Week 2 & 3 Assignment

Name – Sumit Phalke

Class – TY CSE

PRN – 23510023

Q.1 Employee Rating (Workload Analysis)

Question1:
You are given an array `workload[]` where each element represents the hours worked by an employee on a particular day. The employee's **rating** is the maximum number of consecutive days where they worked more than 6 hours.

Algorithm:

1. Start with the first day.
2. Keep a counter for consecutive days with more than 6 hours.
3. If the employee worked more than 6 hours, increase the counter.
4. If they worked 6 hours or less, reset the counter to 0.
5. Track the maximum value of the counter throughout the process.
6. The result is the highest number of consecutive days above 6 hours.

Input Screenshot & Output Screenshot:

```
int employeeRating(vector<int>& workload) {
    int n = workload.size();
    int maxDays = 0;

    for (int i = 0; i < n; i++) {
        int count = 0;
        for (int j = i; j < n; j++) {
            if (workload[j] > 6) {
                count++;
                maxDays = max(maxDays, count);
            } else {
                break;
            }
        }
    }
    return maxDays;
}

void question1() {
    cout << "Question 1: Employee rating based on workload" << endl;
```

```
    vector<int> workload = {7, 8, 5, 9, 10, 11, 4, 7, 8};

    cout << "Workload array: ";
    for (int hrs : workload) cout << hrs << " ";
    cout << endl;

    int rating = employeeRating(workload);
    cout << "Employee Rating (Max consecutive days > 6 hrs): " << rating << endl << endl;
}
```

```
Question 1: Employee rating based on workload
Workload array: 7 8 5 9 10 11 4 7 8
Employee Rating (Max consecutive days > 6 hrs): 3
```

Q.2 Candy Distribution in Boxes

Question:
You have `N` boxes and `K` candies. Candies are placed in a zig-zag fashion: forward from box 1 to box N, then backward from box N-1 to box 1, then forward again, until all candies are placed. Find the box number where the K-th candy will go.

Algorithm:

1. Start with the first box.
2. Place candies one by one while moving in the current direction.
3. If the last box is reached, reverse the direction.
4. Repeat until the K-th candy is placed.
5. The box where the K-th candy is placed is the answer.

Input Screenshot & Output Screenshot:

```
int findBoxBruteForce(int N, int K) {
    int currentBox = 1;
    int direction = 1;

    for (int candy = 1; candy <= K; candy++) {
        if (candy == K) {
            return currentBox;
        }
        currentBox += direction;
        if (currentBox > N) {
            currentBox = N - 1;
            direction = -1;
        } else if (currentBox < 1) {
```

```cpp
            currentBox = 2;
            direction = 1;
        }
    }
    return currentBox;
}

void question2() {
    cout << "Question 2: Find box index for K-th candy" << endl;

    int N = 4;
    int K = 11;

    cout << "Number of boxes: " << N << endl;
    cout << "K-th candy: " << K << endl;

    int boxIndex = findBoxBruteForce(N, K);
    cout << "The K-th candy is placed in box number: " << boxIndex <<
endl << endl;
}
```

```
Question 2: Find box index for K-th candy
Number of boxes: 4
K-th candy: 11
The K-th candy is placed in box number: 3
```

Q.3 Tower of Hanoi

Question:
Solve the Tower of Hanoi problem for `n` disks. Move all disks from source rod to destination rod following the rules:

- Only one disk can be moved at a time.
- A larger disk cannot be placed on a smaller disk.

Algorithm:

1. If there is only one disk, move it directly from source to destination.
2. Otherwise:
   - Move the top `n-1` disks from source to auxiliary rod.
   - Move the nth (largest) disk from source to destination rod.
   - Move the `n-1` disks from auxiliary to destination rod.

3. Repeat the steps recursively until all disks are moved.

Input Screenshot & Output Screenshot:

```
int movedisks(int n , char from , char to , char aux ){
  if(n == 1){
    cout<<"move disk 1 from "<<from<<" to "<<to<<" rod"<<endl;
    return 1 ;
  }
  int steps = 0 ;
  steps+=movedisks(n-1,from , aux , to );
  cout<<"move disk "<<n<<" from "<<from<<" to "<<to<<" rod"<<endl;
  steps++;
  steps+=movedisks(n-1,aux,to,from);
  return steps;
}

void question3(){
    int n = 3 ;
    cout<<"The number of a Disks are: "<<n<<endl;
   int total  = movedisks(n,'A','B','C');
   cout<<"Total steps required to solve Tower of Hanoi for "<<n<< "
Disks: "<<total<<endl<<endl;
}
```

```
The number of a Disks are: 3
move disk 1 from A to B rod
move disk 2 from A to C rod
move disk 1 from B to C rod
move disk 3 from A to B rod
move disk 1 from C to A rod
move disk 2 from C to B rod
move disk 1 from A to B rod
Total steps required to solve Tower of Hanoi for 3 Disks: 7
```

Q.4 Frog in a Square

Question:
A frog starts at the origin (0,0) and can either move right, up, or stay still every second. After `T` seconds, a villager reports the frog is inside a square with bottom-left corner `(X, Y)` and side length `s`. Find the number of possible integer coordinate points within the square that the frog can occupy after exactly `T` seconds.

