# Week 2 & 3 Assignment

Name – Sumit Phalke

Class – TY CSE

PRN – 23510023

## *Q1) Maximum Element in Bitonic Array*

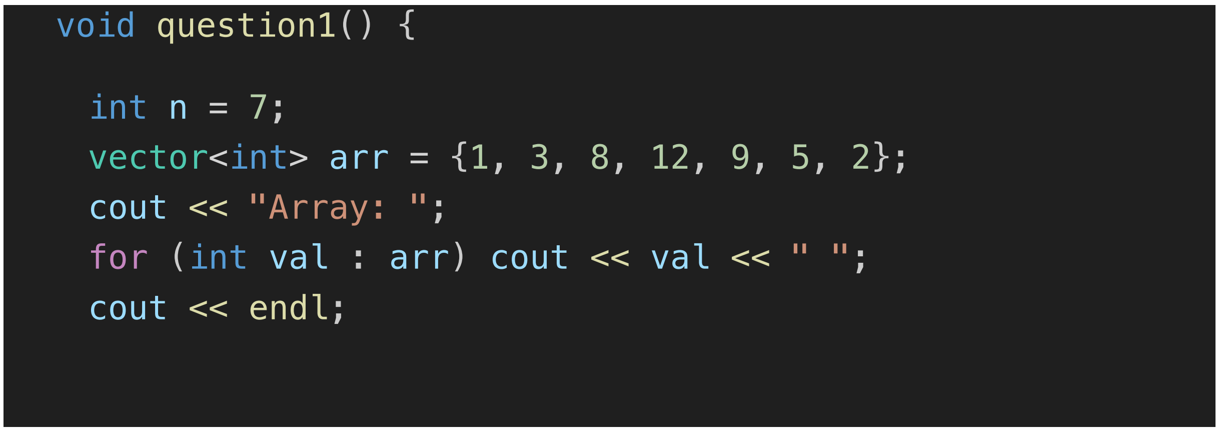
**Question:**
Implement an algorithm to find the maximum element in an array which is first increasing and then decreasing, with a time complexity of **O(log n)**.

**Algorithm:**

1. Start with two pointers, `low` and `high`, at the beginning and end of the array.
2. Repeatedly find the middle index `mid`.
3. If the middle element is greater than both its neighbors, then it is the maximum.
4. If the middle element is smaller than the next element, then the peak lies on the right half, so move `low` to `mid + 1`.
5. Otherwise, the peak lies on the left half, so move `high` to `mid - 1`.
6. Continue until the maximum element is found.

This approach works because the array first increases and then decreases, guaranteeing exactly one peak.

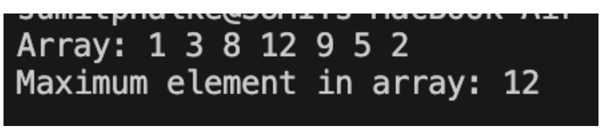
**Input Screenshot:**



```
void question1() {

  int n = 7;
  vector<int> arr = {1, 3, 8, 12, 9, 5, 2};
  cout << "Array: ";
  for (int val : arr) cout << val << " ";
  cout << endl;
```

```
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (mid > 0 && mid < n - 1 && arr[mid] > arr[mid -
1] && arr[mid] > arr[mid + 1]) {
            cout << "Maximum element in array: " <<
arr[mid] << endl;
            return;
        }
        else if (mid < n - 1 && arr[mid] < arr[mid + 1]) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    cout << "Invalid array input!" << endl;
}
```

