# PROBLEM STATEMENT

The skyline problem is a classic problem in computer science and mathematics, which involves finding the outer contour of a city's skyline based on the heights and locations of all the buildings in that city. The problem has many applications, including in urban planning, architecture, and computer graphics. This problem can be solved using various approaches, including brute force approach, divide and conquer approach, and priority queue approach. In this paper, we will discuss each of these approaches and compare their performance and efficiency.

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. The task is to find the skyline formed by these buildings collectively, given the locations and heights of all the buildings. Each building is represented by an array of three values: lefti, righti, and heighti, where 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, and heighti is the height of the ith building. All buildings are perfect rectangles grounded on an absolutely flat surface at height 0.
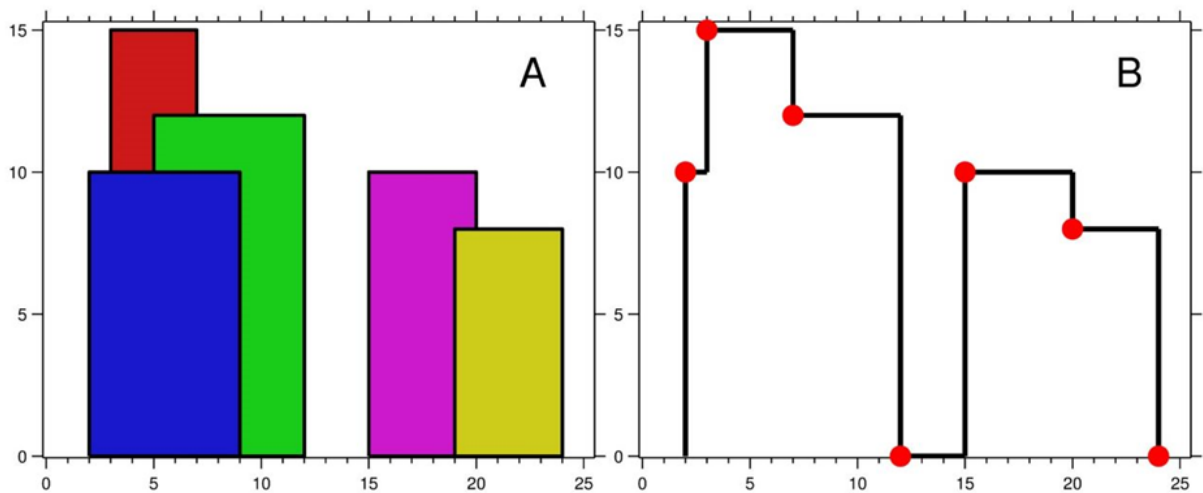
 The skyline should be represented as a list of "key points" sorted by their x-coordinate in the form [[x1,y1],[x2,y2],...]. Each key point is the left endpoint of some horizontal segment in the skyline except the last point in the list, which always has a y-coordinate 0 and is used to mark the skyline's termination where the rightmost building ends. Any ground between the leftmost and rightmost buildings should be part of the skyline's contour.

One important condition is that 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],...].

Approaches used are:

·   Brute force approach: In the brute force approach, we consider every possible point on the skyline and determine its height by comparing it with the heights of all the buildings at that point. This approach has a time complexity of $O(n^2)$ and is not efficient for large inputs.

·   Divide and conquer approach: In the divide and conquer approach, we divide the set of buildings into two halves and recursively compute the skyline of each half. We then merge the two skylines to obtain the final skyline. This approach has a time complexity of O(nlogn) and is more efficient than the brute force approach.

·   Priority queue approach: In the priority queue approach, we maintain a priority queue of all the building endpoints sorted by their x-coordinates. We then process the endpoints one by one and maintain a heap of the heights of the active buildings. Whenever the maximum height changes, we add the new point to the skyline. This approach has a time complexity of O(nlogn) and is the most efficient among the three approaches.

In conclusion, the skyline problem is an important problem with many applications. We have discussed three different approaches to solve the problem: brute force approach, divide and conquer approach, and priority queue approach. The priority queue approach is the most efficient among these approaches. However, the choice of approach depends on the specific problem and the size of the input.

