# Question

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]:

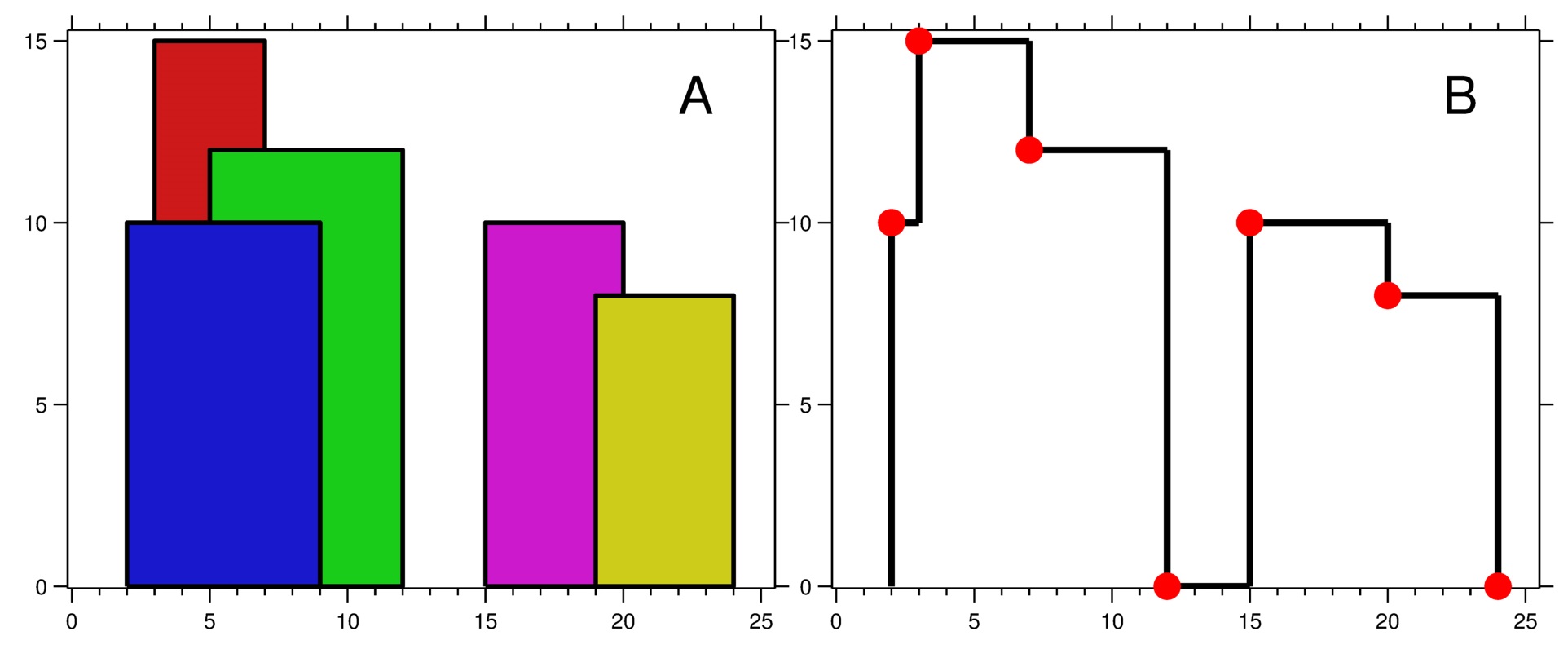
* lefti is the x coordinate of the left edge of the ith building.
* righti is the x coordinate of the right edge of the ith building.
* heighti is the height of the ith building.

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]]

**Explanation:**

Figure A shows the buildings of the input.

Figure B shows the skyline formed by those buildings. The red points in figure B represent the key points in the output list.

**Example 2:**

**Input:** buildings = [[0,2,3],[2,5,3]]

**Output:** [[0,3],[5,0]]

**Constraints:**

* 1 <= buildings.length <= 104
* 0 <= lefti < righti <= 231 - 1
* 1 <= heighti <= 231 - 1
* buildings is sorted by lefti in non-decreasing order.

# Solution

#### **Approach 1: Divide and Conquer**

**Solution Template**

The problem is a classical example of divide and conquer approach, and typically implemented exactly the same way as merge sort algorithm.