**Output Screenshot:**



```
Array: 1 3 8 12 9 5 2
Maximum element in array: 12
```

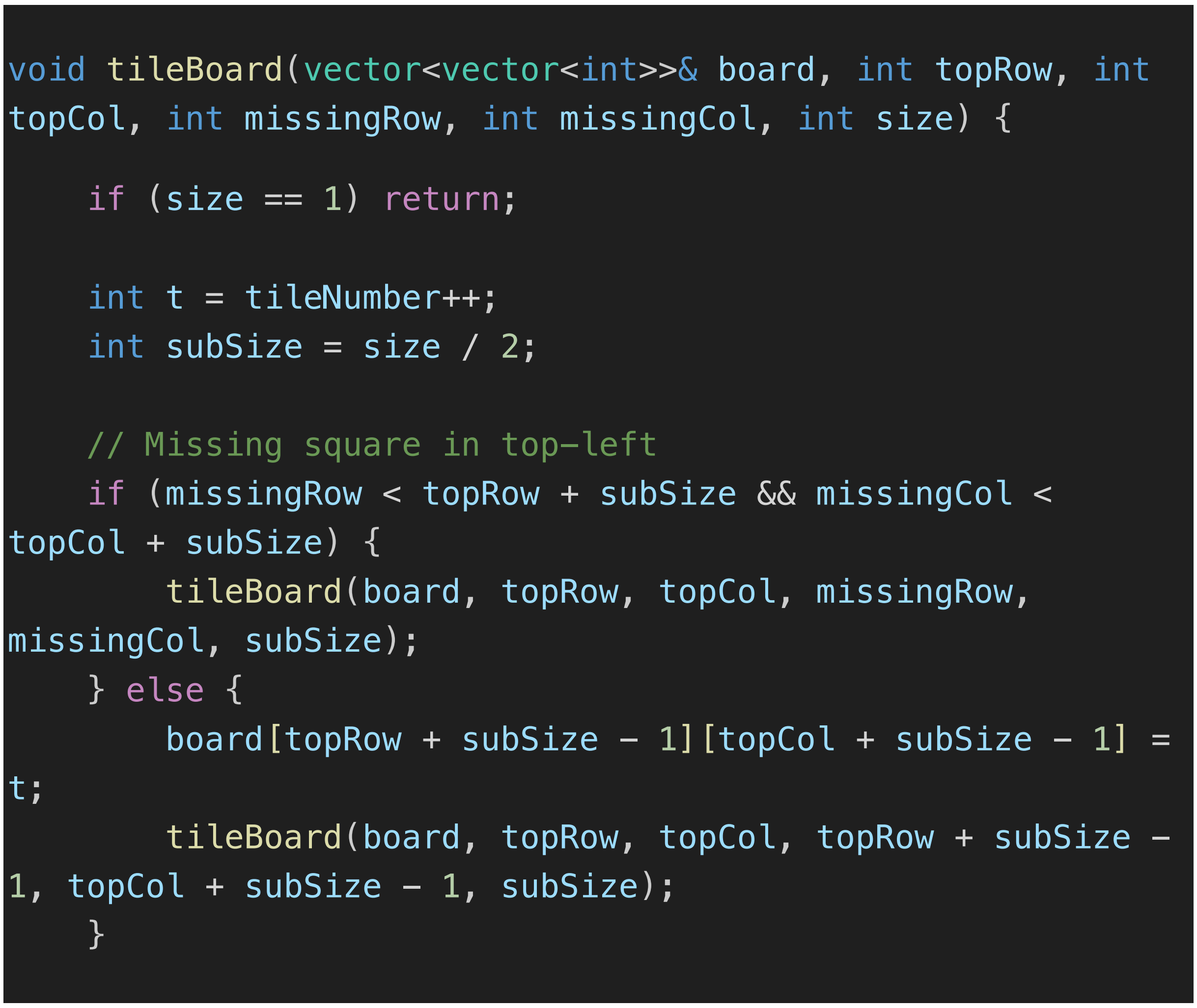
## *Q2) Tiling Problem (Board with One Missing Cell)*

**Question:**
Implement an algorithm for the **tiling problem**: Given an `n × n` board (where `n` is a power of 2), with one missing cell, tile the board completely using **L-shaped tiles**. Each L-shaped tile is a `2 × 2` square with one cell missing.

