Solution: K Closest Points to Origin

Let's solve the K Closest Points to Origin problem using the Top K Elements pattern.

We'll cover the following

Statement

Given a list of points on a plane, where the plane is a 2-D array with (x, y) coordinates, find the k closest points to the origin $(0, 0)$.

> **Note:** Here, the distance between two points on a plane is the Euclidean distance: $\sqrt{x^2 + y^2}$

Constraints:

- $1 \leq$ `k` $\leq$ `points.length` $\leq 10^4$
- $-10^4 <$ `x[i]`, `y[i]` $< 10^4$

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

When thinking about how to solve this problem, it may help to solve a simpler problem—find the point closest to the origin. This would involve a linear scan through the unsorted list of points, with, at each step, a comparison between the closest point discovered so far and the current point from the list. The point closer to the origin would then continue as the candidate solution. This has a runtime complexity of $O(n)$ as opposed to the naive solution of sorting the points by distance from the origin and picking the first one, which has a complexity of $O(n \log n)$.

For this reason, when extending the solution to the k closest points to the origin, we'd ideally like to do one scan through the list of points. However, if we have to check the current set of k closest points with the current point under consideration at each step, we'll end up with a time complexity of $O(n.k)$.

Optimized approach using top K elements:

Through a little organization of our set of k closest points, there is a way to reduce the number of comparisons at each step: By storing these points in a max-heap that is sorted on the distance from the origin, we get points in a max-heap that is sorted on the distance from the origin, we get $O(1)$ access to the point, among these k points, that is farthest from the origin.

Now, instead of comparing all k points with the next point from the list, we simply compare the point in the max-heap that is farthest from the origin with the next point from the list. If the next point is closer to the origin, it wins inclusion in the max-heap and ejects the point it was compared with. If not, nothing changes.

In this way, at every step of the scan through the list, the max-heap acts like a sieve, picking out the top k points in terms of their distance from the origin.

And as we'll see, the time complexity is much better than $O(n.k)$.

The Euclidean distance between a point P(x, y) and the origin can be calculated using the following formula:

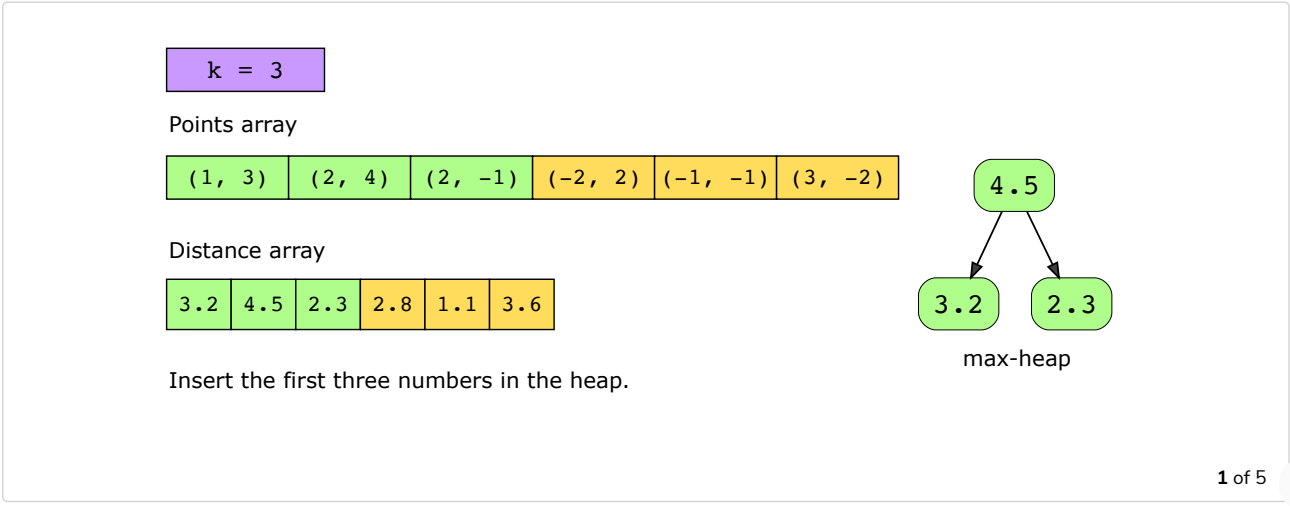
$$\sqrt{x^2 + y^2}$$

Now that we can calculate the distance between $(0, 0)$ and all the points, how will we find the k nearest points? As discussed above, the heap data structure is ideal for this purpose—we'll use a custom comparison function to define the order of the elements in a heap. Since we plan to populate the heap with coordinate pairs, we'll define a class `Point` that will contain `x` and `y` coordinates of a point and a function `distFromOrigin()` that will calculate the distance from the origin. We'll will compare the distances of the two points relative to the origin. The point closer to the origin will be considered less than the other point. We'll iterate through the given list of points, and if we find one that is closer to the origin than the point at the root of the max-heap, we do the following two things:

- Pop from the max-heap—that is, remove the point in the heap farthest from the origin.
- Push the point that is closer to the origin onto the max-heap.

As we move through the given list of points, this will ensure that we always have the k points in our heap that are the closest to the origin.

Below is an illustration of this process.



Let's look at the code for this solution below:

Java

main.java

Point.java

```java
class ClosestPoints {
    public static List<Point> kClosest(Point[] points, int k) {
        PriorityQueue<Point> maxHeap = new PriorityQueue< >((p1, p2) -> p2.di
        // put first 'k' points in the max heap
        for (int i = 0; i < k; i++)
            maxHeap.add(points[i]);

        // go through the remaining points of the input array, if a Location
        // of the max-heap, remove the top Location from heap and add the Loc
        for (int i = k; i < points.length; i++) {
            if (points[i].distFromOrigin() < maxHeap.peek().distFromOrigin())
                maxHeap.poll();
                maxHeap.add(points[i]);
            }
        }

        // the heap has 'k' points closest to the origin, return them in a li
        return new ArrayList< >(maxHeap);
    }

    public static void main(String[] args) {
        Point[] pointsOne = new Point[] {
            new Point(1, 3), new Point(3, 4), new Point(2, -1)
        };
        Point[] pointsTwo = new Point[] {
        Point[] pointsThree = new Point[] {
```

K Closest Points to Origin

Solution summary

To recap, the solution to this problem can be divided into the following parts:

1. Push the first k points to the max-heap.
2. Calculate and compare the distances of the points list with the distance of the top of the max-heap.
3. If the point distance is smaller than the max-heap top's distance, pop the top from the max heap and push the point.

Time complexity

The algorithm's time complexity is $O(n\ logk)$, where n is the total number of points and k is the number of points closest to the origin. This is because we need to iterate over all the n points and perform operations on a heap of size k, which takes $O(n\ logk)$ time in the worst case.

Space complexity

The memory complexity will be $O(k)$ because we need to store k points in the heap.

Mark as Completed