Let's follow here a solution template for divide and conquer problems :

* Define the base case(s).
* Split the problem into subproblems and solve them recursively.
* Merge the subproblems solutions into the problem solution.

**Algorithm**

getSkyline for n buildings :

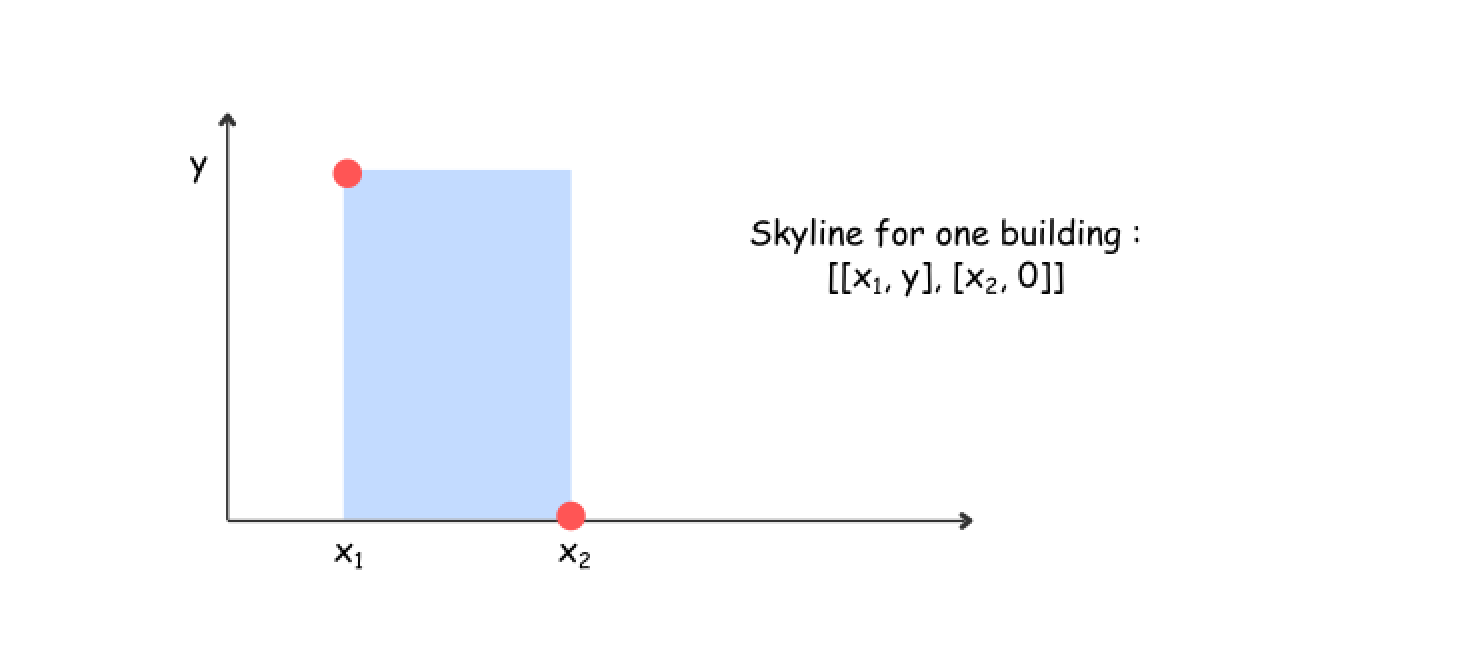
* If n == 0 : return an empty list.
* If n == 1 : return the skyline for one building (it's straightforward).
* leftSkyline = getSkyline for the first n/2 buildings.
* rightSkyline = getSkyline for the last n/2 buildings.
* Merge leftSkyline and rightSkyline.

Now let's discuss each step in more details.

