# Experiment 5

**Student Name: Ankit Kumar Yadav** **UID: 22BCS11046**
**Branch: BE-CSE** **Section/Group: 638_A**
**Semester: 6th** **Date of Performance:18/02/25**
**Subject Name: Advance Programming Lab -2** **Subject Code: 22CSP-351**

## 1. problem:-Beautiful Array

## 2. Aim:-

An array nums of length n is beautiful if:

nums is a permutation of the integers in the range [1, n].

For every 0 <= i < j < n, there is no index k with i < k < j where 2 * nums[k] == nums[i] + nums[j].

Given the integer n, return any beautiful array nums of length n. There will be at least one valid answer for the given n.

Example 1:

Input: n = 4

Output: [2,1,4,3]

Example 2:

Input: n = 5

Output: [3,1,2,5,4]

Link:- Beautiful Array - LeetCode

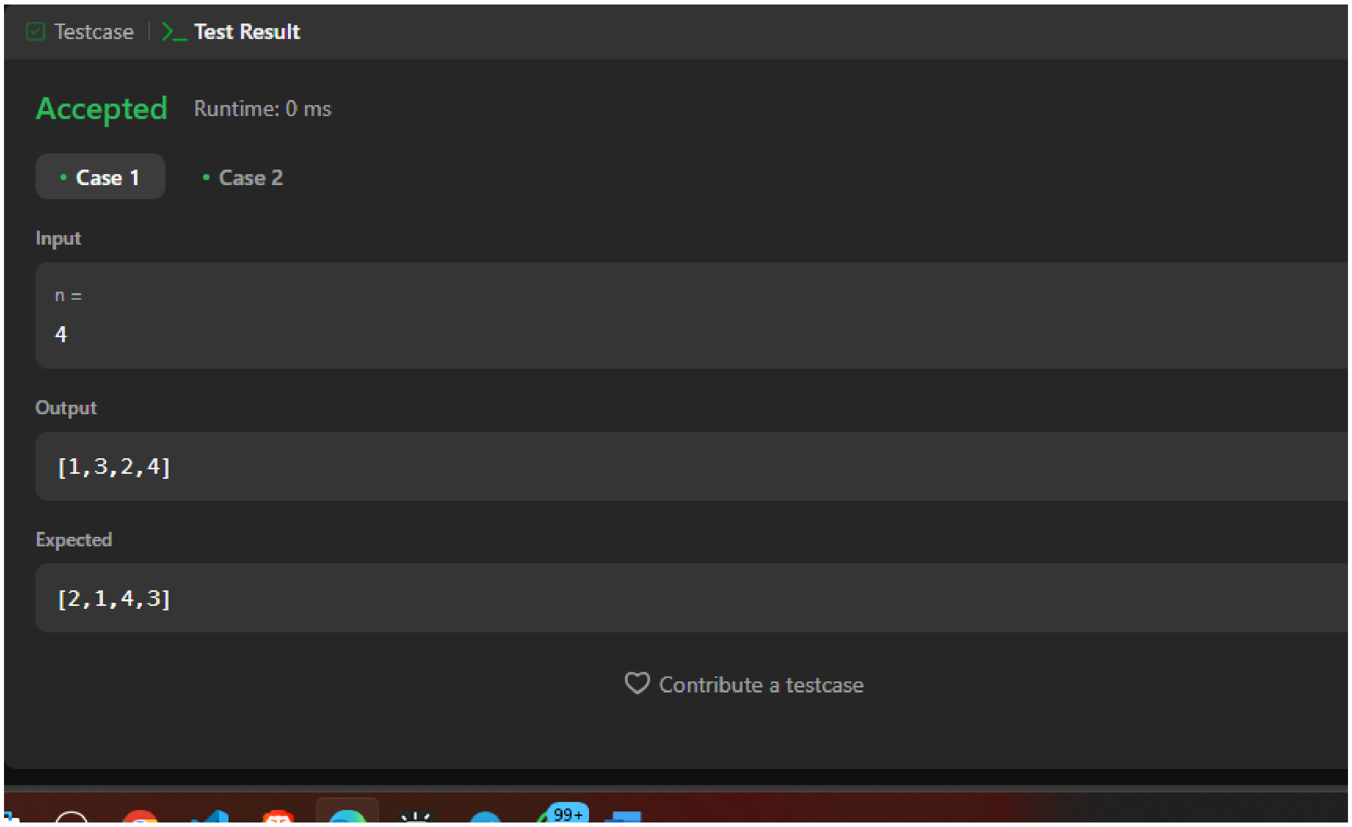
## 3. Objective:

To generate a beautiful array of length **n**, we ensure it is a permutation of integers from **[1, n]**, avoiding any triplet **(i, j, k)** that satisfies **2 * nums[k] == nums[i] + nums[j]**. A divide-and-conquer approach helps construct the array recursively by placing odd numbers first and even numbers second, maintaining the required constraints. This guarantees that no middle element forms an arithmetic sequence, preserving the beautiful array property. The method runs in **O(n log n)** time complexity, efficiently producing at least one valid solution for **n ≤ 1000**.