**Algorithm:**

1. The board is divided into four quadrants.
2. Identify in which quadrant the missing square lies.
3. Place one L-shaped tile at the center so that it covers one square from each of the other three quadrants (avoiding the quadrant with the actual missing square).
4. Recursively solve the problem for each quadrant:
    - The quadrant with the original missing square is solved as is.
    - For the other three quadrants, the newly placed L-shaped tile creates a "virtual missing square" which is used for recursion.
5. Repeat until the board is fully tiled (base case: `2 × 2` board).

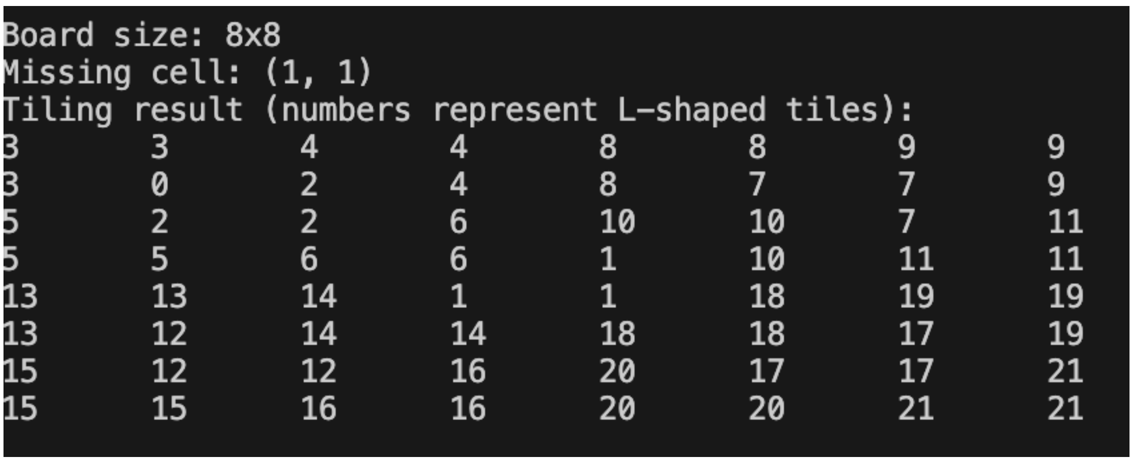
**Input Screenshot:**



```
void tileBoard(vector<vector<int>>& board, int topRow, int
topCol, int missingRow, int missingCol, int size) {

    if (size == 1) return;

    int t = tileNumber++;
    int subSize = size / 2;

    // Missing square in top-left
    if (missingRow < topRow + subSize && missingCol <
topCol + subSize) {
        tileBoard(board, topRow, topCol, missingRow,
missingCol, subSize);
    } else {
        board[topRow + subSize - 1][topCol + subSize - 1] =
t;
        tileBoard(board, topRow, topCol, topRow + subSize -
1, topCol + subSize - 1, subSize);
    }
    // Missing square in top-right
    if (missingRow < topRow + subSize && missingCol >=
topCol + subSize) {
        tileBoard(board, topRow, topCol + subSize,
missingRow, missingCol, subSize);
    } else {
        board[topRow + subSize - 1][topCol + subSize] = t;
        tileBoard(board, topRow, topCol + subSize, topRow +
subSize - 1, topCol + subSize, subSize);
    }

    // Missing square in bottom-left
    if (missingRow >= topRow + subSize && missingCol <
topCol + subSize) {
        tileBoard(board, topRow + subSize, topCol,
missingRow, missingCol, subSize);
    } else {
        board[topRow + subSize][topCol + subSize - 1] = t;
        tileBoard(board, topRow + subSize, topCol, topRow +
subSize, topCol + subSize - 1, subSize);
    }

    // Missing square in bottom-right
    if (missingRow >= topRow + subSize && missingCol >=
topCol + subSize) {
        tileBoard(board, topRow + subSize, topCol +
subSize, missingRow, missingCol, subSize);
    } else {
        board[topRow + subSize][topCol + subSize] = t;
        tileBoard(board, topRow + subSize, topCol +
subSize, topRow + subSize, topCol + subSize, subSize);
    }
}
```

```
void question2() {
    // Predefined input
    int n = 8;
    int missingRow = 1;
    int missingCol = 1;

    cout << "\nBoard size: " << n << "x" << n << endl;
    cout << "Missing cell: (" << missingRow << ", " <<
missingCol << ")\n";

    vector<vector<int>> board(n, vector<int>(n, 0));
    tileBoard(board, 0, 0, missingRow, missingCol, n);

    cout << "Tiling result (numbers represent L-shaped
tiles):\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << board[i][j] << "\t";
        }
        cout << "\n";
    }
}
```