**Base Cases**

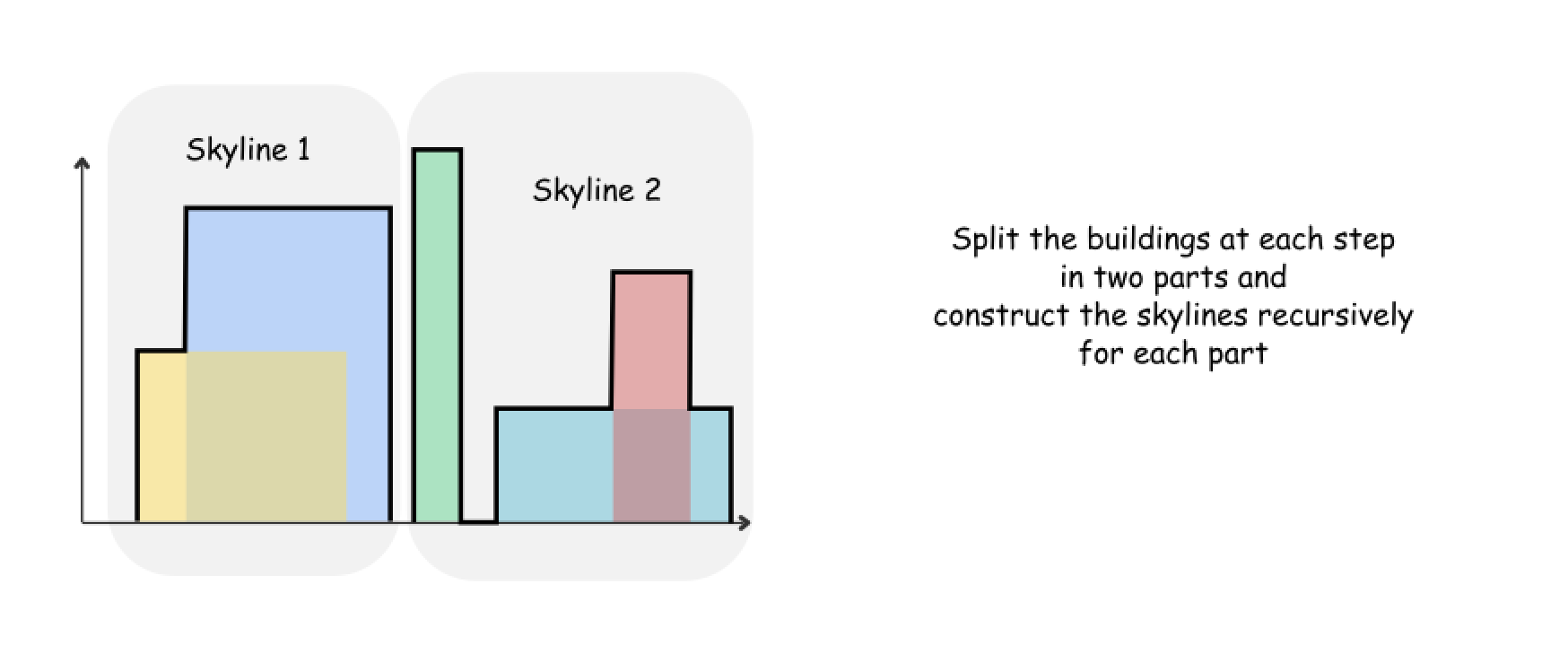
The first base case is an empty buildings list. Then the skyline is an empty list, too.

The second base case is the only one building in the list, when the skyline construction is quite straightforward.