## 4. Implementation/Code:

```
class Solution {
    public int[] beautifulArray(int n) {
        List<Integer> result = new ArrayList<>();
        result.add(1);

        while (result.size() < n) {
            List<Integer> temp = new ArrayList<>();
            for (int num : result) {
                if (2 * num - 1 <= n) {
                    temp.add(2 * num - 1);
                }
            }

            for (int num : result) {
                if (2 * num <= n) {
                    temp.add(2 * num);
                }
            }

            result = temp;
        }

        int[] ans = new int[n];
        for (int i = 0; i < n; i++) {
            ans[i] = result.get(i);
        }

        return ans;
    }

}
```

**Output:-**



Accepted Runtime: 0 ms

Input

```
n =
4
```

Output

```
[1,3,2,4]
```

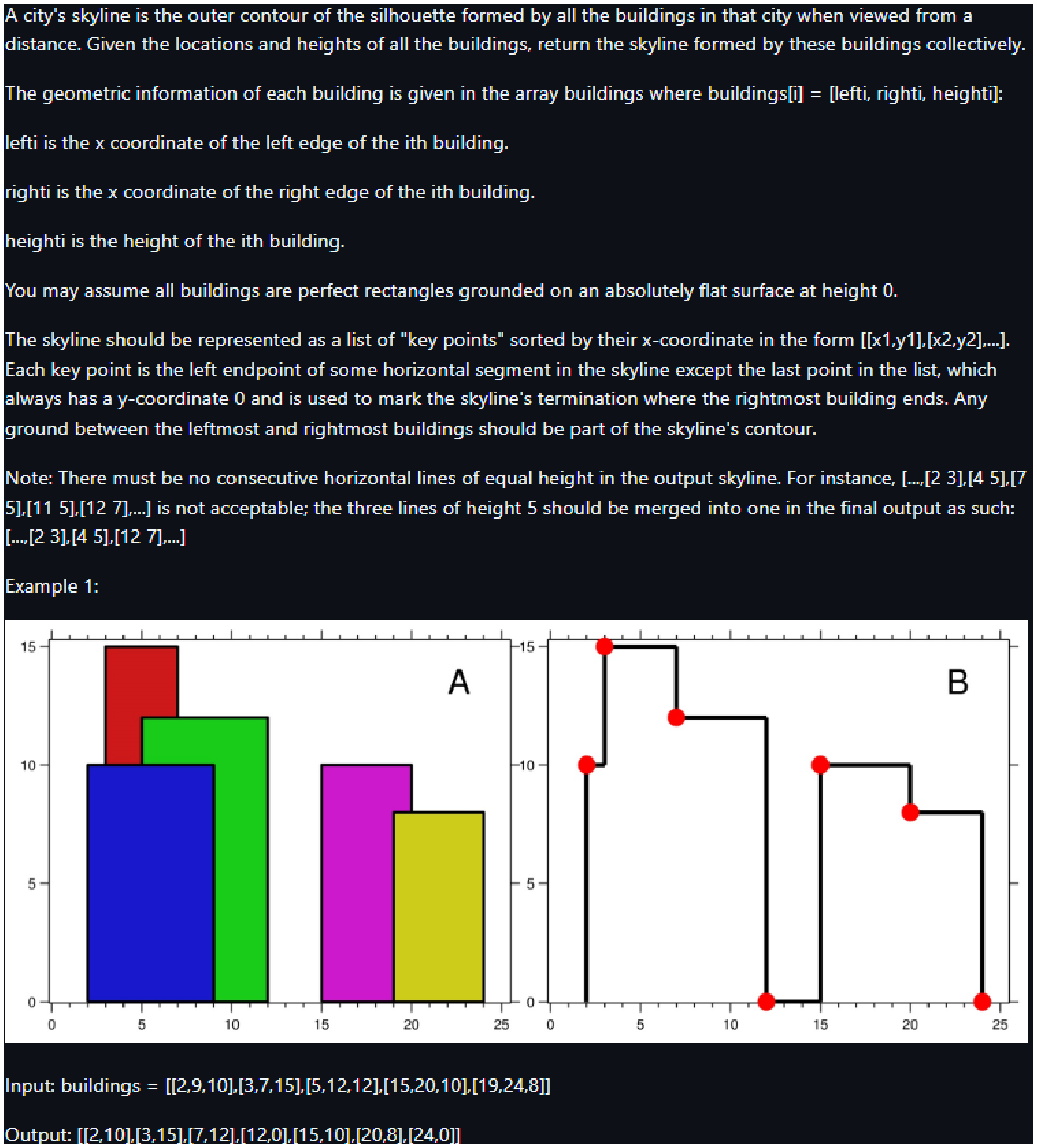
Expected

```
[2,1,4,3]
```

## 5. Learning outcomes:

1. **Understanding Beautiful Arrays**: Learn how to construct a permutation of [1, n] that avoids forming arithmetic sequences.
2. **Divide-and-Conquer Approach**: Use recursion to build the array by separating odd and even numbers.
3. **Algorithm Efficiency**: Implement an **O(n log n)** solution that scales well for large n.
4. **Practical Implementation**: Apply sorting and sequence generation techniques in C++/Java/Python..

1. **Problem: The Skyline Problem**
2. **Aim:**



