Algorithms Fall 2019

Lecture : Inversions and Closest Pair

Imdad Ullah Khan

Contents

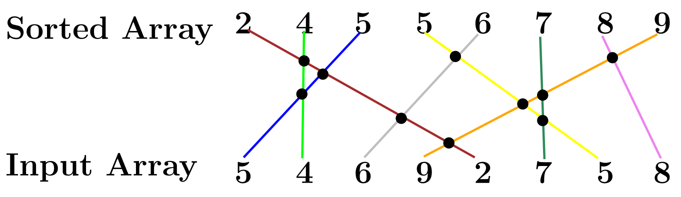
1 Divide and Conquer

The merge sort algorithm is the canonical example given for the Divide and Conquer design paradigm. The divide and conquer approach requires recursively breaking a problem into smaller subproblems (the divide part) until they become easy enough to be solved directly. In merge sort, for example we keep dividing the array into two parts until the size of the array is 1 and we know how we can sort an array with just one element (It's already sorted). The next step is to somehow combine the solutions of the smaller subproblems to obtain the solution of the original problem (the conquer part). This step is usually tricky. In merge sort we used the "merge" algorithm to sort the larger array from two sorted halves.

Next, we will discuss two more Divide and Conquer algorithms. The first is the problem of counting inversions, as follows. The second is the problem of finding closest pair of points among a set of points in 2-dimensional Euclidean space. We present the brute-force approach to solve this problem and improve it using the divide and conquer approach.

2 Counting Inversions

Problem : Given an array A an inversion is defined as a pair i, j such that $i < j \ \wedge \ A[i] > A[j]$. We want to find how many such pairs are in A.



Crossing Points (black dots) represent inversions

Example : Let A = $\begin{array}{|c|c|c|c|c|c|c|c|}\hline 5&4&6&9&2&7&5&8\\\hline\end{array}$. Then the inversions for this array are

$$\{(1,2),(1,5),(2,5),(3,5),(3,7),(4,5),(4,6),(4,7),(4,8),(6,7)\}$$

Solution : One way to solve is problem is to simply check all pairs i, j and see how many of them are inversions. There are $\binom{n}{2}$ pairs so that would take about $n^2/2$ steps. That algorithm would work as follows

Algorithm 1 : Counting Inversions Using Brute Force

```
count ← 0
for i = 1 to n do
    for j = i+1 to n do
        if A[i] > A[j] then
            count ← count + 1
```

As computer scientists, however, we always want to come with better, more efficient algorithms. But why should we think that a better solution should exist for this problem? Well one reason is that this problem is somewhat related to sorting an array (Infact the number of inversions is used as a measure for the sorted-ness of an array). When we're sorting an array we're basically removing all the inversions. And we know we can sort an array in $n\log(n)$ steps. The algorithm for that was discussed in the last class. So it makes sense that there should be a similar algorithm that can count inversions in about the same time too. We used the divide and conquer approach to obtain a better running time for sorting. So let's try to do something similar for counting inversions too. How should be break this problem?

If we divide an array into two halves, then we can split the inversions into 3 types: Left-Left, Right-Right inversions and Left-Right inversions. For example, consider the following array A:

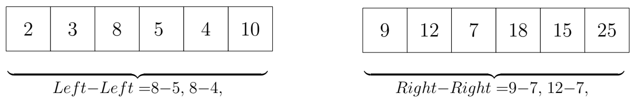
2	3	8	5	4	10	9	12	7	18	15	25

Divide the array into two.

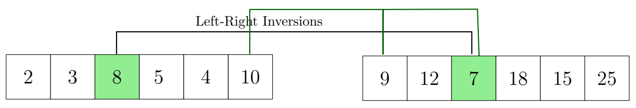
2	3	8	5	4	10

9	12	7	18	15	25

Recursively count inversions in each half.



Count inversions where a_i and a_j are in different halves and return total inversions count.



