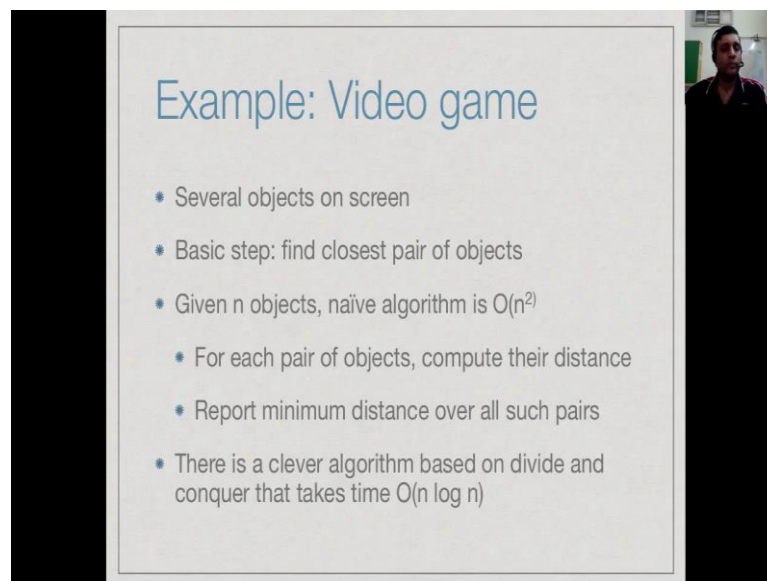


Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

Module – 07
Lecture - 38
Divide and Conquer: Closest Pair of Points

We now look at another divide and conquer algorithm, this is the geometric problem, given a set of points we would like to compute the closest pair of points among them.

(Refer Slide Time: 00:11)

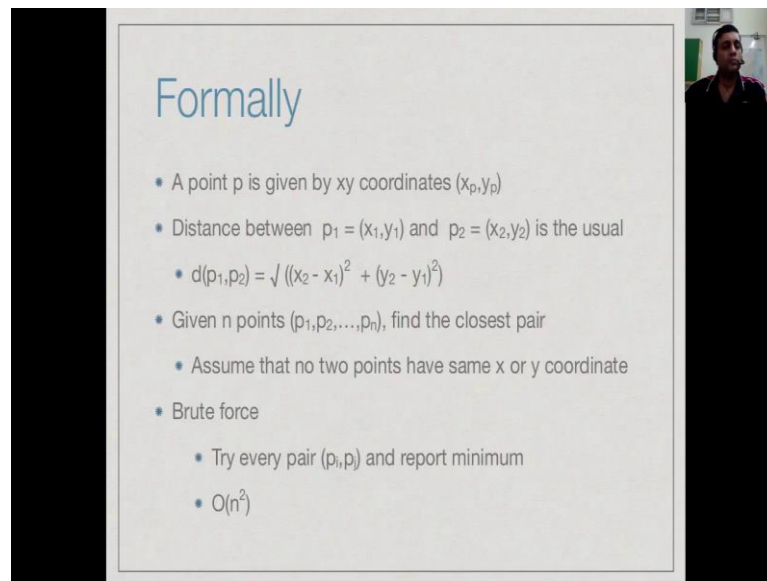


Example: Video game

- Several objects on screen
- Basic step: find closest pair of objects
- Given n objects, naïve algorithm is $O(n^2)$
 - For each pair of objects, compute their distance
 - Report minimum distance over all such pairs
- There is a clever algorithm based on divide and conquer that takes time $O(n \log n)$

So, recall that at the beginning of this set of lectures to motivate the need for more efficient algorithms, you consider the example of the video game, if there are several objects on the screen and you might want to find it at any given point, the closest pair of objects among them. So, for this again a naive algorithm could be to explicitly compute the distance between every pair of these n objects, which should be an order n squared algorithm. So, what we are going to see is that we can actually use divide and conquer and produce an order $n \log n$ algorithm for this problem.