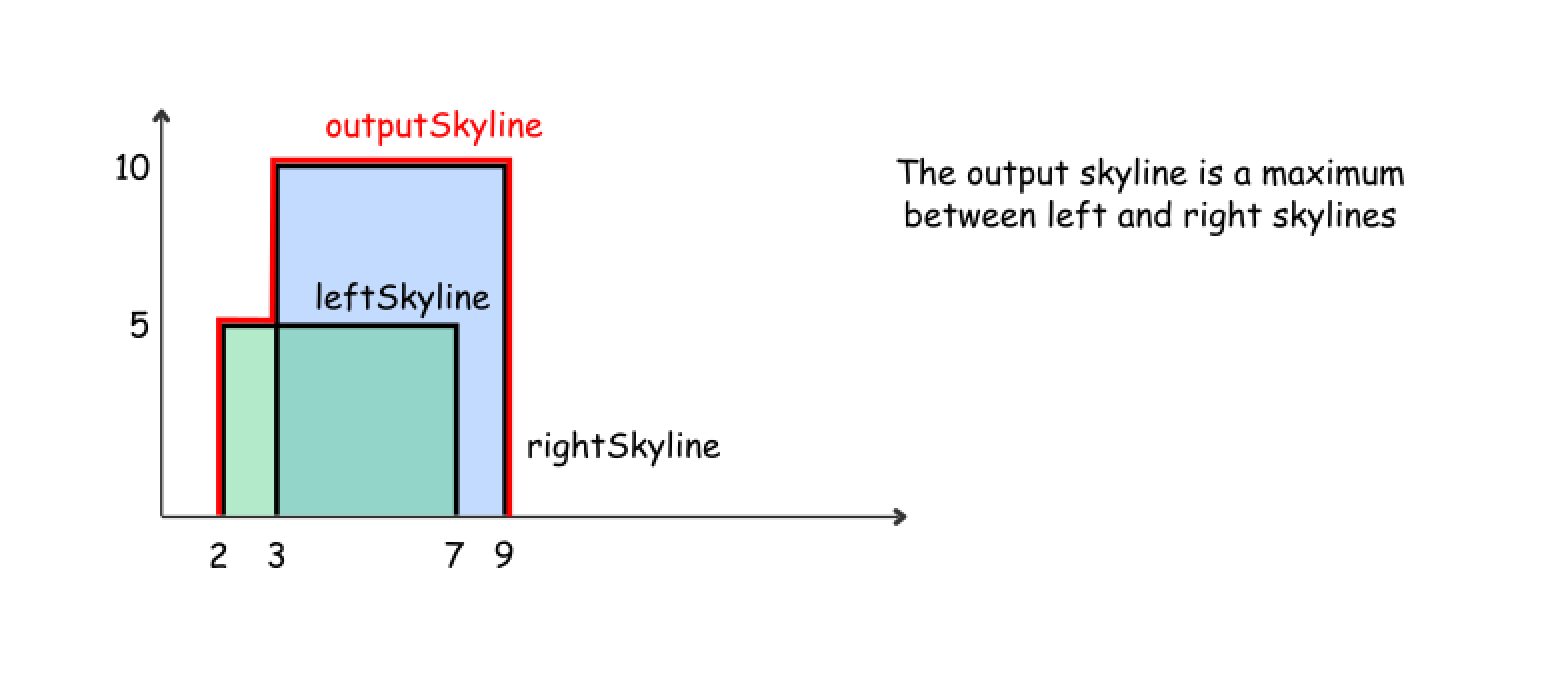
**How to split the problem**

The idea is the same as for merge sort : at each step split the list exactly in two parts : from 0 to n/2 and from n/2 to n, and then construct the skylines recursively for each part.



**How to merge two skylines**

The algorithm for merge function is quite straightforward and based on the same merge sort logic : the height of an output skyline is always a maximum between the left and right skylines.

