Geometrical Algorithms

Taein Kim, Utkarsh Goyal

May 2021

1 Introduction

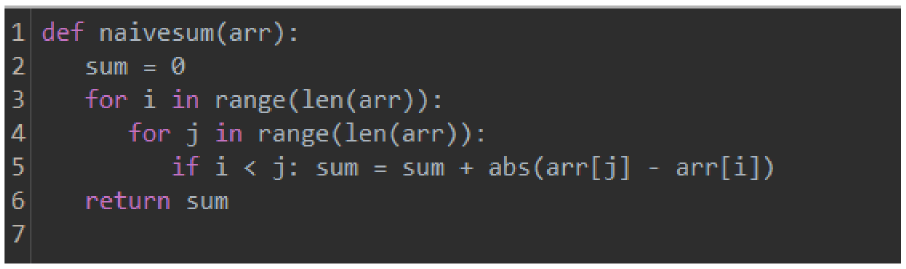
Geometrical Algorithms are algorithms that deal with with distance, area, and other geometrical components in a 2D or 3D plane. The input is usually the location of some given points and lines, which you will have to write code to calculate the result.

2 Sum of Manhattan Distances Between a Set of Points

This is a simple but prime example of a geometrical algorithm problem. The Manhattan distance between two points is the sum of their x- and y-coordinate differences. Given a set of points, you want to find the sum of every Manhattan distance between every pair.

2.1 The Naive Approach

The easiest approach to go about this problem is using a nested loop to recurse through every pair of points. Although this is simple, it runs in $O(n^2)$ time and is inefficient. Here is an example:



```python
def naivesum(arr):
   sum = 0
   for i in range(len(arr)):
      for j in range(len(arr)):
         if i < j: sum = sum + abs(arr[j] - arr[i])
   return sum

```

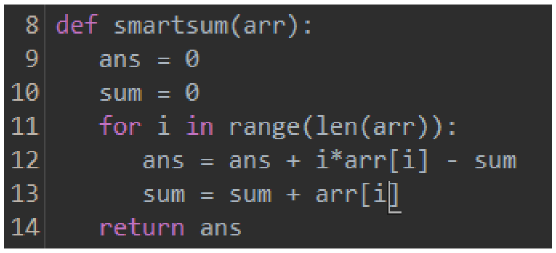
Figure 1: The Naive Code

2.2 The Efficient Approach

For us to run the same program in $O(NlogN)$ time, we need to realize what adding one point to an existing array does to the algorithm. We simply have to add the sum of every distance from the new point. Thus, with a sorted array,

$$newsum = oldsum + \sum (x_i - x) = oldsum + x_i * i - \sum x$$

This is implemented in code:



```python
def smartsum(arr):
    ans = 0
    sum = 0
    for i in range(len(arr)):
        ans = ans + i*arr[i] - sum
        sum = sum + arr[i]
    return ans
```

Figure 2: The Efficient Code

3 Klee's Algorithm

Given an array of segments on a line, is there a way to find the length of the union of segments? Hint: you'll have to find a way to subtract the overlaps from the total length of the segments (Greedy Algorithm?) Try to make it run in O(N log N) time.

3.1 Explanation

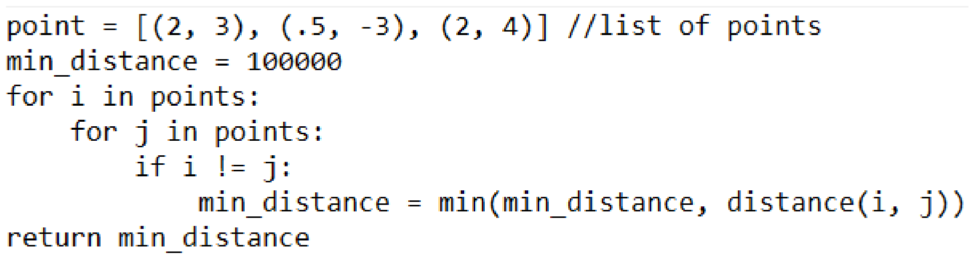
Obviously in a union, the order of the coordinates does not matter; you can just store every coordinate in a vector array, indicating whether the coordinate is a starting point or endpoint. After that, sort the array and traverse; keep count of the number of "open" sections, making sure that you reach 0 at the end. Otherwise, only add the difference between the previous and current points to your sum when there are open lines traversing the current section.