BRUTE FORCE APPROACH

The brute force approach to solve the skyline problem involves a straightforward algorithm that considers every x-coordinate and finds the maximum height of the buildings at that point. To implement this approach, the algorithm iterates through each building in the input array and considers each x-coordinate within the building's left and right edges. For each x-coordinate, the algorithm checks the height of the building and updates the maximum height seen so far. Although the brute force approach is not the most efficient, it provides a simple way to solve the skyline problem, especially for small inputs. Additionally, it provides a benchmark for evaluating the efficiency of other approaches.

C++ Code:

```cpp
vector<pair<int, int>> getSkyline(vector<vector<int>>& buildings) {

    vector<pair<int, int>> result;

    int n = buildings.size();

    for (int i = 0; i < n; i++) {

        int x1 = buildings[i][0];

        int x2 = buildings[i][1];

        int h = buildings[i][2];

        for (int j = x1; j <= x2; j++) {

            int max_height = 0;

            for (int k = 0; k < n; k++) {

                    if (buildings[k][0] <= j && buildings[k][1] >= j) {

                    max_height = max(max_height, buildings[k][2]);

                    }

            }

            result.push_back(make_pair(j, max_height));

        }

    }

    return result;

}
```
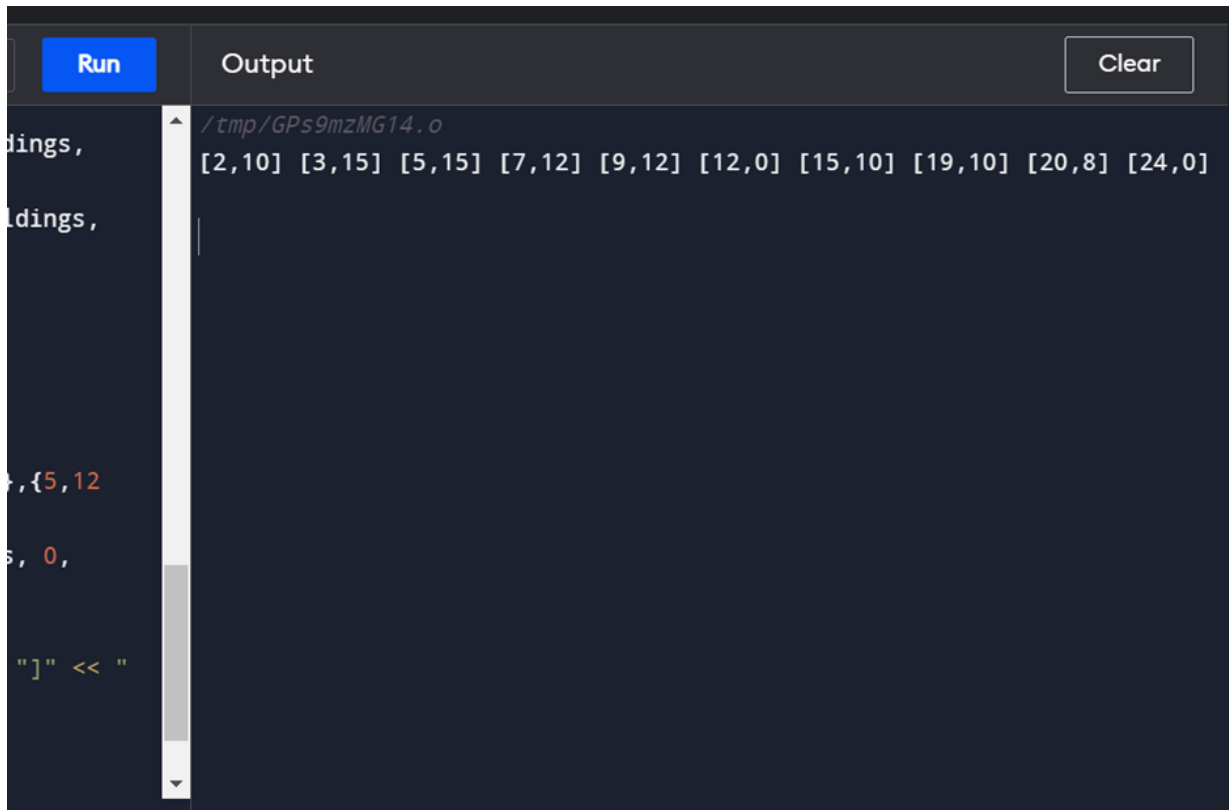
Output:

consider the following example input:

vector<vector<int>> buildings = {{2,9,10},{3,7,15},{5,12,12},{15,20,10},{19,24,8}}

