**Batch:T6**

**Practical No.3**

**Title of Assignment: Divide and conquer strategy**

**Student Name: Parshwa Herwade**

**Student PRN: 22510064**

**Q1)** Implement algorithm to Find the maximum element in an array which is first increasing and then decreasing, with Time Complexity *O(Logn).*

ANS.

Pseudocode:

Function findMaximum(arr, n):

left = 0

right = n - 1

While left <= right:

mid = left + (right - left) // 2

If mid > 0 and mid < n-1:

If arr[mid] > arr[mid-1] and arr[mid] > arr[mid+1]:

Return arr[mid]

ElseIf arr[mid-1] > arr[mid]:

right = mid - 1

Else:

left = mid + 1

ElseIf mid == 0:

Return max(arr[0], arr[1])

ElseIf mid == n-1:

Return max(arr[n-1], arr[n-2])

Return -1

CODE:

#include <iostream>

using namespace std;

int findMaximum(int arr[], int n) {

    int left = 0, right = n - 1;

    while (left <= right) {

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

        if (mid > 0 && mid < n - 1) {

            if (arr[mid] > arr[mid - 1] && arr[mid] > arr[mid + 1]) {

                return arr[mid];

            } else if (arr[mid - 1] > arr[mid]) {

                right = mid - 1;

            } else {

                left = mid + 1;

            }

        } else if (mid == 0) {

            return max(arr[0], arr[1]);

        } else if (mid == n - 1) {

            return max(arr[n - 1], arr[n - 2]);

        }

    }

    return -1;

}

int main() {

    int n;

    cout << "Enter the number of elements in the array: ";

    cin >> n;

    int arr[n];

    cout << "Enter the elements of the array: ";

    for (int i = 0; i < n; i++) {

        cin >> arr[i];

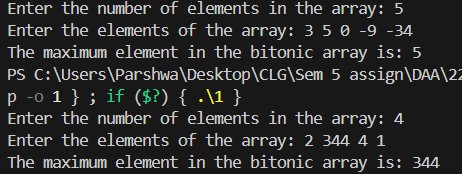
    }

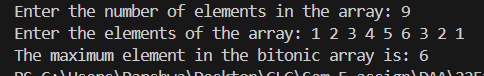
    int maxElement = findMaximum(arr, n);

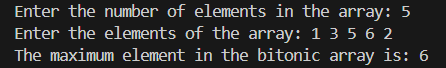
    cout << "The maximum element in the bitonic array is: " << maxElement << endl;

    return 0;

}

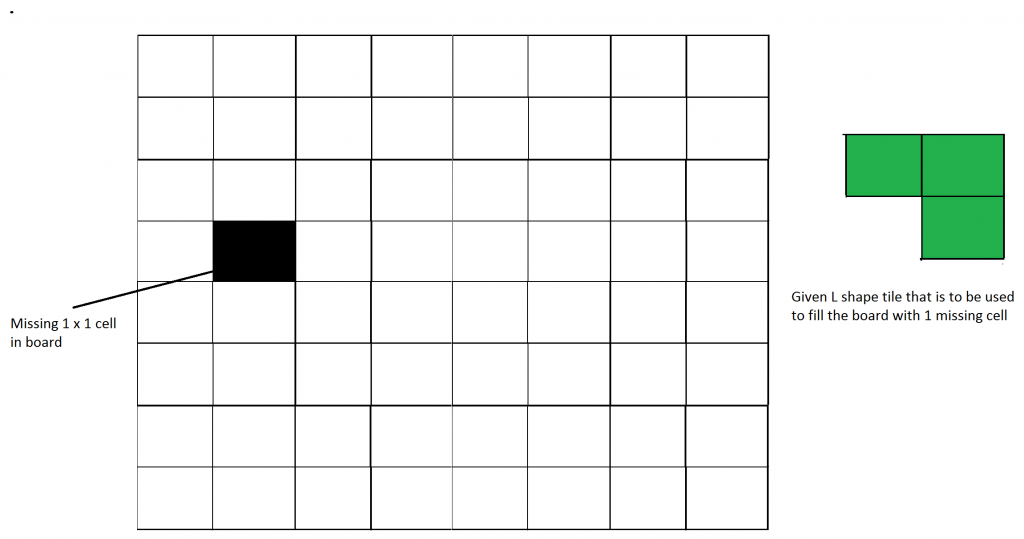
OUTPUT:  