(Refer Slide Time: 00:45)



Formally

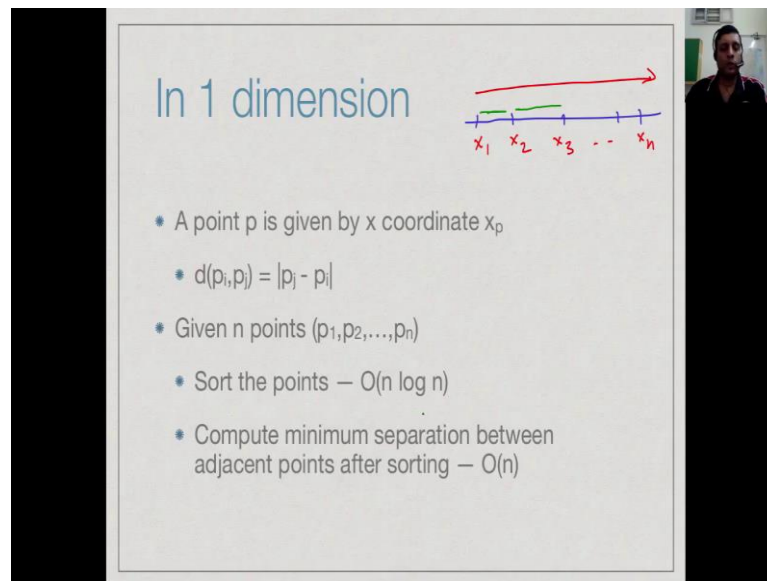
- A point p is given by xy coordinates (x_p, y_p)
- Distance between $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is the usual
 - $d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
- Given n points $(p_1, p_2, \dots, p_n)$, find the closest pair
 - Assume that no two points have same x or y coordinate
- Brute force
 - Try every pair (p_i, p_j) and report minimum
 - $O(n^2)$

So, formally we are looking at points in two dimensions, so each point is given by an x coordinate x_p , y coordinate y_p and we are using the usual utility and notion of distance that is the distance given by Pythagoras used formula, which is that the distance between p_1 and p_2 is the square root of x_2 minus x_1 whole square plus y_2 minus y_1 whole square. So, you just assume that there is a distance formula which we can use whenever we want to compute the distance between a pair of points.

So, our target is given a set of n points p_1 to p_n to find the closest pair among them and it will be convenient for the analysis of the algorithm that we are going to suggest that no two points in this set have the same x or y coordinate. So, every x coordinate among these n x coordinates is different, every y coordinate among these n coordinates is different. Now, it can be extended by algorithm we are going to show, it can be extended to deal with the case where this assumption is not true, but it will then unnecessarily complicate the understanding of the algorithm.

So, let us just assume that we are solving this special case of the problem, where every point is at a different x coordinate and at different y coordinate from every other point. So, as we have seen a brute force solution would be to try every pair, compute d of p_i, p_j and then report the minimum amount of distances. So, this would be an order n squared algorithm.

(Refer Slide Time: 02:08)



In 1 dimension

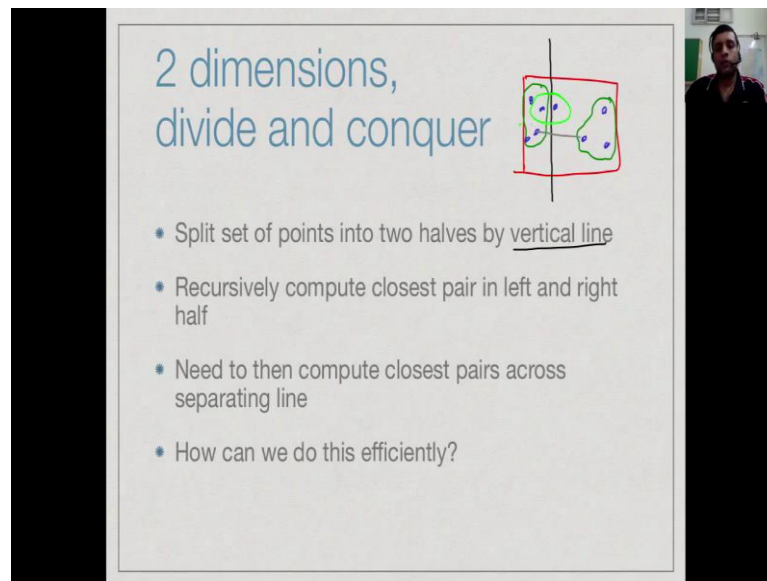
$x_1 \quad x_2 \quad x_3 \quad \dots \quad x_n$

- A point p is given by x coordinate x_p
- $d(p_i, p_j) = |p_i - p_j|$
- Given n points $(p_1, p_2, \dots, p_n)$
- Sort the points — $O(n \log n)$
- Compute minimum separation between adjacent points after sorting — $O(n)$

So, let us see first the same problem if we had only one dimensional points. If we have one dimensional points then all these points lie along the line, which we can assume this the x axis. So, we have a bunch of points and we want to find the closest point. So, what we can do is we can first sort them, so that we have the points in increasing order of x coordinate and then it is easy to see that the distances that we need are the distances between two adjacent points.

