Batch - T5
Assignment No. - 3
Title - Divide and Conquer Strategy
Student Name - Sharaneshwar Bharat Punjal
Student PRN – 23520011

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

Algorithm:

1. Initialize two pointers: low at the start of the array and high at the end.
2. While low is less than or equal to high:
   Calculate the middle index mid.
   Check if mid is the peak element
3. If the element at mid is greater than both its neighbours, then arr[mid] is the maximum element.
4. If the array is increasing at mid, then move the low pointer to mid + 1.
5. Otherwise, move the high pointer to mid - 1.
6. The loop will eventually converge on the maximum element.

Code:

```java
import java.util.Scanner;

public class Q1 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++)
            arr[i] = sc.nextInt();
        sc.close();

        System.out.println(findMaximum(arr, n));
    }

    public static int findMaximum(int[] arr, int n) {
        int left = 0;
        int right = n - 1;

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

            if (left == right)
                return arr[left];

            if (right == left + 1)
                return Math.max(arr[left], arr[right]);

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

```
                return arr[mid];

            if (arr[mid] > arr[mid - 1] && arr[mid] < arr[mid + 1])
                left = mid + 1;
            else
                right = mid - 1;
        }

        return -1;
    }
}
```

Test Case 1:

```
10
1 3 5 7 9 12 15 10 6 2
```

Output:

```
15
```

Test Case 2:

```
15
5 10 15 20 25 30 35 30 25 20 15 10 5 3 1
```

Output:

```
35
```

Test Case 3:

```
20
2 5 10 15 20 30 40 50 45 35 25 15 10 5 2 1 0 -1 -2 -3
```

Output:

```
50
```

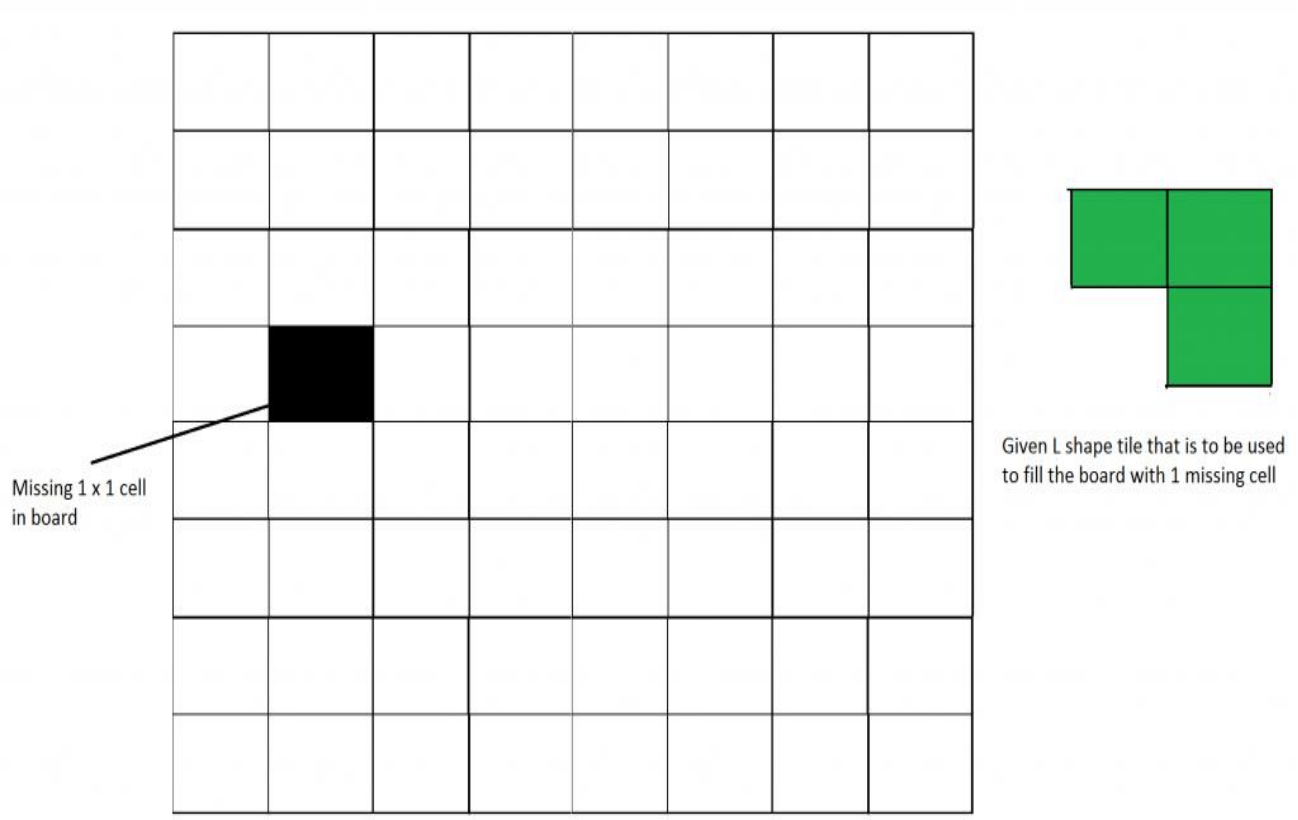
Test Case 4:

```
25
3 6 12 18 27 35 40 38 30 20 15 10 5 2 1 0 -2 -5 -10 -15 -20 -25 -30 -35 -40
```

Output:

```
40
```

2) 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.



Algorithm:

1. Start with an empty board of size n x n initialized with all zeros.
2. Mark the pre-filled cell (given as input) with -1.
3. Define a recursive function tile(size, r, c, board) to fill the board.
4. If the current sub-board size is 2 x 2, fill it with a unique tile number, ensuring that the missing tile is respected.
5. Find the coordinates of the pre-filled cell within the current sub-board.
6. Based on the location of this pre-filled cell, place an L-shaped tromino in the other three quadrants.
7. Recursively apply the tile function to the four quadrants.
8. The base case is when the size of the sub-board is 2 x 2. In this case, fill the empty cells with the current tile number.
9. After the board is fully tiled, output the board.

Code:

```
import java.util.Scanner;

public class Q2 {

    static int tileNum = 0;
    static char[][] board;

    private static void place(int x1, int y1, int x2, int y2, int x3, int y3) {
```

```
        tileNum++;
        board[x1][y1] = '*';
        board[x2][y2] = '*';
        board[x3][y3] = '*';
    }

    private static void tile(int size, int r, int c) {
        if (size == 2) {
            tileNum++;
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    if (board[r + i][c + j] == ' ') {
                        board[r + i][c + j] = '*';
                    }
                }
            }
            return;
        }

        int mr = -1, mc = -1;
        for (int i = r; i < r + size; i++) {
            for (int j = c; j < c + size; j++) {
                if (board[i][j] != ' ') {
                    mr = i;
                    mc = j;
                    break;
                }
            }
            if (mr != -1) break;
        }

        if (mr < r + size / 2 && mc < c + size / 2) {
            place(r + size / 2, c + size / 2 - 1, r + size / 2, c + size / 2, r + size
/ 2 - 1, c + size / 2);
        } else if (mr >= r + size / 2 && mc < c + size / 2) {
            place(r + size / 2 - 1, c + size / 2, r + size / 2, c + size / 2, r + size
/ 2 - 1, c + size / 2 - 1);
        } else if (mr < r + size / 2 && mc >= c + size / 2) {
            place(r + size / 2, c + size / 2 - 1, r + size / 2, c + size / 2, r + size
/ 2 - 1, c + size / 2 - 1);
        } else if (mr >= r + size / 2 && mc >= c + size / 2) {
            place(r + size / 2 - 1, c + size / 2, r + size / 2, c + size / 2 - 1, r +
size / 2 - 1, c + size / 2 - 1);
        }

        tile(size / 2, r, c + size / 2);
        tile(size / 2, r, c);
        tile(size / 2, r + size / 2, c);
        tile(size / 2, r + size / 2, c + size / 2);
    }
```

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int mr = sc.nextInt();
        int mc = sc.nextInt();
        sc.close();

        board = new char[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                board[i][j] = ' ';
            }
        }

        board[mr][mc] = '+';

        tile(n, 0, 0);

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                System.out.print(board[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Test Case 1:

```
4
1 2
```

Output:

```
* * * *
* * + *
* * * *
* * * *
```

Test Case 2:

```
8
2 2
```

Output:

```
* * * * * * * *
* * * * * * * *
* * + * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
```

Test Case 3:

16

4 3

Output:

```
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * + * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
```

Test Case 4:

2

1 1

Output:

```
* *
* +
```

3) 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.

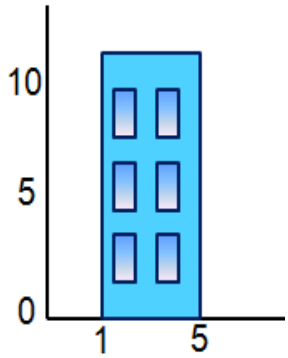
All buildings share common bottom and every building is represented by triplet (left, ht, right)

'left': is x coordinated of left side (or wall).

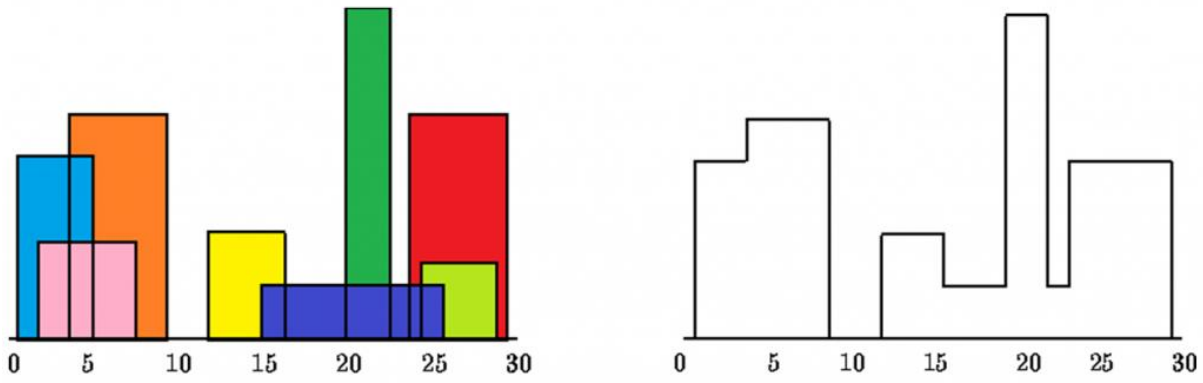
'right': is x coordinate of right side

'ht': is height of building.

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



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.
Required Time Complexity - O (n log n)



Algorithm:

1. The input is a list of buildings, where each building is represented by a triplet [Li, Ri, Hi], where Li is the left coordinate, Ri is the right coordinate, and Hi is the height.
2. For each building, generate two points:
   - A start point (Li, -Hi) which represents the beginning of the building with negative height to distinguish start from end.
   - An end point (Ri, Hi) which represents the end of the building with positive height.
3. Sort the points. If two points have the same x-coordinate, the point with the smaller height comes first. If two points have the same height, the start point (negative height) comes before the end point (positive height).
4. Use a max-heap to keep track of the current heights of the buildings as you iterate through the points. For each point:
   - If it's a start point, add its height to the heap.
   - If it's an end point, remove its height from the heap.
   - Compare the current maximum height (top of the heap) with the previous maximum height. If they differ, this means the skyline changes at this point, so record this point in the result.

Code:

```
import java.util.*;

public class Q3 {

    static class Point {
        int x, height;

        Point(int x, int height) {
            this.x = x;
            this.height = height;
        }
    }
```

```
    public static List<List<Integer>> getSkyline(int[][] buildings) {
        List<List<Integer>> result = new ArrayList<>();
        TreeSet<Integer> heights = new TreeSet<>(Collections.reverseOrder());
        List<Point> points = new ArrayList<>();

        for (int[] building : buildings) {
            points.add(new Point(building[0], -building[2]));
            points.add(new Point(building[1], building[2]));
        }

        points.sort((a, b) -> {
            if (a.x != b.x)
                return Integer.compare(a.x, b.x);
            return Integer.compare(b.height, a.height);
        });

        int ongoingHeight = 0;

        for (Point point : points) {
            int currentPoint = point.x;
            int heightAtCurrentPoint = point.height;

            if (heightAtCurrentPoint < 0) {
                heights.add(-heightAtCurrentPoint);
            } else {
                heights.remove(heightAtCurrentPoint);
            }

            int currentHeight = heights.isEmpty() ? 0 : heights.first();

            if (ongoingHeight != currentHeight) {
                ongoingHeight = currentHeight;
                result.add(Arrays.asList(currentPoint, ongoingHeight));
            }
        }

        return result;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[][] buildings = new int[n][3];

        for (int i = 0; i < n; i++) {
            buildings[i][0] = sc.nextInt();
            buildings[i][1] = sc.nextInt();
            buildings[i][2] = sc.nextInt();
        }
```

```
        List<List<Integer>> result = getSkyline(buildings);
        for (List<Integer> r : result) {
            System.out.println(r.get(0) + " " + r.get(1));
        }

        sc.close();
    }
}
```

Test Case 1:

```
3
2 9 10
3 7 15
5 12 12
```

Output:

```
2 10
3 15
7 12
12 0
```

Test Case 2:

```
2
1 2 2
3 4 3
```

Output:

```
1 2
2 0
3 3
4 0
```

Test Case 3:

```
4
1 4 4
2 5 6
4 6 2
5 7 3
```

Output:

```
1 4
2 6
5 2
5 3
7 0
```

Test Case 4:

```
5
1 3 4
2 5 6
4 7 3
6 8 5
9 10 2
```

Output:

```
1 4
2 6
5 3
6 5
8 0
9 2
10 0
```