The expected output for this input is:

```
/tmp/GPs9mzMG14.o
[2,10] [3,15] [5,15] [7,12] [9,12] [12,0] [15,10] [19,10] [20,8] [24,0]
```

DIVIDE AND CONQUER APPROACH

The divide and conquer approach is a more efficient solution for the skyline problem compared to the brute force approach. It involves dividing the set of buildings into two halves recursively until the subproblems can be solved easily. Then, the results of the subproblems are merged together to obtain the final solution. The first step in the divide and conquer approach is to divide the set of buildings into two halves. This can be done by selecting a mid-point and dividing the set of buildings into two subsets. The subsets are then solved recursively using the divide and conquer approach. This step is repeated until the subsets can be solved easily, which usually occurs when the subsets contain only one or two buildings. The second step is to merge the results of the subproblems. This involves finding the common skyline points between the left and right subsets, and merging them together to obtain the final skyline. To find the common skyline points, the maximum height of the buildings at each x-coordinate is computed for both subsets. Then, the points where the maximum height changes are identified and added to the final skyline.

C++ Code:

```cpp
vector<pair<int, int>> merge(vector<pair<int, int>> left, vector<pair<int, int>> right) {

    vector<pair<int, int>> result;

    int i = 0, j = 0, h1 = 0, h2 = 0;

    while (i < left.size() && j < right.size()) {

        if (left[i].first < right[j].first) {

            int x = left[i].first;

            h1 = left[i].second;

            int height = max(h1, h2);

            result.push_back(make_pair(x, height));

            i++;

        } else if (left[i].first > right[j].first) {

            int x = right[j].first;

            h2 = right[j].second;

            int height = max(h1, h2);

            result.push_back(make_pair(x, height));

            j++;

        } else {
```

```cpp
            int x = left[i].first;

            h1 = left[i].second;

            h2 = right[j].second;

            int height = max(h1, h2);

            result.push_back(make_pair(x, height));

            i++;

            j++;

        }

    }

    while (i < left.size()) {

        result.push_back(left[i++]);

    }

    while (j < right.size()) {

        result.push_back(right[j++]);

    }

    return result;

}


vector<pair<int, int>> getSkyline(vector<vector<int>>& buildings) {

    int n = buildings.size();

    if (n == 0) {

        return vector<pair<int, int>>();

    } else if (n == 1) {

        int x = buildings[0][0];

        int height = buildings[0][2];

        vector<pair<int, int>> result;

        result.push        (make_pair(x, height));
```

```cpp
        x = buildings[0][1];

        height = 0;

        result.push_back(make_pair(x, height));

        return result;

    }

    int mid = n / 2;

    vector<vector<int>> left(buildings.begin(), buildings.begin() + mid);

    vector<vector<int>> right(buildings.begin() + mid, buildings.end());

    vector<pair<int, int>> left_skylines = getSkyline(left);

    vector<pair<int, int>> right_skylines = getSkyline(right);

    return merge(left_skylines, right_skylines);

}
```

Output:

consider the following example input:

vector<vector<int>> buildings = {{2,9,10},{3,7,15},{5,12,12},{15,20,10},{19,24,8}};

The expected output for this input is:

```
Run          Output                                                    Clear

dings,       /tmp/GPs9mzMG14.o
             [2,10] [3,15] [5,15] [7,12] [9,12] [12,0] [15,10] [19,10] [20,8] [24,0]
dings,


,{5,12

, 0,

"]" << "
```

PRIORITY QUEUE APPROACH

The priority queue approach is a more efficient way to solve the skyline problem than the brute force approach. In this approach, a priority queue is used to keep track of the current heights of the buildings. The priority queue is sorted in decreasing order by height so that the highest building can be easily identified at any point in time. The algorithm for this approach starts by initializing an empty priority queue and an empty output list. Then, the buildings are sorted by their left x-coordinate in ascending order. The algorithm iterates through the buildings in this sorted order and performs the following steps for each building. While the priority queue is not empty and the current building's left x-coordinate is greater than or equal to the right x-coordinate of the building at the top of the priority queue, remove that building from the priority queue. If the height of the current building is greater than the height of the building at the top of the priority queue (or if the priority queue is empty), add the current building's left x-coordinate and height to the output list. Push the current building onto the priority queue. After iterating through all the buildings, any remaining buildings in the priority queue are added to the output list. Finally, the last key point with a height of 0 is added to mark the end of the skyline.