4 Divide and Conquer

The divide and conquer algorithm works exactly as the name suggests; you divide and then you conquer. This may sound really simple at first, but this algorithm is used a lot in problems involving geometric algorithms. One very common example is finding the closest pair of points in O(n log n^2).

4.1 Naive Approach

Naive Approach: As many of you probably know, you can just go through each possible pair of points and just find the maximum distance. However, this takes $O(n^2)$ time.



```
point = [(2, 3), (.5, -3), (2, 4)] //list of points
min_distance = 100000
for i in points:
    for j in points:
        if i != j:
            min_distance = min(min_distance, distance(i, j))
return min_distance
```

Figure 3: Naive Code to find minimum distance between two points

4.2 Divide and Conquer Approach

Here are the following steps:

1. Sort the array with respect to the x-coordinates(it doesn't really matter if you were to sort with respect to y-coordinates)
2. Split the array into two halves, from $0 - n/2$ and $(n/2 + 1) - (n - 1)$
3. Recursively find the smallest distances between points from each of the array, and then find the minimum of the two distances you get. Let's call this value d.
4. Now, since we haven't accounted for the distance of points that are in both strips, we realize that we only have to consider the ones that are d away from the middle point. This is because if the distance from the middle is greater than d, then it won't have a smaller distance than the distance we've already gotten.(Reference Figure 4)
5. We can sort all the points from step 4 by y-coordinate(O(n log n)); its worth mention that this can be optimized to O(n) if you use merge sort.
6. From here, we find the smallest distance in this sorted set of points.
7. We finally just return the minimum of d and the value from the previous step.

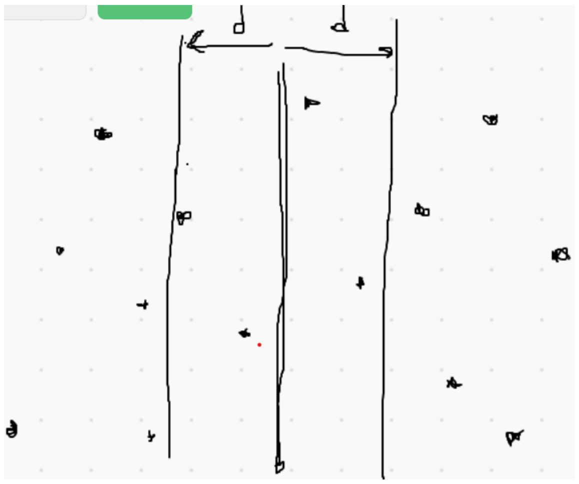


Figure 4: Reference for step 4 in divide and conquer algorithm

4.3 Further Explanation of Step 6 and Time Complexity

For step 6, even though this may seem like $O(n^2)$ time complexity, we can actually make it $O(n)$ as its sorted. In reality, we actually just have to check 7 points after the current one. (I'll leave proving this as an exercise to the reader.) To find the overall time complexity of a recursive algorithm like this, we can mark the complexity as $T(n)$. Then, with dividing the array into two arrays of reduced length, $T(n/2)$, and finding and sorting the strip of coordinates by y-coordinates in O(n log $n + n$), and finding the minimum distance in the strip in $O(n)$, we have $T(n) = T(n/2) + O(n \log n + n) + O(n)$. Simplifying this, we get $T(n) = T(n(\log n)^2)$.

5 Challenge Problem

Given a set of points on the 2D plane, how could we find the maximum number of them that lie on the same line?