Algorithm:

1. Loop through all integer points inside the square `(X, Y)` to `(X+s, Y+s)`.
2. For each point `(i, j)`, check if it can be reached within exactly `T` steps.
    - The frog can only move `i+j` steps at minimum.
    - If `i+j <= T`, then the frog can be there.
3. Count all such valid points.

Input Screenshot & Output Screenshot:

```
void question4(){
    int x=2, y=2, s=4, t=5, count=0;

    cout<<"Enter the Start X coordinate: "<<x<<endl;

    cout<<"Enter the Start Y coordinate: "<<y<<endl;

    cout<<"Enter the Side Length: "<<s<<endl;

    cout<<"Enter the Time in Seconds: "<<t<<endl;

    for(int i=x;i<=x+s;i++){
        for(int j=y;j<=y+s;j++){
            if(i+j<=t){
                count++;
            }
        }
    }
    cout<<"Total Points secured by the frog: "<<count<<endl<<endl;
}
```

```
Enter the Start X coordinate: 2
Enter the Start Y coordinate: 2
Enter the Side Length: 4
Enter the Time in Seconds: 5
Total Points secured by the frog: 3
```

Q.5 Lost Package Tracker

Question:
You are given a sorted list of timestamps. Sometimes values are missing due to scanning errors. Find the first missing timestamp.

Algorithm:

1. Start from the first timestamp.
2. Compare each timestamp with the expected next value.
3. If the current timestamp equals the expected value, increase expected by 1.
4. If the current timestamp is larger than expected, the missing value is found.
5. Output the missing timestamp.

Input Screenshot & Output Screenshot:

```
void question5(){
    int size = 5;
    vector<int> time = {3, 4, 5, 7, 8};

    sort(time.begin(), time.end());
    int exp = time[0];

    for (int i = 0; i < size; i++) {
        if (time[i] < exp) continue;
        if (time[i] == exp) {
            exp++;
        } else {
            break;
        }
    }
    cout << "The missing time is " << exp << endl << endl;

}
```

```
The missing time is 6
```

Q.6 Linear Search

Question:
Search for a given element (key) in an unsorted array using **linear search**.

Algorithm:

1. Start from the first element.
2. Compare each element with the key.
3. If a match is found, print the index and stop.
4. If the end of the array is reached without finding the key, print “Key not found”.

Input Screenshot & Output Screenshot:

```cpp
void question6(){
    int n = 6;
    vector<int> arr = {10, 20, 30, 40, 50, 60};
    int key = 40;

    bool found = false;
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            cout << "Key found at index " << i << endl << endl;
            found = true;
            break;
        }
    }
    if (!found) cout << "Key not found" << endl << endl;
}
```

```
Key found at index 3
```

Q.7 Binary Search

Question:
Search for a given element (key) in a sorted array using **binary search**.

Algorithm:

1. Set `low = 0` and `high = n-1`.
2. Find the middle index `mid = (low + high) / 2`.
3. If `arr[mid] == key`, return the index.
4. If `arr[mid] < key`, search the right half (`low = mid + 1`).
5. Otherwise, search the left half (`high = mid - 1`).
6. Repeat until the element is found or search space is empty.

Input Screenshot & Output Screenshot:

```
void question7(){
    int n = 7;
    vector<int> arr = {5, 10, 15, 20, 25, 30, 35};
    int key = 25;
    int low = 0, high = n - 1;
    bool found = false;

    while (low <= high) {
        int mid = (low + high) / 2;

        if (arr[mid] == key) {
            cout << "Key found at index " << mid << endl << endl;
            found = true;
            break;
        } else if (arr[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    if (!found) cout << "Key not found"  << endl;

}
```

```
Key found at index 4
```

Q.8 Signal Drop Detector

Question:
A signal drop is defined as a strictly decreasing subsequence of at least 3 consecutive readings. Count the number of such drops in a given signal array.

Algorithm:

1. Start from the first element.
2. Keep a counter `len` for consecutive decreasing readings.
3. If the next reading is smaller, increase `len`.
4. If not, check if `len >= 3`; if yes, increase the drop count. Then reset `len = 1`.
5. At the end, check once more if a drop ends at the last element.
6. Return the number of drops.

Input Screenshot & Output Screenshot:

```
void question8(){
    int n = 10;
    vector<int> signal = {100, 95, 90, 85, 120, 110, 100, 99, 98, 150};

    int drops = 0;
    int len = 1;

    for (int i = 1; i < n; i++) {
        if (signal[i] < signal[i - 1]) {
            len++;
        } else {
            if (len >= 3) drops++;
            len = 1;
        }
    }
    if (len >= 3) drops++;

    cout << "Number of signal drops: " << drops  << endl;

}
```

```
Number of signal drops: 2
```