**Complexity Analysis:**

* **Best Time Complexity:** O(1)
* **Worst Time Complexity:** O(log n)
* **Average Time Complexity:** O(log n)
* **Space Complexity:** O(1)

**Q2)** Implement algorithm for Tiling problem: Given an *n by n* board where n is of form *2k* where *k >= 1* (Basically n is a power of *2* with minimum value as *2*). The board has one missing cell (of size *1 x 1*). Fill the board using L shaped tiles. An *L* shaped tile is a *2 x 2* square with one cell of size *1×1* missing

ANS.

Pseudocode:  
Function tileBoard(board, topX, topY, missingX, missingY, size):

If size == 2:

Place an L-shaped tile covering the 3 other cells and return

halfSize = size // 2

Find the quadrant of the missing cell

Case 1: Missing cell in the top-left quadrant

Place an L-shaped tile at the center excluding the top-left cell

Recurse on the four quadrants

Case 2: Missing cell in the top-right quadrant

Place an L-shaped tile at the center excluding the top-right cell

Recurse on the four quadrants

Case 3: Missing cell in the bottom-left quadrant

Place an L-shaped tile at the center excluding the bottom-left cell

Recurse on the four quadrants

Case 4: Missing cell in the bottom-right quadrant

Place an L-shaped tile at the center excluding the bottom-right cell

Recurse on the four quadrants

CODE:

#include <iostream>

#define N 128

using namespace std;

int board[N][N];

int tile = 1;

void tileBoard(int topX, int topY, int missingX, int missingY, int size) {

    if (size == 2) {

        for (int i = topX; i < topX + size; i++) {

            for (int j = topY; j < topY + size; j++) {

                if (i != missingX || j != missingY) {

                    board[i][j] = tile;

                }

            }

        }

        tile++;

        return;

    }

    int halfSize = size / 2;

    if (missingX < topX + halfSize && missingY < topY + halfSize) {

        tileBoard(topX, topY, missingX, missingY, halfSize);

    } else {

        board[topX + halfSize - 1][topY + halfSize - 1] = tile;

        tileBoard(topX, topY, topX + halfSize - 1, topY + halfSize - 1, halfSize);

    }

    if (missingX < topX + halfSize && missingY >= topY + halfSize) {

        tileBoard(topX, topY + halfSize, missingX, missingY, halfSize);

    } else {

        board[topX + halfSize - 1][topY + halfSize] = tile;

        tileBoard(topX, topY + halfSize, topX + halfSize - 1, topY + halfSize, halfSize);

    }

    if (missingX >= topX + halfSize && missingY < topY + halfSize) {

        tileBoard(topX + halfSize, topY, missingX, missingY, halfSize);

    } else {

        board[topX + halfSize][topY + halfSize - 1] = tile;

        tileBoard(topX + halfSize, topY, topX + halfSize, topY + halfSize - 1, halfSize);

    }

    if (missingX >= topX + halfSize && missingY >= topY + halfSize) {

        tileBoard(topX + halfSize, topY + halfSize, missingX, missingY, halfSize);

    } else {

        board[topX + halfSize][topY + halfSize] = tile;

        tileBoard(topX + halfSize, topY + halfSize, topX + halfSize, topY + halfSize, halfSize);

