Finding Second Closest Pair of Points Using Divide and Conquer

Khondker Salman Sayeed

1805050

Target Complexity:

O(n lg n) for n points.

Complexity Analysis:

Instead of checking all pairs of points on $O(n^2)$, we use a divide and conquer approach.

Base Case:
The base case is solved in constant time for each case. We solve the problem trivially using brute force in the base case. The threshold is set for 3 points. So There are 3 combinations. Therefore the comparisons for all points is: 3 * n / 3 = n.

```
void _do_bruteforce(int lo, int hi, const std::vector<Point>& points) {
    // hi - lo <= 3, so const time.
    for (int i = lo; i < hi - 1; i++) {
        for (int j = i + 1; j < hi; j++) {
            _update_closest_two(points[i], points[j]);
        }
    }
}
```

Therefore the base case is done in O(n) for each recursive level.

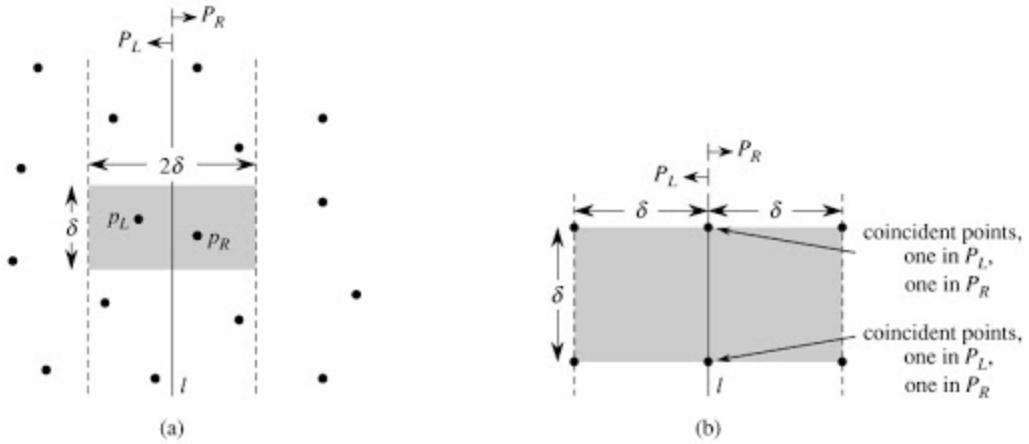
Divide Step:
The division involves calling the function recursively, and partitioning the points previously sorted on their x and y coordinates.

```
std::vector<Point> _partition(int lo, int hi, const Point& midpoint,
    const std::vector<Point>& ysorted) {


    std::vector<Point> partitioned(ysorted.size());


    int ptrl = lo, ptrr = (lo + hi) / 2;
    for (int i = lo; i < hi; i++) {
        if (ysorted[i].x < midpoint.x) {
            partitioned[ptrl++] = ysorted[i];
        } else {
            partitioned[ptrr++] = ysorted[i];
        }
    }


    return partitioned;
}
```

This partition task just goes through the entire sorted array and puts the individual points into another array based on its coordinates. This takes linear time. So complexity is O(n).

Combine Step:
Based on the result from each divided region, we have to compare the results and combine them to get one result. We take the value of the second closest from the left subproblem and the right subproblem. Now there might be cases where points from the two partitioned sides might be a better choice that the individual side points. So we check for these crossing points. The max width of these points can be only the minimum found from the subproblems. So we form a strip of points we will compare to find the second closest pair.

This might seem like a $O(n^2)$ operation, but using packing argument we show that it is linear time:



Because the points on one side of the strip have to be at least the minimum amount from the subproblems, we can argue about the point pairs we have to compare in this step. Here we can see only four points can reside in the delta square. For each point on the strip, we have to compare points in the δ x 2δ rectangle. There can be 8 points at max in that rectangle. So for each point we only have to check at most 7 points. Therefore the points on the strip can be compared in linear time.

We will use the preexisting sorted array on y coordinates to find the points in ascending y order. This will take linear time as well. If we used a sort on the y coordinate of the points, it would take O(n lgn), and our algorithm would take O(n lg^2n). Using a linear ordering process we minimize the complexity further.

Complexity O(n).

Recursive Step Total:
For each recursive step, all steps in it can be completed in O(n) time.

Therefore the recursive step is of O(n).

Number of Recursion:
We partition the array in half in each recursion. Therefore, there can be only lgn recursive calls.

Recurrence Relation:

If finding the second closest pair of an n element array takes T(n), we divide the problem as follows:
T(n) <= 2T(n / 2) + O(n).

Where the problem is divided into the same problem but half the size. Each recursive steps take O(n) in achieving partitioning and combining.

From the master theorem we have a = 2, b = 2. We have $\log_b a = 1$. Which is the same power as in O(n). Therefore the solution to this recurrence is O(n lgn).

Therefore our target complexity is achieved.