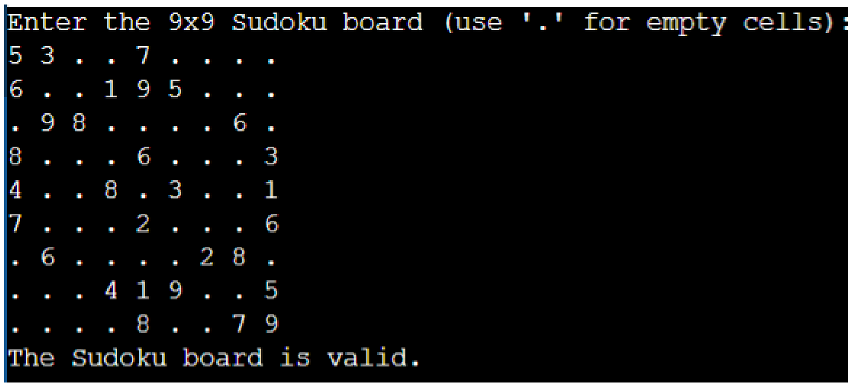
Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated according to the following rules:

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.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <unordered_set>
using namespace std;
class Solution {
public:
    bool isValidSudoku(vector<vector<char>>& board) {
        unordered_set<char> rows[9];
        unordered_set<char> cols[9];
        unordered_set<char> boxes[9];
        for (int r = 0; r < 9; ++r) {
            for (int c = 0; c < 9; ++c) {
                if (board[r][c] == '.') {
                    continue;
                }
                char value = board[r][c];
                int boxIndex = (r / 3) * 3 + (c / 3);
                if (rows[r].count(value) || cols[c].count(value) || boxes[boxIndex].count(value))
{
                    return false;
                }
                rows[r].insert(value);
                cols[c].insert(value);
                boxes[boxIndex].insert(value);
            }
        }

        return true;
    }
};

int main() {
    vector<vector<char>> board(9, vector<char>(9));
    cout << "Enter the 9x9 Sudoku board (use '.' for empty cells):" << endl;

    for (int i = 0; i < 9; ++i) {
        for (int j = 0; j < 9; ++j) {
            cin >> board[i][j];
        }
    }
    Solution solution;
    bool isValid = solution.isValidSudoku(board);

    if (isValid) {
        cout << "The Sudoku board is valid." << endl;
    } else {
        cout << "The Sudoku board is invalid." << endl;
    }
    return 0;
}
```

Output:



```
Enter the 9x9 Sudoku board (use '.' for empty cells):
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
The Sudoku board is valid.
```

15. Jump Game II *(Medium)*

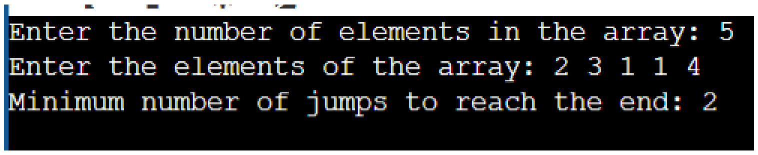
You are given a 0-indexed array of integers nums of length n. You are initially positioned at nums[0].

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 <= j <= 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].

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
class Solution {
public:
    int jump(vector<int>& nums) {
        int near = 0, far = 0, jumps = 0;
        while (far < nums.size() - 1) {
            int farthest = 0;
            for (int i = near; i <= far; i++) {
                farthest = max(farthest, i + nums[i]);
            }
            if (farthest <= far) {
                return -1;
            }
            near = far + 1;
            far = farthest;
            jumps++;
        }
        return jumps;
    }
};
int main() {
    int n;
    cout << "Enter the number of elements in the array: ";
    cin >> n;
    vector<int> nums(n);
    cout << "Enter the elements of the array: ";
    for (int i = 0; i < n; ++i) {
        cin >> nums[i];
    }
    Solution solution;
    int result = solution.jump(nums);

    if (result == -1) {
        cout << "It's not possible to reach the end of the array." << endl;
    } else {
        cout << "Minimum number of jumps to reach the end: " << result << endl;
    }
    return 0;
}
```

Output:



```
Enter the number of elements in the array: 5
Enter the elements of the array: 2 3 1 1 4
Minimum number of jumps to reach the end: 2
```

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:

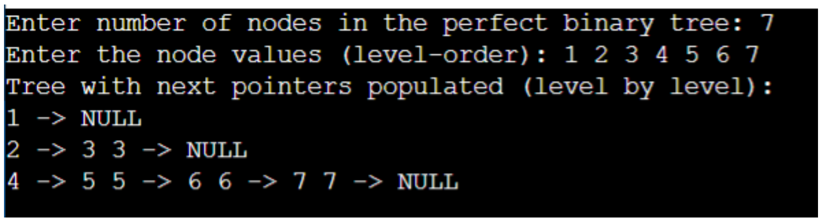
```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

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.

Implementation/Code:

```
#include <iostream>
#include <queue>
using namespace std;
struct Node {
    int val;
    Node* left;
    Node* right;
    Node* next;
    Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}
};
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return nullptr;
        queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            Node* rightNode = nullptr;
            for (int i = q.size(); i > 0; i--) {
                Node* cur = q.front();
                q.pop();
                cur->next = rightNode;
                rightNode = cur;
                if (cur->right) {
                    q.push(cur->right);
                    q.push(cur->left);
                }
            }
        }
        return root;
    }
};
void printTreeWithNext(Node* root) {
    while (root) {
        Node* cur = root;
        while (cur) {
            cout << cur->val << " -> " << (cur->next ? to_string(cur->next->val) : "NULL")
<< " ";
            cur = cur->next;
        }
        cout << endl;
        root = root->left;
    }
}
Node* buildTree(const vector<int>& values) {
    if (values.empty()) return nullptr;
    Node* root = new Node(values[0]);
    queue<Node*> q;
    q.push(root);
    int i = 1;
    while (i < values.size()) {
        Node* cur = q.front();
        q.pop();
        cur->left = new Node(values[i++]);
        q.push(cur->left);
        if (i < values.size()) {
            cur->right = new Node(values[i++]);
            q.push(cur->right);
        }
    }
    return root;
}
int main() {
    int n;
    cout << "Enter number of nodes in the perfect binary tree: ";
    cin >> n;
    cout << "Enter the node values (level-order): ";
    vector<int> values(n);
    for (int& x : values) cin >> x;
    Node* root = buildTree(values);
    Solution solution;
    root = solution.connect(root);
    cout << "Tree with next pointers populated (level by level): " << endl;
    printTreeWithNext(root);
    return 0;
}
```

Output:



```
Enter number of nodes in the perfect binary tree: 7
Enter the node values (level-order): 1 2 3 4 5 6 7
Tree with next pointers populated (level by level):
1 -> NULL
2 -> 3 3 -> NULL
4 -> 5 5 -> 6 6 -> 7 7 -> NULL
```

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.

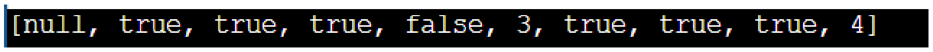
Implementation/Code:

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
class MyCircularQueue {
private:
    vector<int> queue;
    int front, rear, capacity, size;
public:
    MyCircularQueue(int k) {
        capacity = k;
        queue.resize(k, -1);
        front = 0;
        rear = -1;
        size = 0;
    }
    bool enQueue(int value) {
        if (isFull()) return false;
        rear = (rear + 1) % capacity;
        queue[rear] = value;
        size++;
        return true;
    }
    bool deQueue() {
        if (isEmpty()) return false;
```

```
            queue[front] = -1;
            front = (front + 1) % capacity;
            size--;
            return true;
        }
        int Front() {
            return isEmpty() ? -1 : queue[front];
        }
        int Rear() {
            return isEmpty() ? -1 : queue[rear];
        }
        bool isEmpty() {
            return size == 0;
        }
        bool isFull() {
            return size == capacity;
        }
    };
    int main() {
        vector<string> operations = {"MyCircularQueue", "enQueue", "enQueue", "enQueue",
    "enQueue", "Rear", "isFull", "deQueue", "enQueue", "Rear"};
        vector<vector<int>> arguments = {{3}, {1}, {2}, {3}, {4}, {}, {}, {}, {4}, {}};
        vector<string> results;
        MyCircularQueue* circularQueue = nullptr;
        for (size_t i = 0; i < operations.size(); ++i) {
            if (operations[i] == "MyCircularQueue") {
                circularQueue = new MyCircularQueue(arguments[i][0]);
                results.push_back("null");
            } else if (operations[i] == "enQueue") {
                bool res = circularQueue->enQueue(arguments[i][0]);
                results.push_back(res ? "true" : "false");
            } else if (operations[i] == "deQueue") {
                bool res = circularQueue->deQueue();
                results.push_back(res ? "true" : "false");
            } else if (operations[i] == "Front") {
                int res = circularQueue->Front();
                results.push_back(to_string(res));
            } else if (operations[i] == "Rear") {
```

```
            int res = circularQueue->Rear();
            results.push_back(to_string(res));
        } else if (operations[i] == "isEmpty") {
            bool res = circularQueue->isEmpty();
            results.push_back(res ? "true" : "false");
        } else if (operations[i] == "isFull") {
            bool res = circularQueue->isFull();
            results.push_back(res ? "true" : "false");
        }
    }
    cout << "[";
    for (size_t i = 0; i < results.size(); ++i) {
        cout << results[i];
        if (i != results.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    return 0;
}
```

Output:



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

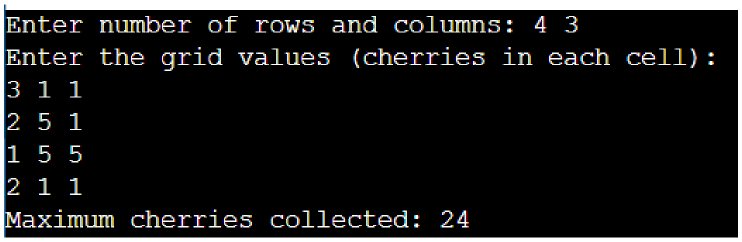
18. Cherry Pickup II *(Hard)*

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.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
using namespace std;
class Solution {
public:
    int rows, cols;
    int dp[70][70][70];
    int cherryPickup(vector<vector<int>>& grid) {
        rows = grid.size();
        cols = grid[0].size();
        memset(dp, -1, sizeof(dp));
        return dfs(grid, 0, 0, cols - 1);
    }
    int dfs(vector<vector<int>>& grid, int r, int c1, int c2) {
        if (c1 < 0 || c1 >= cols || c2 < 0 || c2 >= cols) return 0;
        if (dp[r][c1][c2] != -1) return dp[r][c1][c2];
        int cherries = grid[r][c1];
        if (c1 != c2) cherries += grid[r][c2];
        if (r == rows - 1) return cherries;
        int maxCherries = 0;
        for (int move1 = -1; move1 <= 1; ++move1) {
            for (int move2 = -1; move2 <= 1; ++move2) {
                maxCherries = max(maxCherries, dfs(grid, r + 1, c1 + move1, c2 + move2));
            }
        }
        return dp[r][c1][c2] = cherries + maxCherries;
    }
};
int main() {
    Solution solution;
    int rows, cols;
    cout << "Enter number of rows and columns: ";
    cin >> rows >> cols;
    vector<vector<int>> grid(rows, vector<int>(cols));
    cout << "Enter the grid values (cherries in each cell):\n";
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            cin >> grid[i][j];
        }
    }
    int maxCherries = solution.cherryPickup(grid);
    cout << "Maximum cherries collected: " << maxCherries << endl;

    return 0;
}
```

Output:



```
Enter number of rows and columns: 4 3
Enter the grid values (cherries in each cell):
3 1 1
2 5 1
1 5 5
2 1 1
Maximum cherries collected: 24
```

19. Maximum Twin Sum of Linked List *(Very Hard)*

In a linked list of size n, where n is **even**, the ith node (**0-indexed**) of the linked list is known as the **twin** of the (n-1-i)th node, if 0 <= i <= (n / 2) - 1.

- For example, if n = 4, then node 0 is the twin of node 3, and node 1 is the twin of node 2. These are the only nodes with twins for n = 4.

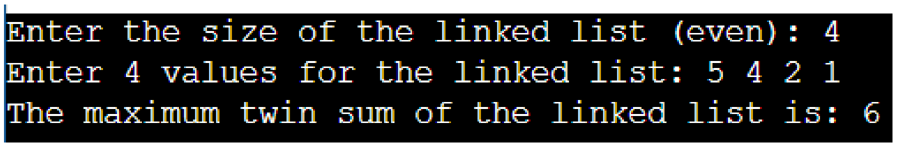
The **twin sum** is defined as the sum of a node and its twin.
Given the head of a linked list with even length, return *the **maximum twin sum** of the linked list*.

Implementation/Code:

```
#include <iostream>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(nullptr) {}
};
class Solution {
```

```
public:
    int pairSum(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head;
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }
        ListNode* prev = nullptr;
        while (slow) {
            ListNode* nextNode = slow->next;
            slow->next = prev;
            prev = slow;
            slow = nextNode;
        }
        int maxTwinSum = 0;
        while (prev) {
            maxTwinSum = max(maxTwinSum, head->val + prev->val);
            head = head->next;
            prev = prev->next;
        }
        return maxTwinSum;
    }
};
ListNode* createLinkedList(int n) {
    cout << "Enter " << n << " values for the linked list: ";
    int val;
    cin >> val;
    ListNode* head = new ListNode(val);
    ListNode* current = head;
    for (int i = 1; i < n; i++) {
        cin >> val;
        current->next = new ListNode(val);
        current = current->next;
    }
    return head;
}
int main() {
    int n;
    cout << "Enter the size of the linked list (even): ";
    cin >> n;
    if (n % 2 != 0) {
        cout << "The size of the linked list must be even." << endl;
        return 0;
    }
    ListNode* head = createLinkedList(n);
    Solution solution;
    int result = solution.pairSum(head);
    cout << "The maximum twin sum of the linked list is: " << result << endl;
    return 0;
}
```

Output:



```
Enter the size of the linked list (even): 4
Enter 4 values for the linked list: 5 4 2 1
The maximum twin sum of the linked list is: 6
```