    }

}

int main() {

    int n, missingX, missingY;

    cout << "Enter the size of the board (2^k): ";

    cin >> n;

    cout << "Enter the coordinates of the missing cell (x y): ";

    cin >> missingX >> missingY;

    tileBoard(0, 0, missingX, missingY, n);

    cout << "Tiled board:" << endl;

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            cout << board[i][j] << " ";

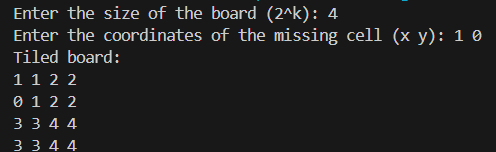
        }

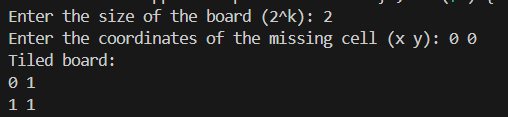
        cout << endl;

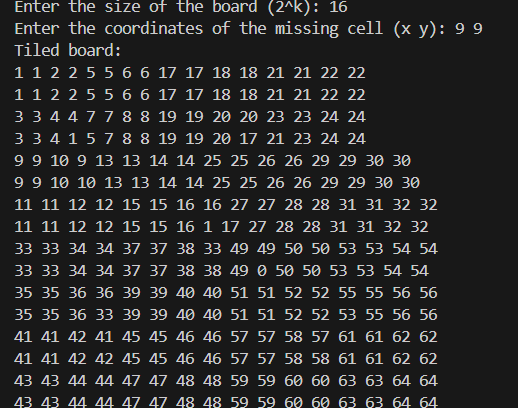
    }

    return 0;

}

OUTPUT:  






**Complexity Analysis:**

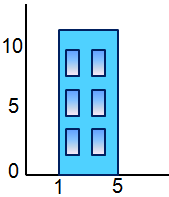
* **Best Time Complexity:** O(n^2)
* **Worst Time Complexity:** O(n^2)
* **Average Time Complexity:** O(n^2)
* **Space Complexity:** O(n^2) (For storing the board)

Q3) Implement algorithm for The Skyline Problem: Given n rectangular buildings in a 2-dimensional city, computes the skyline of these buildings, eliminating hidden lines. The main task is to view buildings from a side and remove all sections that are not visible.

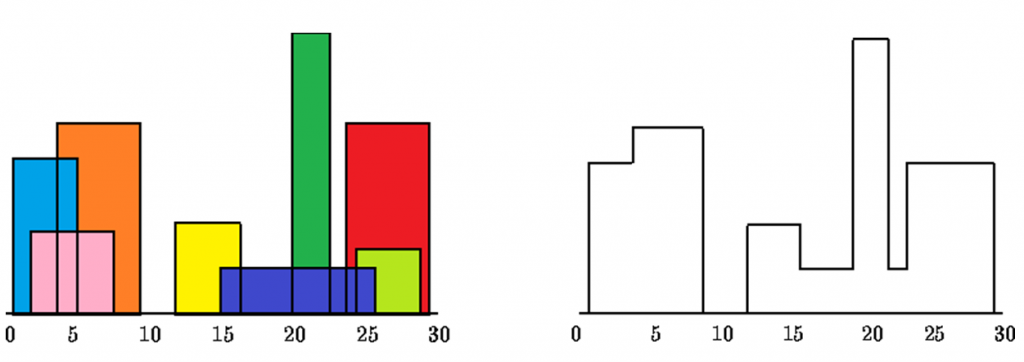
All buildings share common bottom and every **building**is represented by triplet (left, ht, right)

‘left’: is x coordinated of left side (or wall).  
‘right': is x coordinate of right side  
‘ht': is height of building.

For example, the building on right side is represented as *(1, 11, 5)*

[](http://www.geeksforgeeks.org/divide-and-conquer-set-7-the-skyline-problem/building/)

A **skyline**is a collection of rectangular strips. A rectangular **strip**is represented as a pair (left, ht) where left is x coordinate of left side of strip and ht is height of strip.