C++ Code:

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

    vector<pair<int, int>> result;

    int n = buildings.size();

        if (n == 0) {

        return result;

    }

    priority_queue<pair<int, int>> heights;

    int i = 0, x = 0, y = 0;

    while (i < n || !heights.empty()) {

        if (heights.empty() || (i < n && buildings[i][0] <= heights.top().second)) {

            x = buildings[i][0];

            while (i < n && buildings[i][0] == x) {

                heights.push(make_pair(buildings[i][2], buildings[i][1]));

                i++;

            }

        } else {
```

```
            x = heights.top().second;

            while (!heights.empty() && heights.top().second <= x) {

                    heights.pop();

            }

        }

        y = (heights.empty() ? 0 : heights.top().first);

        if (result.empty() || result.back().second != y) {

            result.push_back(make_pair(x, y));

        }

    }

    return result;

}
```

Output:

consider the following example input:

vector<vector<int>> buildings = {{2,9,10},{3,7,15},{5,12,12},{15,20,10},{19,24,8}};

The expected output for this input is:

Output

/tmp/GPs9mzMG14.o
[2,10] [3,15] [5,15] [7,12] [9,12] [12,0] [15,10] [19,10] [20,8] [24,0]

TIME COMPLEXITY

n is the number of buildings.

1. For Brute Force Algorithm: O(n^2): This is because, in the worst-case scenario, each building may have a unique x-coordinate, leading to n^2 iterations. This time complexity makes the brute force approach impractical for large inputs, where the number of buildings is significant.

2. For Divide and Conquer Algorithm: O(nlogn): The time complexity of the divide and conquer approach is O(nlogn), where n is the number of buildings. This is because the set of buildings is divided into two subsets at each recursive step, resulting in a tree-like structure with a depth of log(n). At each level of the tree, the maximum height of the buildings at each x-coordinate is computed, resulting in a time complexity of O(n). Therefore, the overall time complexity is O(nlogn).

3. For Priority Queue Algorithm: O(nlogn): The time complexity of the priority queue approach is O(nlogn), where n is the number of buildings. This is because each building is pushed onto the priority queue once and removed from the priority queue at most once, which takes O(logn) time for each operation. Additionally, the buildings are sorted in O(nlogn) time before iterating through them.

CONCLUSION

In the skyline problem, we have three approaches to find the skyline formed by the given buildings. The brute force approach involves iterating through every x-coordinate and finding the maximum height of the buildings at that point. The time complexity of this approach is $O(n^2)$, where n is the number of buildings. This approach is straightforward but inefficient, especially for larger input sizes, and can lead to time limit exceeded errors.

The divide and conquer approach involves recursively dividing the set of buildings into two halves and merging the results. The time complexity of this approach is $O(n\log n)$, where n is the number of buildings. This approach is more efficient than the brute force approach since it avoids redundant computations by dividing the input into smaller subproblems. However, the implementation of this approach can be more complex than the brute force approach.

The priority queue approach involves maintaining a priority queue of the heights of the buildings. The time complexity of this approach is $O(n\log n)$, where n is the number of buildings. This approach is also more efficient than the brute force approach since it avoids redundant computations by maintaining the maximum height of the buildings in a priority queue. This approach is often faster in practice than the divide and conquer approach due to a smaller constant factor, which makes it more suitable for larger input sizes.

In summary, the divide and conquer and priority queue approaches are more efficient than the brute force approach. Although the divide and conquer approach has the best theoretical time complexity, in practice, the priority queue approach is often faster due to a smaller constant factor. However, the implementation of these approaches can be more complex than the brute force approach, which is straightforward but inefficient. Therefore, the choice of approach depends on the input size and the specific requirements of the problem.

REFERENCES

1. https://www.programiz.com/cpp-programming/online-compiler/
2. https://algorithmist.com/wiki/UVa_105_-_The_Skyline_Problem
3. https://www.codingninjas.com/codestudio/library/the-skyline-problem
4. https://openai.com/blog/chatgpt