Because, if I look at a point, the nearest point is either the one on its left and one on its right. So, I just need to scan this x_2 minus x_1 distance, then x_3 minus x_2 , so I just need to scan these n minus 1 distances and then keep track of the smallest gap between these two points and that would give me the smallest distance among the overall pair of overall n points. So, here the algorithm is $n \log n$, because it takes $n \log n$ times to sort the points in x coordinate, after that finding the minimum is actually very easy. So, in one dimension this problem is very easy to solve, the challenge is to solve it in two dimensions.

(Refer Slide Time: 03:12)



2 dimensions,
divide and conquer

- Split set of points into two halves by vertical line
- Recursively compute closest pair in left and right half
- Need to then compute closest pairs across separating line
- How can we do this efficiently?

The diagram shows a set of points (blue dots) within a red rectangular boundary. A vertical black line splits the points into two groups. The points on the left are enclosed in a green oval, and the points on the right are enclosed in a green oval. A small inset image of a person is visible in the top right corner of the slide.

So, in two dimensions if we are going to use divide and conquer, we need a way of separating the points into two groups, a roughly equal size or a hopefully exactly equal size. So, a natural way is this is the geometric problem is to separate them based on their positions. So, this is my overall set of points, then it's natural to try and draw some kind of a line and say that half the points are here and half the points are there.

So, we will do it using a vertical line, so we will split this set not by some arbitrary line like this, but rather we will try and split it by a vertical line. So, somewhere which may not be half way across, because the points may be scattered in an uneven way, we will split it, so that there are exactly the same number of points to the left of the line and to the right of the line.

So, now because of the divide and conquer thing what we will do is, we will compute the smallest distance among the points to the left, separately we will compute the smallest distance points from the right. But, this does not tell us anything about distances between points on the left and points on the right and they could very well be points very close to each other across the boundary.

So, I could have added four points here, four points there and I could have added a pair of points which spans this black line, which are actually closer together, than any two points to the left or two points to the right. So, we need to compute the closest pairs across the separating line and this is the challenge.

(Refer Slide Time: 04:44)

Sorting points by x and y

- Given n points $P = \{p_1, p_2, \dots, p_n\}$, compute
- P_x , P sorted by x coordinate
- P_y , P sorted by y coordinate
- Divide P by vertical line into equal size sets Q and R
- Need to efficiently compute Q_x, Q_y, R_x, R_y

The diagram shows a set of points P in a 2D plane. A vertical red line is drawn through the points, labeled P_y with a red arrow pointing upwards. A horizontal red line is drawn at the bottom, labeled P_x with a red arrow pointing to the right.

So, let us look a little closer at how we do this. So, we will make the further step before we do this thing recursively. Given a points P we will compute points, the set of points we will compute two sorted orders. So, we will first scan these points by x coordinate from left to right and we will listed in this order and call it P_x , then we will scan this, the list P from...

So, we will sort on the y coordinate and call this P_y , so from P we will produce two list, one sorted by x and one sorted by y . Then, so we will assume that we have done this of course, we can do this we know in order $n \log n$ time right to the beginning. Now, the next step is to do this recursive call and so when we do the recursive call, because we have P_x sort it by x coordinate, we know that the line that we need to draw is the one that separates P_x into two equal parts.

So, we need to go to the midpoint of P_x and draw a line at the x coordinate separating the midpoint from the next point. So, the position of this line is fixed once we know P_x , fix meaning we know between which two points. Remember, we assume that no two points at the same x coordinate, no two points at the same y coordinate. So, if I just look for the median value or the value at the middle of x , I draw a line there.