A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Given the locations and heights of all the buildings, return the skyline formed by these buildings collectively.

The geometric information of each building is given in the array buildings where buildings[i] = [lefti, righti, heighti]:

lefti is the x coordinate of the left edge of the ith building.

righti is the x coordinate of the right edge of the ith building.

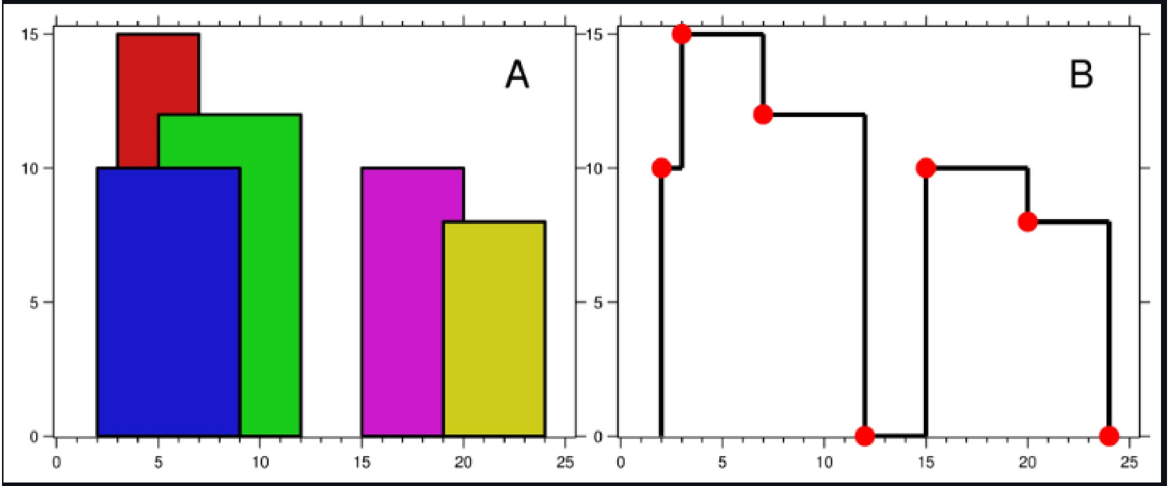
heighti is the height of the ith building.

You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

The skyline should be represented as a list of "key points" sorted by their x-coordinate in the form [[x1,y1],[x2,y2],...]. Each key point is the left endpoint of some horizontal segment in the skyline except the last point in the list, which always has a y-coordinate 0 and is used to mark the skyline's termination where the rightmost building ends. Any ground between the leftmost and rightmost buildings should be part of the skyline's contour.

Note: There must be no consecutive horizontal lines of equal height in the output skyline. For instance, [...,[2 3],[4 5],[7 5],[11 5],[12 7],...] is not acceptable; the three lines of height 5 should be merged into one in the final output as such: [...,[2 3],[4 5],[12 7],...]

Example 1:



Input: buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]

Output: [[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]

