

K Closest Points to Origin

Euclidean distance + Max-Heap of size K — geometry meets heaps ■

● MEDIUM

■ Hook

GPS apps find your nearest restaurants. Recommendation engines find your closest interests. Machine learning finds nearest neighbors. All of these reduce to: **find the K closest points to some origin**. This problem teaches applying the heap pattern to custom distance metrics.

■ Core Intuition — Max-Heap to Keep Closest

Use a **max-heap of size K** ordered by distance. The root = the farthest among the K closest. When a new point is closer than the root (farthest in heap), it evicts the root and joins. At the end, all K elements in the heap are the K closest points.

■ Squared Distance — Skip the sqrt!

Distance = $\sqrt{x^2 + y^2}$. Comparing $\sqrt{a} < \sqrt{b}$ is equivalent to $a < b$ (since sqrt is monotonic). Use x^2+y^2 directly — avoids floating point and is faster. This is the standard optimization.

■ Visual — points=[[1,3],[-2,2],[5,8],[0,1]], K=2

```
Distances2 from origin:
[1,3]:  12+32 = 10
[-2,2]: 4+4   = 8
[5,8]:  25+64 = 89
[0,1]:  0+1   = 1
Max-heap (by dist2), size K=2:
Add [1,3] (10):  heap=[(10,[1,3])]           size=1
Add [-2,2] (8):  heap=[(10,[1,3]),(8,[-2,2])] size=2
Add [5,8] (89):  89 > 10 → poll max(10), add 89
                heap=[(89,[5,8]),(8,[-2,2])]
Add [0,1] (1):   1 < 89 → poll max(89), add 1
                heap=[(8,[-2,2]),(1,[0,1])]
★ ANSWER = [[-2,2],[0,1]] (the 2 closest points)
```

■ Code Implementation (Java)

```

import java.util.PriorityQueue;
public class KClosestPoints {
    public int[][] kClosest(int[][] points, int k) {
        // Max-heap ordered by squared distance (largest first)
        PriorityQueue<int[]> maxHeap = new PriorityQueue<>(
            (a, b) -> (b[0]*b[0] + b[1]*b[1]) - (a[0]*a[0] + a[1]*a[1])
        );
        for (int[] point : points) {
            maxHeap.offer(point);
            if (maxHeap.size() > k) {
                maxHeap.poll(); // evict the farthest
            }
        }
        int[][] result = new int[k][2];
        for (int i = 0; i < k; i++) {
            result[i] = maxHeap.poll();
        }
        return result;
    }
}
// Note: comparator (b - a) = MAX-heap (largest distance at root)
// We want to KEEP closest → evict FARTHEST → max-heap makes sense

```

■ Time & Space Complexity

■ TIME COMPLEXITY

$O(n \log K)$

■ SPACE COMPLEXITY

$O(K)$

n points $\times O(\log K)$ per heap operation = $O(n \log K)$. Space: $O(K)$ for the heap. Sorting all: $O(n \log n)$ — worse when $K \ll n$.

■ Common Mistakes

■ Using min-heap instead of max-heap here

For 'K closest', use a MAX-heap so the farthest (largest distance) is at the root for easy eviction. Min-heap would keep evicting the closest, which is backwards.

■ Practice Problems

#	Problem	Diff	Pattern
1	K Closest Points to Origin (LC 973)	Medium	Max-Heap by dist ²
2	Find K Closest Elements (LC 658)	Medium	Binary Search + Two Pointer

3	Kth Nearest Obstacle Queries	Medium	Max-Heap
4	Closest Binary Search Tree Values II	Hard	Heap/Deque
5	K-diff Pairs in an Array (LC 532)	Medium	HashMap

■ Quick Revision

- ✓ Use MAX-heap of size K for 'K closest' problems
- ✓ Max-heap root = farthest of the K closest — easy to evict
- ✓ Comparator: (b-a pattern) for max-heap in Java
- ✓ Use x^2+y^2 instead of $\text{sqrt}(x^2+y^2)$ — same ordering, no float
- ✓ Time: $O(n \log K)$ | Space: $O(K)$
- ✓ Pattern: 'K best by some metric' = heap of size K

■ MENTAL MODEL — Remember This Forever

Think of the max-heap as a 'waiting room with K seats, sorted by how far away you live'. The person sitting in the worst seat (farthest = root of max-heap) is always at risk of being replaced. When someone closer arrives, they kick out the farthest person and take their seat. At the end, all K people in the room are the closest.