Now, I separately doubt into two equal parts they assumption, because I have done this able midpoint of P_x . So, I have two set Q and R , but in order to continue this recursion I need to assume that Q is sorted in both x and y and so is R . So, I need to assume that I have some efficient way of extracting for the each of these some problems, a similar

ordering of the points sorted by x, sorted by y. So, I need to efficiently compute Q_x and Q_y , R_x and R_y .

(Refer Slide Time: 06:39)

Sorting points by x and y

- Need to efficiently compute Q_x , Q_y , R_x , R_y
- Q_x is first half of P_x , R_x is second half of P_x
- When splitting P_x , note the largest x coordinate in Q , x_Q
- Separate P_y as Q_y , R_y by checking x coordinate with x_Q
- All $O(n)$

The diagram illustrates a set of points P in a 2D plane. A vertical line at x_Q splits the points into two groups: Q (left) and R (right). Above the diagram, a horizontal line represents the y-axis, with points Q_y and R_y indicated. Handwritten notes show $x < x_Q \rightarrow Q_y$ and $x > x_Q \rightarrow R_y$. Below the diagram, a horizontal line represents the x-axis, with points Q_x and R_x indicated.

Q_x and Q_y is easy, because this whole thing earlier was P_x and then we split this thing midpoint. So, everything to the left of the midpoint is Q_x and everything to the right of the midpoint is R_x . So, in one scan of P_x I go up to half way and I put everything in Q_x and then in half way point I put everything in R_x . Now, what about y ? Now, the problem is as I am going up these two points are in Q_y , then this point goes into R_y because in the overall scheme of things in P_y , these are all listed globally by y coordinate.

So, I would have to move this to R , then I would to put this back into... So, next one again possibly R any one how you use this, the next one is Q and so on. So, as I am going up some points going Q_y and some going R_y . Now, how do I determine that without going over this too many times, so the key is that having done this split of x , we note this dividing line. So, we know which x coordinate separates Q from R .

So, as we go through P_y , we will look at each point and if the x coordinate is less than this midpoint, so let us call this x_Q , if it is less than x_Q , then we push it into Q_y , if it is x is greater than x_Q then we push it into R_y and because if it is originally sorted, we are building up Q_y and R_y also in sorted order. So, one scan of P_y you get Q_y and R_y . So, in linear time we can take the given sorted list P_x and P_y and separately divide into sorted list Q_x , Q_y for the left half R_x , R_y for the right half.

(Refer Slide Time: 08:29)

2 dimensions, divide and conquer

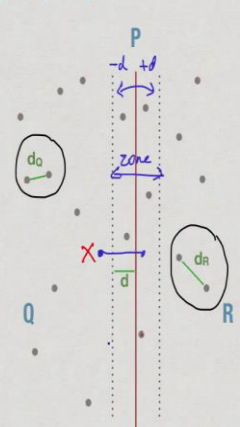
- Basic recursive call is $\text{ClosestPair}(P_x, P_y)$
- Set up recursive calls $\text{ClosestPair}(Q_x, Q_y)$ and $\text{ClosestPair}(R_x, R_y)$ for left and right half of P in time $O(n)$
- How to combine these recursive solutions?

So, now when we call our algorithm with closest pair P_x, P_y it will split recursively into closest pair Q_x, Q_y and R_x, R_y to the left and right half. So, we will, you can assume as usual that we will solve these recursively, we will get the closest distance there. Now, we have to worry about how to combine them, how to take care of these points whose distance spans the interval, the line separating the two halves.

(Refer Slide Time: 08:58)

Combining solutions

- Let d_Q be closest distance in Q and d_R be closest distance in R
- Let d be $\min(d_Q, d_R)$
- Only need to consider points across the separator at most distance d from separator
- Any pair outside this band cannot be closest pair overall



So, if we solve this problem on the left among all the points in Q , I will identify some pair of points as being the nearest pair with the distance d_Q . And now similarly, if I do solve the problem right I would identify some pair of points as being the minimum of the right with distance d_R . So, now we are interested in this smaller of these two, because

the smaller of these two is a candidate for the overall minimum distance.

So, now we are looking at points which could be within at distance d . d_Q is smaller than the d_R across this boundary. So, let d be the minimum of d_Q and d_R , so it is the smaller of these two, so in this particular example d is d_Q . Now, the claim is that if you look at this zone here which is plus minus d distance away from the separator line, then if I have some point outside this and if I want to look at any point across on the other side, this distance from here to there is key plus some distance on either side therefore, it is more than d .