We'll refer to the left and right halves of the array as L and R. Then Left-Left inversions are those which only involve pairs from L. Similarly the Right-Right inversions are the inversions that involve pairs from only R. Left-Right inversions are pairs that involve one element from L and one element from R. The Left-Left and Right-Right inversions will be found by recursively dividing the array into even smaller parts. The problem is finding the Left-Right inversions. If we can find the Left-Right inversions in n steps then we can solve the problem in $n \log(n)$ steps. The algorithm would have the same recurrence relation as the merge sort algorithm and hence the same running time.

The problem of finding the left-right inversions is equivalent to finding the ranks in R of all the elements of L. We know that if the two arrays R and L are sorted then we can find the ranks in n steps. So while counting the inversions in the left and right halves of the array we'll do some extra work and also sort them. The idea is that even though we do the some extra work in sorting, the overall time will be reduced because now we can find the inversions much faster. Let's see how that algorithm would work.

Algorithm 2 : Counting Inversions using Divide and Conquer

```
function CountInversions(A)
    if Size($A$) == 1 then return $(A, 0)$
    $L \leftarrow A[1, 2, \ldots, n/2]$
    $R \leftarrow A[n+1, n+2, \ldots, n]$
    $(sortedL, inv_{l,l}) \leftarrow$ CountInversions($L$)
    $(sortedR, inv_{r,r}) \leftarrow$ CountInversions($R$)
    $inv_{l,r} \leftarrow sum$(FindRanks($L, R$))                    ▷ takes $n$ steps
    return (Merge($L, R$), $inv_{l,l} + inv_{r,r} + inv_{l,r}$)      ▷ merge takes $n$ steps
```

So our recurrence relation turns out to be

$$T(n) = 2T\left(\frac{n}{2}\right) + 2n$$

And solving this we get

$$T(n) = 2n \log(n)$$

Which is a huge improvement over the n^2 brute force algorithm.

2.1 Collaborative Filtering

One of the areas in which the problem of counting inversions comes up is in Recommender Systems. A recommender system tries to predict the 'rating' a user gives to a particular item. Like how YouTube tries to recommend you videos based on your viewing history, or how IMDb recommends movies. One technique used in recommender systems is that of Collaborative Filtering. "Collaborative filtering is a method of making automatic predictions (filtering) about the interests of a user by collecting preferences or taste information from many users (collaborating). The underlying assumption of the collaborative filtering approach is that if a person A has the same opinion as a person B on an issue, A is more likely to have B's opinion on a different issue x than to have the opinion on x of a person chosen randomly" (Wikipedia).

The way this is done is as follows. Based on your activity on the website(How many times you view a particular item, the review you've given some item etc) the website builds a list containing your relative preference to various items.

Table 1: User 1

$item_A$	$item_B$	$item_C$	$item_D$	$item_E$
1	2	3	4	5

It then builds a similar list for other users. It then finds the users most similar to you by counting inversions between your list and their's. Once the website has a list of similar users it can then suggest items to you based on what these similar people like.

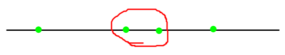
Table 2: User 2

$item_A$	$item_B$	$item_C$	$item_D$	$item_E$
3	1	2	4	5

3 Closest Pair

The second problem is as follows: Given an array of points in the plane, find the pair of points which is closest with respect to Euclidean distance.

Naive Approach: Notice that if points are in $1-D$, (an array of real numbers), with sorting it is easy to find the closest pair. In $2-D$, a simple way to solve the problem would be to compare every point with every other point and find minimum distant pair. This takes $O(n^2)$ time.



Divide and conquer approach

Input: $2-D$ array P of points.

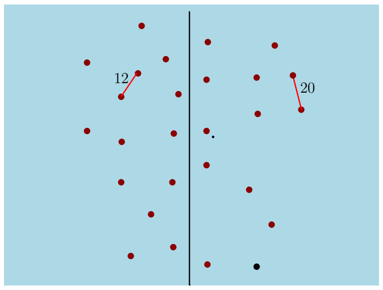
Step 1. *Sort the array P with respect to $x-$coordinates to get $2-D$ array P_x and sort with respect to $y-$coordinates to get $2-D$ array P_y.*

Divide Part:

Step 2. *Consider the mid point (median) m of array P_x. Lets the set of points on the left side of m be S_1 and on right of m be S_2. We make array of points in S_1 and call it P_{1x} which is in fact left sub-array of P_x and P_{2x} with points in S_2. Note that P_{1x} and P_{2x} are sorted in $x-$direction. We also compute arrays P_{1y} and P_{2y}, which are essentially points in S_1 and S_2 respectively but sorted in $y-$direction. We compute the arrays P_{1y} and P_{2y} in the following way.*

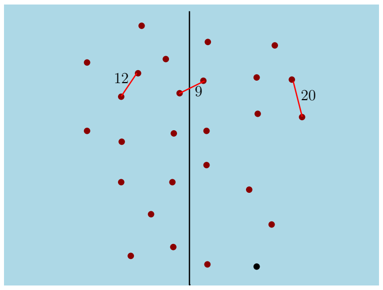
1. *Pick first point in P_y and compare its $x-$coordinate with m.*
2. *If $x \leq m$, insert the point in P_{1y}, otherwise insert in P_{2y}*
3. *continue inserting elements in both arrays.*

Note that points are inserted in sorted order in both arrays since P_y is sorted. ***Analysis*** *It takes $O(n)$ steps to make the arrays P_{1y}, P_{2y}, P_{1x} and P_{2x}.*



Conquer Part:

Step 3. *Recursively find closest pairs in S_1 with input data $P_{1x}, P_{1,y}$, and S_2 with input data P_{2x}, P_{2y}, with shortest distance δ_1 and δ_2 respectively.*



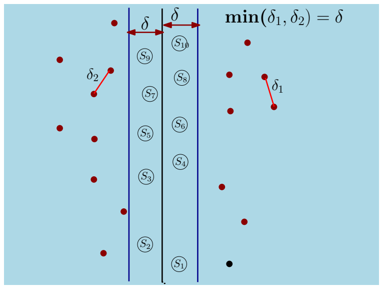
Consider smaller distance of both δ_1 and δ_2. Let $\delta = min\{\delta_1, \delta_2\}$. Only thing left to consider in the problem is, what if closest pair has one point in S_1 and other in S_2?

Combine Part:

Step 4. *Consider $N_1 \subseteq S_1$ and $N_2 \subseteq S_2$ such that points in N_1 and N_2 are in δ distance strip of m. Sort $N_1 \cup N_2$ with respect to $Y-$coordinates of points. We compute and sort points in $2-\delta$ strip of m in the following way.*

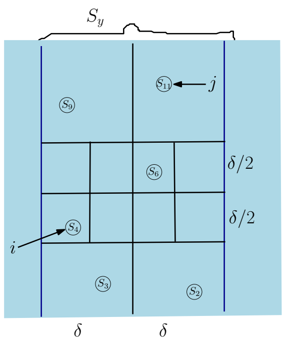
1. *Consider first point in P_y array and check if its $x-$coordinate is less than δ distance from m.*
2. *if $x-$distance from m is less than δ then put this point in array $N = N_1 \cup N_2$.*
3. *check this for every point in P_y.*

Analysis *This step takes $O(n)$ time.*



Note that if the closest pair has one point in S_1 and one in S_2, then that pair must be in N.

For each point x of N, there can be atmost 7 points in N such that distance of x with those 7 points is less than δ. Since if x has distance less than δ with some point, it will be in one of its neighboring squares, shown in diagram below. And there can not be more than one point in any of the small squares otherwise it will contradict the minimality of δ. We are not considering squares below the point x because these points are sorted and points below have been processed (assuming that traversing is in ascending order).



Step 5. *Compute the distance of each point in N to next 7 points in the array and compare it with δ. If distance is smaller than δ then update the closest pair and continue.*

Analysis *This step takes $O(n)$ time to execute since each point is being compared to constant number of other points.*

Runtime Analysis: In each divide part, we get two half size problems and combining takes $O(n)$ time. So run time for the whole algorithm is

$$T(n) = \begin{cases} 1 & \text{if } n < 3 \\ 2T(\frac{n}{2}) + O(n) & \text{if } n \geq 3 \end{cases}$$

Solving this recursively, $T(n) = O(nlogn)$