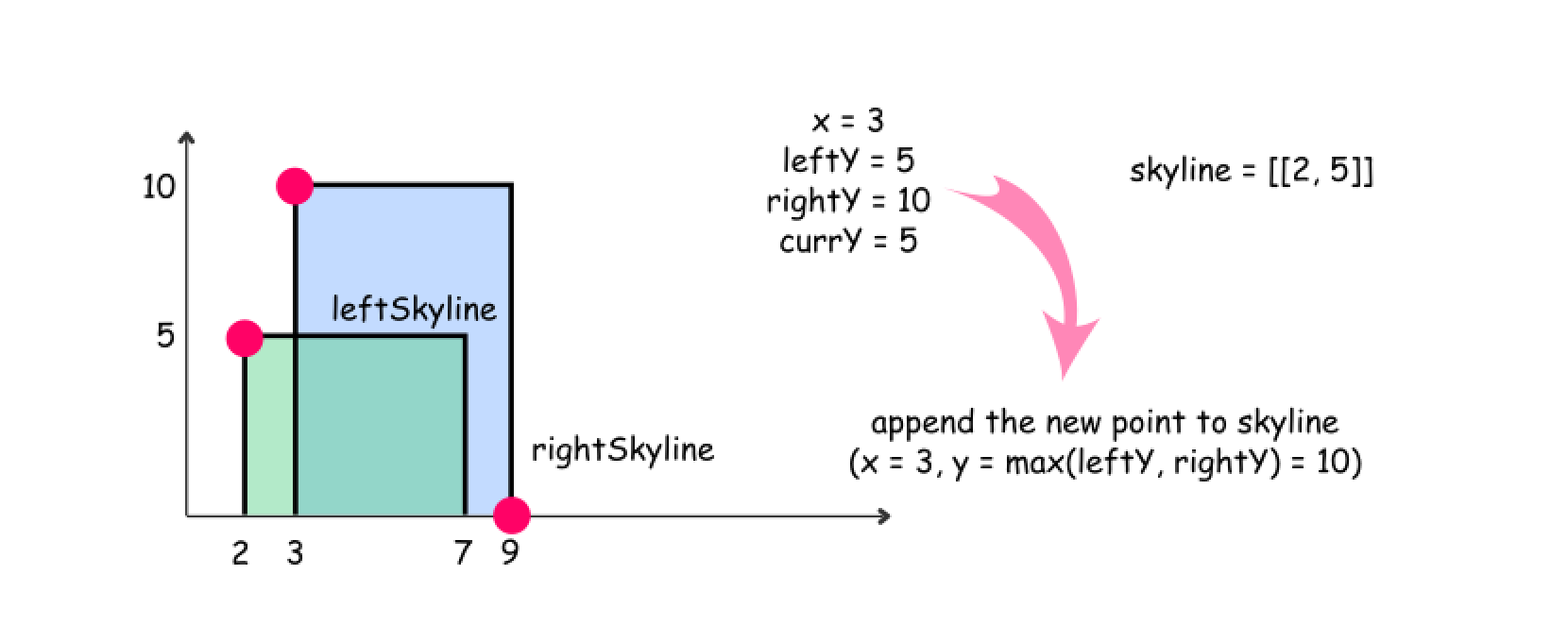


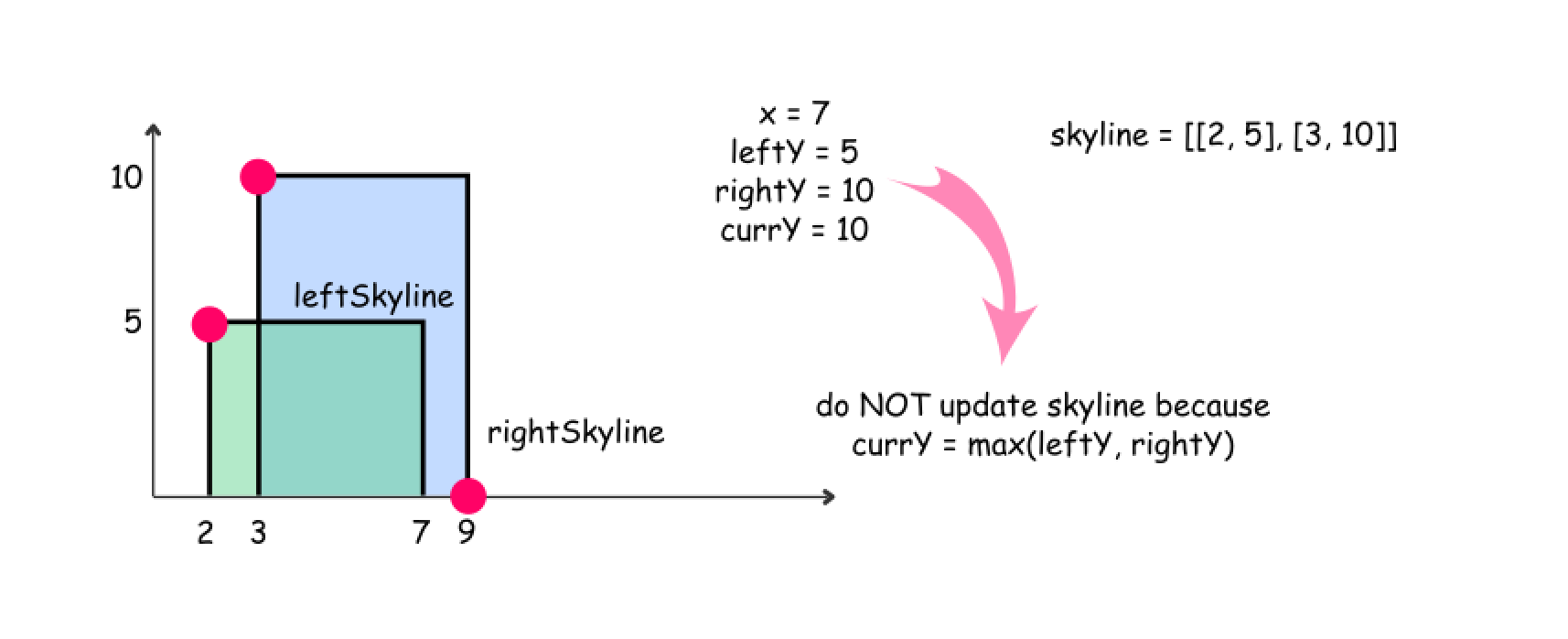
Let's use here two pointers pR and pL to track the current element index in both skylines, and three integers leftY, rightY, and currY to track the current height for the left skyline, right skyline and the merged skyline.