With Time Complexity *O(nLogn)*

Pseudocode:

Function mergeSkylines(leftSkyline, rightSkyline):

Initialize newSkyline

Merge the two skylines while maintaining the correct height

Function skyline(buildings, left, right):

If left == right:

Return a skyline with the single building

mid = (left + right) // 2

leftSkyline = skyline(buildings, left, mid)

rightSkyline = skyline(buildings, mid + 1, right)

Return mergeSkylines(leftSkyline, rightSkyline)

CODE:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

typedef pair<int, int> Point;

vector<Point> mergeSkylines(vector<Point>& leftSkyline, vector<Point>& rightSkyline) {

    vector<Point> mergedSkyline;

    int h1 = 0, h2 = 0, currentHeight = 0;

    int i = 0, j = 0;

    while (i < leftSkyline.size() && j < rightSkyline.size()) {

        int x1 = leftSkyline[i].first;

        int x2 = rightSkyline[j].first;

        int x;

        if (x1 < x2) {

            x = x1;

            h1 = leftSkyline[i].second;

            i++;

        } else if (x1 > x2) {

            x = x2;

            h2 = rightSkyline[j].second;

            j++;

        } else {

            x = x1;

            h1 = leftSkyline[i].second;

            h2 = rightSkyline[j].second;

            i++;

            j++;

        }

        int maxHeight = max(h1, h2);

        if (currentHeight != maxHeight) {

            mergedSkyline.push\_back({x, maxHeight});

            currentHeight = maxHeight;

        }

    }

    while (i < leftSkyline.size()) {

        mergedSkyline.push\_back(leftSkyline[i++]);

    }

    while (j < rightSkyline.size()) {

        mergedSkyline.push\_back(rightSkyline[j++]);

    }

    return mergedSkyline;

}

vector<Point> skyline(vector<vector<int>>& buildings, int left, int right) {

    if (left == right) {

        return {{buildings[left][0], buildings[left][1]}, {buildings[left][2], 0}};

    }

    int mid = (left + right) / 2;

    vector<Point> leftSkyline = skyline(buildings, left, mid);

    vector<Point> rightSkyline = skyline(buildings, mid + 1, right);

    return mergeSkylines(leftSkyline, rightSkyline);

}

int main() {

    int n;

    cout << "Enter the number of buildings: ";

    cin >> n;

    vector<vector<int>> buildings(n, vector<int>(3));

    cout << "Enter the buildings in format (left ht right): " << endl;

    for (int i = 0; i < n; i++) {

        cin >> buildings[i][0] >> buildings[i][1] >> buildings[i][2];

    }

    vector<Point> result = skyline(buildings, 0, n - 1);

    cout << "The skyline is: ";

    for (auto p : result) {

        cout << "(" << p.first << ", " << p.second << ") ";

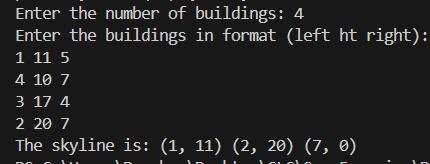
    }

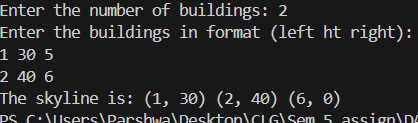
    cout << endl;

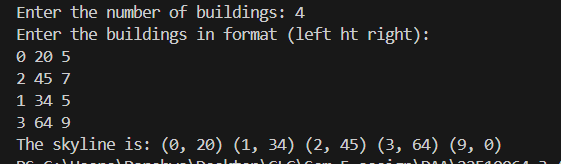
    return 0;

}

OUTPUT:





  
Complexity Analysis:

* Best Time Complexity: O(n log n)
* Worst Time Complexity: O(n log n)
* Average Time Complexity: O(n log n)
* Space Complexity: O(n)