So, it is more than the smaller of d_Q and d_R , so it cannot be a candidate for overall smallest distance. So, these kind of things are useless, there is no point looking for any line which is any pair whose one end point is outside this zone. Because, if it is outside the zone, then it can only be at least b plus something away from something on the other side to the line. So, it is enough to look at points inside the zone and both sides. So, we only need to consider points across the separator which lie within this plus minus d , any pair outside this cannot be the closest pair of the line.

(Refer Slide Time: 10:44)

Combining solutions

- Divide the distance d zone into boxes of side $d/2$
- Cannot have two points in same box
- Diagonal is $\sqrt{2}d/2$ *0.707d*
- Any point within distance d must lie in a neighbourhood of 4×4 boxes
- Need to check each point against 15 others

The diagram shows a grid of boxes. A central box is highlighted with a red circle. A green line segment of length d is drawn from the center of the grid to the top-right corner of the 4×4 neighborhood. The grid is divided into boxes of side $d/2$. The distance d is marked at the top, and the diagonal of the 4×4 neighborhood is marked as $\sqrt{2}d/2$ or $0.707d$.

So, let us take a closer look, so this is my plus minus these on, let us this is minus d , this is plus d . Now, what we are going to do is, we are going to further break it up into these squares of size $d/2$ by $d/2$, so I had $d/2$ plus $d/2$. So, this whole thing with d , I am going to consider all these steps. So, the claim is that inside such a box I can have at most one point, I cannot have two points. Why, because the furthest separation within a

box is across the diagonal, the further step two points can be in the box is a $2n$ points to the diagonal.

But, the diagonal of this square of size d by 2 is square root of d by 2 , so this is some points $0.047d$. So, this is less than d strictly less than d , but notice that this square is completely on one side the right times, either on the left or the right and both on the left and right we know that the minimum separation is d , there is no point on one side of the line there are no two points closer than d , because d is a minimum of d_Q and d_R .

So, therefore in each of these boxes we can have at most one point and now we start looking at any points. So, look at a point here in this box, now the claim is that if I have to be within distance d , then how far away can I be well if you think about it the furthest you can be or the furthest box you can go to, go to that box. So, you can be a little more careful about is, but it certainly cannot be any further than that box.

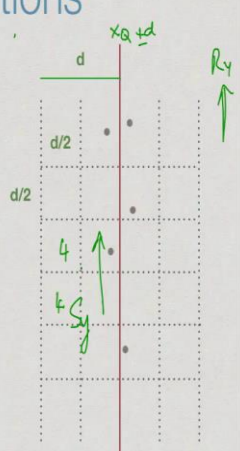
So, if you have to go from this box basically if we have to go more than this distance away, then it will be more than d away. So, the claim is that any point within this distance d must lie within the next 4 by 4 segments. So, we have to compare each point only against 15 . Now of course, we will compare it will one scan from bottom to top. So, this could have the points below this they could again compared when we consider those and before.

So, we will consider, we will do this scan in this order as we will see in the net. So, therefore, we only need to consider this point against points in these 16 squares around it. So, that is the fix number independent of how many points they actually have.

(Refer Slide Time: 13:09)

Combining solutions

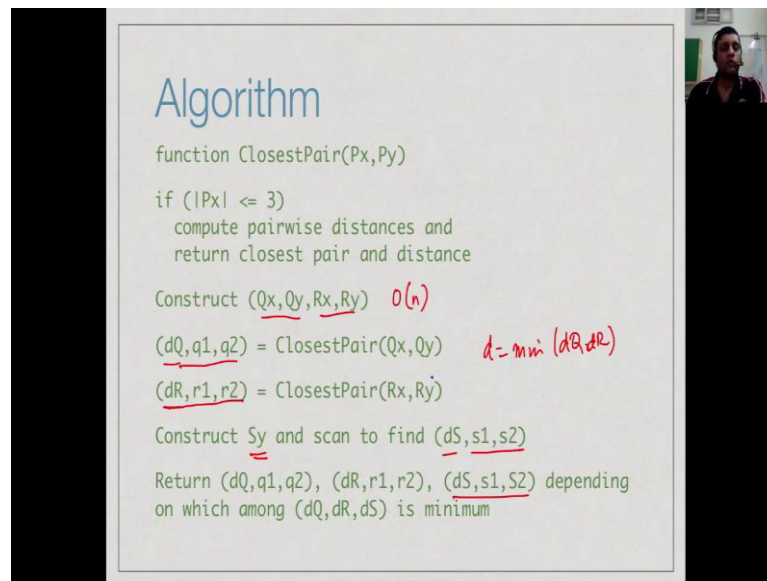
- From Q_y , R_y , extract S_y , points in d band sorted by y coordinate
- Scan S_y from bottom to top, comparing each point against next 15 points in S_y
- Linear scan