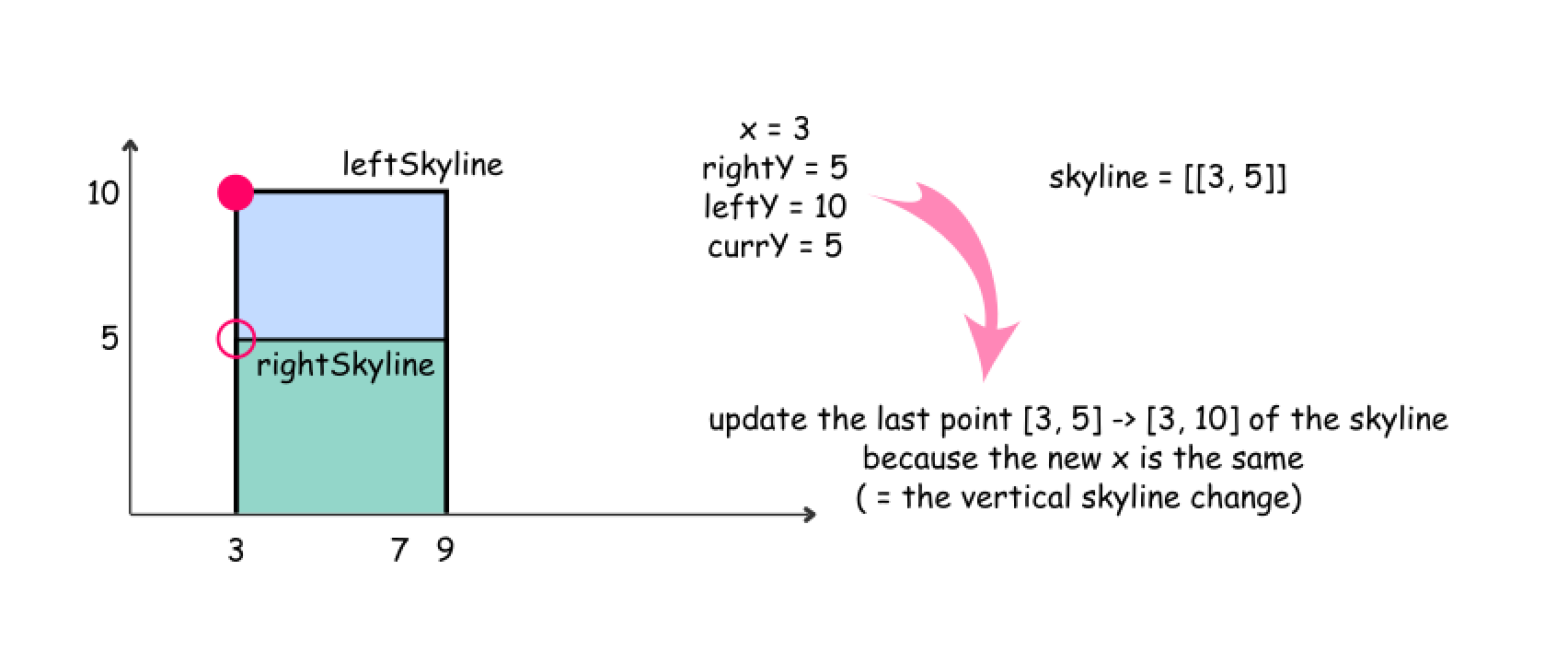
mergeSkylines (left, right) :

* currY = leftY = rightY = 0
* While we're in the region where both skylines are present (pR < nR and pL < nL) :
  + Pick up the element with the smallest x coordinate. If it's an element from the left skyline, move pL and update leftY. If it's an element from the right skyline, move pR and update rightY.
  + Compute the largest height at the current point : maxY = max(leftY, rightY).
  + Update an output skyline by (x, maxY) point, if maxY is not equal to currY.
* While there are still elements in the left skyline (pL < nL), process them following the same logic as above.
* While there are still elements in the right skyline (pR < nR), process them following the same logic as above.
* Return output skyline.