**Output Screenshot:**



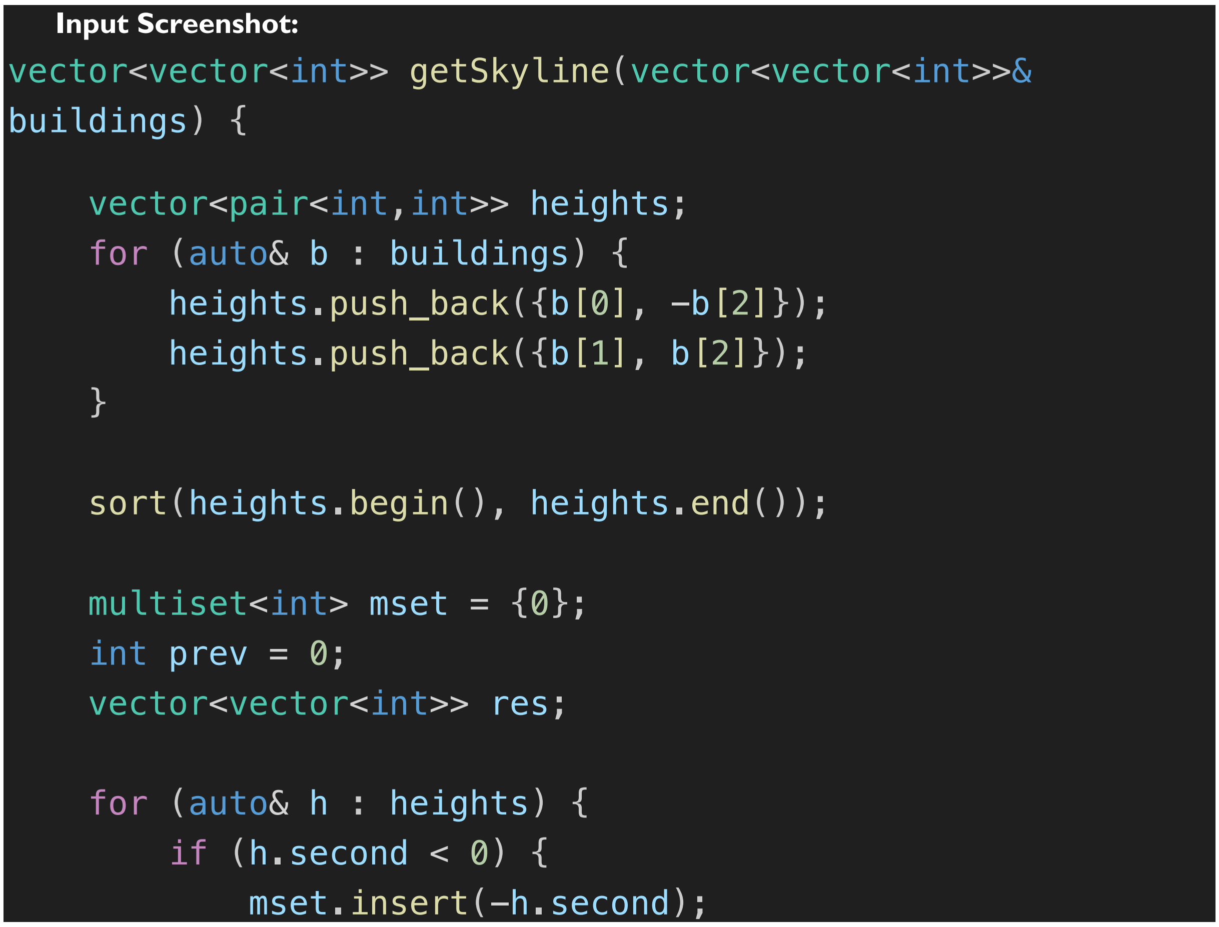
```
Board size: 8x8
Missing cell: (1, 1)
Tiling result (numbers represent L-shaped tiles):
3       3       4       4       8       8       9       9
3       0       2       4       8       7       7       9
5       2       2       6       10      10      7       11
5       5       6       6       1       10      11      11
13      13      14      1       1       18      19      19
13      12      14      14      18      18      17      19
15      12      12      16      20      17      17      21
15      15      16      16      20      20      21      21
```