So, formally what we will do is we have this and this side we have Q_y which is sorted and this side we have R_y which is sorted, but remember they are both sorted. So, we will do a kind of merge, so we will go through Q_y and R_y we will pick the next one in sorted y order between Q_y and R_y . If the x coordinate is remember this is an x_Q are originally separating point plus d and minus d . If the x coordinate of the point that be find is between x_Q plus d and x_Q minus d , then we will added to a list S_y .

So, S_y will now the sorted list of points within this pan from bottom to top extracted from Q_y and R_y in linear line. Now, within this list these boxes will now we are all ordered. So, if I just scan them then I will find that I all these... So, they would be possibly 4 points which come from these 4 boxes, 4 from these 4 boxes and so on. So, I can definitely find the next 15 points in this list and a compare only with these. So, this is the linear scan.

(Refer Slide Time: 14:15)



```
Algorithm

function ClosestPair(Px, Py)

if (|Px| <= 3)
    compute pairwise distances and
    return closest pair and distance

Construct (Qx, Qy, Rx, Ry)  $O(n)$ 

(dQ, q1, q2) = ClosestPair(Qx, Qy)  $d = \min(dQ, dR)$ 
(dR, r1, r2) = ClosestPair(Rx, Ry)

Construct Sy and scan to find (dS, s1, s2)

Return (dQ, q1, q2), (dR, r1, r2), (dS, s1, s2) depending
on which among (dQ, dR, dS) is minimum
```

So, in other words we have this following algorithm, so we start with by assuming that we have set off a problem. So, that P has been split into two copies P x and P y sort it by x, sort it by y. Now, if we have less than 3 points, 3 points are less then, we just do it by brute force compute them closest pair and return the answer. So, the answer consists of the distance on a pair, the points which are at the distance, if it is more than 3 then we do this recursive thing.

So, we from P x and P y as we said we construct Q x, Q y and R x, R y and this we say we can do linear time. Then, we have recursively solve this solution, the solve problem of Q and R and we will get these two distances d x and d R from this we will take d to be the minimum of d Q and d R. So, using this we will set up this S y the list of all points inside that zone and then we will scan that list to find within that zone which are the points, which are in closest distance d S.

Now, if d S is more than d Q the minimum of d Q and d R, then s does not contribute anything, but if it is less gives as the correct answer. So, we will return either the d Q the answer produce from the left or the answer produce from the right or the answer produced from our in between zone depending on which of these distances as the smallest. So, this is the basic algorithm you have to do a little bit more work actually code it correctly, but this gives the overall structure of the algorithm.

(Refer Slide Time: 15:43)

Analysis

$T(n) = 2T(n/2) + n$

- Computing (P_x, P_y) from P takes $O(n \log n)$
- Recursive algorithm
 - Setting up (Q_x, Q_y, R_x, R_y) from (P_x, P_y) is $O(n)$
 - Setting up S_y from Q_y, R_y is $O(n)$
 - Scanning S_y is $O(n)$
 - Recurrence is same as merge sort
- Overall $T(n) = O(n \log n)$

$O(n \log n)$

So, we have a non recursive part which is to first compute the initial sort of list P_x and P_y , this takes order on $n \log n$ time force. Because, sorting text $n \log n$ time then having set of this recursive thing the overall recursive problem divides n points into two sets of n by 2. So, we have a familiar thing which is T of n is 2 times T of n by 2 and then because all the setting up time and all these combining time is linear, we have exactly the recurrence for merge sort. So, therefore, the recursive part also gives as order $n \log n$, so we have a initial sorting phase which $n \log n$, we have a recursive part which is $n \log n$ and therefore, overall this algorithm is $n \log n$.