Here are three usecases to illustrate the merge algorithm execution







|  |
| --- |
| **class Solution {**  **/\*\***  **\* Divide-and-conquer algorithm to solve skyline problem,**  **\* which is similar with the merge sort algorithm.**  **\*/**  **public List<List<Integer>> getSkyline(int[][] buildings) {**  **int n = buildings.length;**  **List<List<Integer>> output = new ArrayList<List<Integer>>();**  **// The base cases**  **if (n == 0) return output;**  **if (n == 1) {**  **int xStart = buildings[0][0];**  **int xEnd = buildings[0][1];**  **int y = buildings[0][2];**  **output.add(new ArrayList<Integer>() {{add(xStart); add(y); }});**  **output.add(new ArrayList<Integer>() {{add(xEnd); add(0); }});**  **// output.add(new int[]{xStart, y});**  **// output.add(new int[]{xEnd, 0});**  **return output;**  **}**  **// If there is more than one building,**  **// recursively divide the input into two subproblems.**  **List<List<Integer>> leftSkyline, rightSkyline;**  **leftSkyline = getSkyline(Arrays.copyOfRange(buildings, 0, n / 2));**  **rightSkyline = getSkyline(Arrays.copyOfRange(buildings, n / 2, n));**  **// Merge the results of subproblem together.**  **return mergeSkylines(leftSkyline, rightSkyline);**  **}**  **/\*\***  **\* Merge two skylines together.**  **\*/**  **public List<List<Integer>> mergeSkylines(List<List<Integer>> left, List<List<Integer>> right) {**  **int nL = left.size(), nR = right.size();**  **int pL = 0, pR = 0;**  **int currY = 0, leftY = 0, rightY = 0;**  **int x, maxY;**  **List<List<Integer>> output = new ArrayList<List<Integer>>();**  **// while we're in the region where both skylines are present**  **while ((pL < nL) && (pR < nR)) {**  **List<Integer> pointL = left.get(pL);**  **List<Integer> pointR = right.get(pR);**  **// pick up the smallest x**  **if (pointL.get(0) < pointR.get(0)) {**  **x = pointL.get(0);**  **leftY = pointL.get(1);**  **pL++;**  **}**  **else {**  **x = pointR.get(0);**  **rightY = pointR.get(1);**  **pR++;**  **}**  **// max height (i.e. y) between both skylines**  **maxY = Math.max(leftY, rightY);**  **// update output if there is a skyline change**  **if (currY != maxY) {**  **updateOutput(output, x, maxY);**  **currY = maxY;**  **}**  **}**  **// there is only left skyline**  **appendSkyline(output, left, pL, nL, currY);**  **// there is only right skyline**  **appendSkyline(output, right, pR, nR, currY);**  **return output;**  **}**  **/\*\***  **\* Update the final output with the new element.**  **\*/**  **public void updateOutput(List<List<Integer>> output, int x, int y) {**  **// if skyline change is not vertical -**  **// add the new point**  **if (output.isEmpty() || output.get(output.size() - 1).get(0) != x)**  **output.add(new ArrayList<Integer>() {{add(x); add(y); }});**  **// if skyline change is vertical -**  **// update the last point**  **else {**  **output.get(output.size() - 1).set(1, y);**  **}**  **}**  **/\*\***  **\* Append the rest of the skyline elements with indice (p, n)**  **\* to the final output.**  **\*/**  **public void appendSkyline(List<List<Integer>> output, List<List<Integer>> skyline,**  **int p, int n, int currY) {**  **while (p < n) {**  **List<Integer> point = skyline.get(p);**  **int x = point.get(0);**  **int y = point.get(1);**  **p++;**  **// update output**  **// if there is a skyline change**  **if (currY != y) {**  **updateOutput(output, x, y);**  **currY = y;**  **}**  **}**  **}**  **}** |

**Implementation**