## *Q3) Skyline Problem*

**Question:**
Implement an algorithm for the **Skyline Problem**: Given `n` rectangular buildings represented as `(left, right, height)`, compute the skyline formed by these buildings. The skyline is represented as key points `(x, height)` showing visible strips when viewing the buildings from the side. The algorithm must run in **O(n log n)** time.

**Algorithm:**

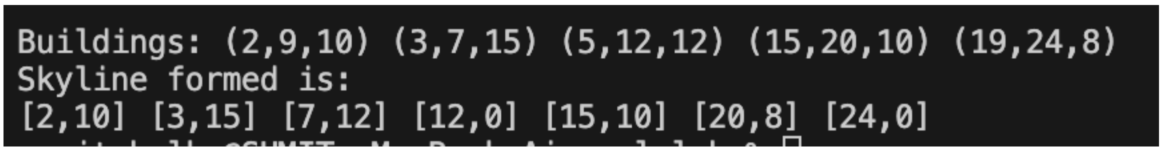
1. For each building, break it into two events:
    - A "start edge" at `left` with height `h`.
    - An "end edge" at `right` with height `h`.
2. Sort all edges by their x-coordinate. If two edges have the same x-coordinate, process start edges before end edges.
3. Use a **multiset (priority queue)** to keep track of active building heights.
    - When encountering a start edge, add the building height.
    - When encountering an end edge, remove the building height.
4. After each update, check the maximum height in the multiset:
    - If the maximum height changes compared to the previous height, record a skyline point `(x, height)`.
5. Continue until all edges are processed. The resulting points form the skyline.


**Input Screenshot:**

```
vector<vector<int>> getSkyline(vector<vector<int>>&
buildings) {

    vector<pair<int,int>> heights;
    for (auto& b : buildings) {
        heights.push_back({b[0], -b[2]});
        heights.push_back({b[1], b[2]});
    }

    sort(heights.begin(), heights.end());

    multiset<int> mset = {0};
    int prev = 0;
    vector<vector<int>> res;

    for (auto& h : heights) {
        if (h.second < 0) {
            mset.insert(-h.second);
```

```cpp
        } else {
            mset.erase(mset.find(h.second));
        }

        int cur = *mset.rbegin();
        if (cur != prev) {
            res.push_back({h.first, cur});
            prev = cur;
        }
    }
    return res;
}
// Q3) Skyline Problem
void question3() {
    vector<vector<int>> buildings = {
        {2, 9, 10},
        {3, 7, 15},
        {5, 12, 12},
        {15, 20, 10},
        {19, 24, 8}
    };

    cout << "\nBuildings: ";
    for (auto& b : buildings) {
        cout << "(" << b[0] << "," << b[1] << "," << b[2]
<< ") ";
    }
    cout << endl;

    vector<vector<int>> skyline = getSkyline(buildings);

    cout << "Skyline formed is:\n";
    for (auto& point : skyline) {
        cout << "[" << point[0] << "," << point[1] << "] ";
    }
    cout << endl;
}
```

**Output Screenshot:**



```
Buildings: (2,9,10) (3,7,15) (5,12,12) (15,20,10) (19,24,8)
Skyline formed is:
[2,10] [3,15] [7,12] [12,0] [15,10] [20,8] [24,0]
```