Link:- The Skyline Problem - LeetCode

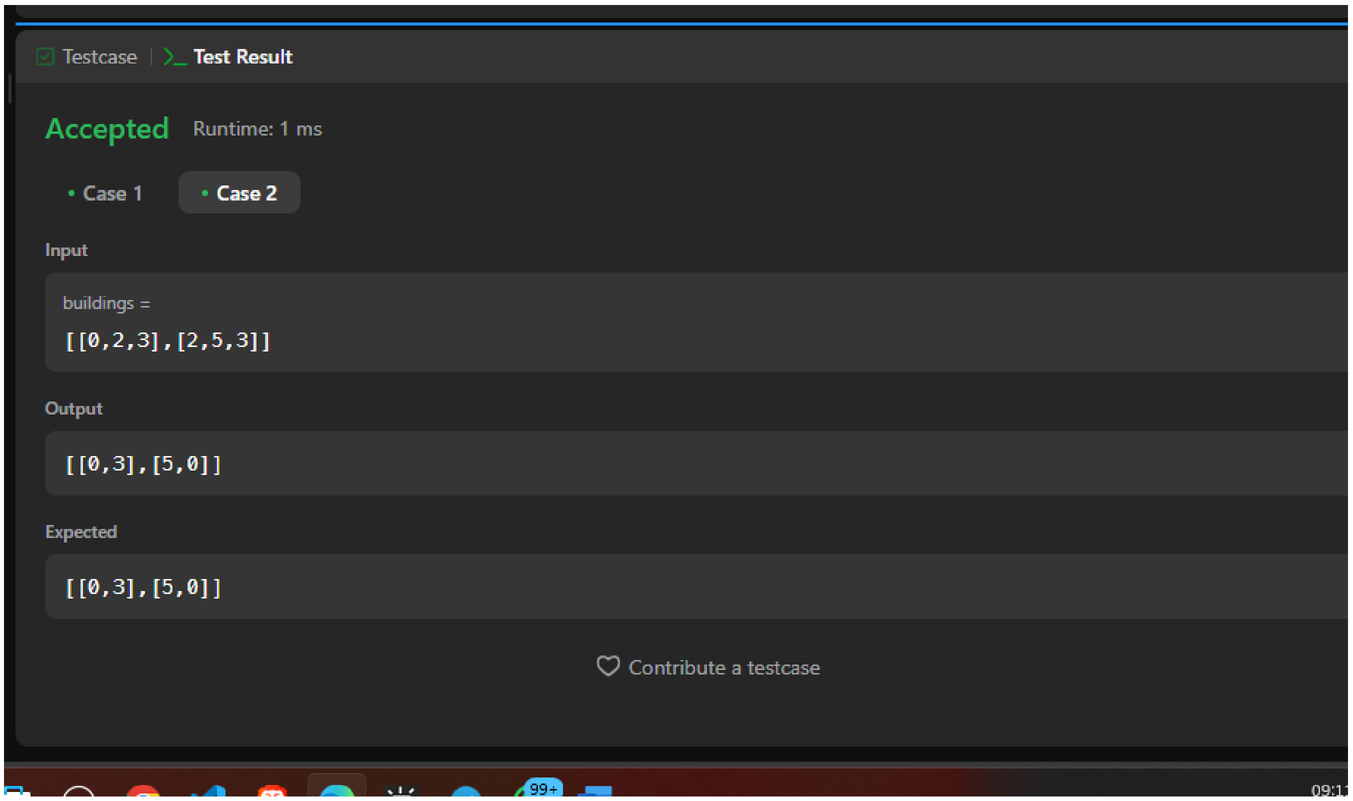
3. **Objective:**

   The objective is to efficiently identify the kth largest element within a given integer array. The target element is determined by its position in the sorted order of the array, not by its uniqueness. The challenge lies in achieving this without resorting to a full sorting of the array, implying the need for a more optimized approach. Efficiency is key, especially considering the potential size of the input array. The goal is to return the value of the kth largest element.

4. **Implementation/Code:**

```
class Solution {
    public List<List<Integer>> getSkyline(int[][] buildings) {
        List<int[]> events = new ArrayList<>();

        for (int[] b : buildings) {
            events.add(new int[]{b[0], -b[2]});
            events.add(new int[]{b[1], b[2]});
        }


        Collections.sort(events, (a, b) -> {
            if (a[0] != b[0]) return a[0] - b[0];
            return a[1] - b[1];
        });


        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
        maxHeap.add(0);
        int prevMax = 0;

        List<List<Integer>> result = new ArrayList<>();

        for (int[] event : events) {
            int x = event[0], h = event[1];

            if (h < 0) {
                maxHeap.add(-h);
            } else {
                maxHeap.remove(h);
            }

            int currMax = maxHeap.peek();
            if (currMax != prevMax) {
                result.add(Arrays.asList(x, currMax));
                prevMax = currMax;
            }
        }

        return result;

    }
}
```

**Output:-**


Testcase | Test Result

Accepted Runtime: 1 ms

Case 1 Case 2

Input

```
buildings =
[[0,2,3],[2,5,3]]
```

Output

```
[[0,3],[5,0]]
```

Expected

```
[[0,3],[5,0]]
```

Contribute a testcase

## 6. Learning outcomes:

1. **Understanding Sweep Line Algorithm**: Learn how to process events efficiently using a **sorted event-based approach**.
2. **Using a Max Heap for Dynamic Height Tracking**: Gain insight into **priority queues** to track active building heights dynamically.
3. **Efficient Sorting and Processing**: Implement sorting techniques to ensure buildings are processed in the correct order, maintaining **O(n log n) complexity**.
4. **Handling Edge Cases in Computational Geometry**: Learn how to manage cases like overlapping buildings and merging consecutive horizontal lines.