

Name:- Sushant Raj

UID:- 22BCS10033

Section:- 22KPIT-901-A

## DAY-2

Q.1. 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.

Program Code:-

```
#include <iostream> using namespace
std; int majorityElement(int nums[], int
n) {    int candidate = nums[0];    int
count = 1;    for (int i = 1; i < n; i++)
{        if (nums[i] == candidate)
{            count++;
        } else {
            count--;
        }        if (count == 0)
{            candidate =
nums[i];            count = 1;
        }
    }    return
candidate;
```

```

} int main()

{

    int n;    cout << "Enter the size of the array: ";    cin >> n;    int*

nums = new int[n]; // Dynamically allocating memory for the array

cout << "Enter the elements of the array: ";    for (int i = 0; i < n; i++)

{        cin >> nums[i];

    }


    int result = majorityElement(nums, n);    cout <<

"The majority element is: " << result << endl;    delete[]

nums; // Deallocate the memory    return 0; }

```

Output:-

```

Output
Enter the size of the array: 10
Enter the elements of the array: 2 6 2 6 2 8 1 4 3 8
The majority element is: 8

=== Code Execution Successful ===

```

Q.2. 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.

Program Code:- #include

<iostream> using

namespace std; struct

ListNode {

int val;

ListNode\* next;

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

};

ListNode\* removeElements(ListNode\* head, int val)

{ ListNode\* dummy = new ListNode(0);

dummy->next = head;

ListNode\* current = dummy;

while (current->next != nullptr) { if

(current->next->val == val)

{ current->next =

current->next->next;

} else { current =

current->next;

```
    }  
}
```

```
    ListNode* newHead = dummy->next;  
  
    delete dummy;    return newHead;  
  
}
```

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

```
int main() {    int n, val;    cout <<  
  
    "Enter the number of nodes: ";    cin >>  
  
    n;  
  
    ListNode* head = nullptr;    ListNode* tail =  
  
    nullptr;    cout << "Enter the elements of the
```

```

linked list: ";    for (int i = 0; i < n; i++) {        int
nodeVal;        cin >> nodeVal;

        ListNode* newNode = new ListNode(nodeVal);        if (head == nullptr)

{            head = newNode;            tail = newNode;

        } else

{            tail->next =

newNode;            tail =

newNode;

        }

    }

    cout << "Enter the value to remove: ";

    cin >> val;    head =

removeElements(head, val);    cout <<

"Linked list after removal: ";

    printList(head);    return 0;

}Output:-

```

### Output

```
Enter the number of nodes: 5
Enter the elements of the linked list: 1 2 3 4 5
Enter the value to remove: 3
Linked list after removal: 1 2 4 5
```

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

Q.3. 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]`.

Program Code:-

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

minJumps(int nums[], int n) {    if (n <= 1) return 0;

int jumps = 0;    int currentEnd = 0;    int

currentFarthest = 0;    for (int i = 0; i < n - 1; i++)

{        currentFarthest = max(currentFarthest, i +

nums[i]);        if (i == currentEnd) {            jumps++;

currentEnd = currentFarthest;            if (currentEnd >=

n - 1) break;

        }}

return jumps;
```

```

} int main()

{

    int n;    cout << "Enter the size of the

array: ";    cin >> n;    int* nums = new

int[n];    cout << "Enter the elements of the

array: ";    for (int i = 0; i < n; i++)

{        cin >> nums[i];

    }

    int result = minJumps(nums, n);    cout << "Minimum number of jumps to reach

the last index: " << result << endl;    delete[] nums;    return 0;

}Output:-

```

```

Output
Enter the size of the array: 5
Enter the elements of the array: 6 3 8 2 4
Minimum number of jumps to reach the last index: 1

=== Code Execution Successful ===

```

Q.4. 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.

```
Program Code:- #include <iostream> #include <algorithm> using
namespace std; int maxCherries(int** grid, int rows, int cols) {    int
dp[rows][cols][cols]; // DP table to store max cherries at each step    //
Initialize the DP table with -1 for non-visited states    for (int i = 0; i <
rows; i++) {        for (int j1 = 0; j1 < cols; j1++) {            for (int j2 = 0;
j2 < cols; j2++) {                dp[i][j1][j2] = -1;
            }
        }
    }

    dp[0][0][cols - 1] = grid[0][0] + grid[0][cols - 1];

    for (int i = 1; i < rows; i++) {        for (int j1 = 0; j1
```



```
< cols; j1++) {
    for (int j2 = 0; j2 < cols;
j2++) {

        if (dp[i - 1][j1][j2] != -1) { // If this state was reachable from the
previous row

            for (int dj1 = -1; dj1 <= 1; dj1++)

                for (int dj2 = -1; dj2 <= 1; dj2++)

                    int newJ1 = j1 + dj1;

int newJ2 = j2 + dj2;

            if (newJ1 >= 0 && newJ1 < cols && newJ2 >= 0 && newJ2 <
cols) {

                int cherries = grid[i][newJ1] + grid[i][newJ2];

                if (newJ1 == newJ2) cherries -= grid[i][newJ1]; // If both
robots are at the same cell, only one picks cherries

                dp[i][newJ1][newJ2] = max(dp[i][newJ1][newJ2], dp[i -
1][j1][j2] + cherries);

            }

        }

    }

}

}
```

```

    }

    int maxCherries = 0;    for (int j1 = 0; j1 < cols; j1++)

    {        for (int j2 = 0; j2 < cols; j2++) {            maxCherries =

max(maxCherries, dp[rows - 1][j1][j2]);

        }

    }

    return maxCherries;

}

```

```

int main() {    int rows, cols;    cout << "Enter the

number of rows and columns: ";    cin >> rows >>

cols;

```

```

    int** grid = new int*[rows];    cout <<

"Enter the grid (cherry counts):\n";    for (int

i = 0; i < rows; i++) {        grid[i] = new

int[cols];        for (int j = 0; j < cols; j++)

    {            cin >> grid[i][j];

        }

```

```
}
```

```
int result = maxCherries(grid, rows, cols);    cout <<
```

```
"Maximum cherries collected: " << result << endl;    for (int
```

```
i = 0; i < rows; i++) {        delete[] grid[i];
```

```
    }    delete[]
```

```
grid;
```

```
return 0;
```

```
}Output:-
```

### Output

```
Enter the number of rows and columns: 3 3
```

```
Enter the grid (cherry counts):
```

```
1 2 3
```

```
4 5 6
```

```
7 8 9
```

```
Maximum cherries collected: 32
```

Q.5. 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 Program

Code:- #include <iostream>

#include <numeric> #include <algorithm> using

namespace std; bool canAssignJobs(int jobs[], int n, int k,

int max\_time) { int workers = 1, current\_time = 0;

for (int i = 0; i < n; i++) { if (current\_time + jobs[i] >

max\_time) { workers++; current\_time =

jobs[i]; if (workers > k) return false;

} else

{ current\_time +=

jobs[i];

}

}

return true;

}

int minimumTimeRequired(int jobs[], int n, int k)

{ int left = \*max\_element(jobs, jobs + n); int

right = accumulate(jobs, jobs + n, 0); while (left

```

< right) {      int mid = left + (right - left) / 2;

if (canAssignJobs(jobs, n, k, mid)) {      right =

mid;      } else {      left = mid + 1;

      }      }

return left; }

int main() {

    int n, k;    cin >> n >> k;

    int jobs[n];    for (int i = 0;

i < n; i++) {      cin >>

jobs[i];

    }

    int result = minimumTimeRequired(jobs, n, k);

    cout << result << endl;    return 0;

}
Output:-

```



```

4 2
5 3 4 1
8

```