## KDTree vs BruteForce in terms of **Time and Memory Complexity**

**🧮 Assumptions**

* Let N = number of points in the point cloud
* Let k = number of neighbors (typically a small constant like 20)
* Each point is in 3D space (XYZ), so dimensionality d = 3

**1. Brute-Force Method**

**🔁 Time Complexity**

* For each of the N points, we compute distances to **all other N points**:
* O(N × N) = O(N²)
* For each point, after getting all distances, we sort them to get the k smallest:
* O(N log N) per point → total: O(N² log N)

However, in practice, we often only need the top k smallest values, which can be done in **O(N)** per point using a min-heap or partial sort:

So overall: O(N²)

**🧠 Memory Complexity**

* Store the N × 3 input array → O(N)
* For each query point, we store N distances → O(N)
* Final distances array → O(N)

**Total:** O(N²) (temporarily for distances), but space-efficient if reusing memory per query.

**2. KDTree Method (with scipy.spatial.KDTree)**

**🔁 Time Complexity**

* Build the KDTree:
* O(N log N)
* For each point, query its k nearest neighbors:
  + Each query: O(log N) for k=1, but O(log N + k) in practice
  + Total for all points: O(N log N + Nk)

**Total:** O(N log N + Nk) ≈ O(N log N) if k is small

**🧠 Memory Complexity**

* Store the tree: O(N)
* Temporary storage for distances: O(N)
* Final inlier mask or distances: O(N)

**Total:** O(N) — much better scalability

**🥊 Summary Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | Time Complexity | Memory Complexity | Pros | Cons |
| Brute Force | O(N²) | O(N²) (temp) | Simple to implement | Slow for large N |
| KDTree | O(N log N + Nk) ≈ O(N log N) | O(N) | Fast, scalable | Requires scipy or similar |

**👀 When to Use What?**

* **Brute Force**: Small point clouds (<10k points), educational purposes, or no external libs allowed.
* **KDTree**: Large point clouds or real-time performance needed.