973. K Closest Points to Origin

Medium | Geometry | Array | Math | Divide and Conquer | Quickselect | Sorting | Heap (Priority Queue) | Leetcode Link

Problem Description

The problem provides us with an array of points called `points`, where each point is itself an array containing an x (`x_i`) and y (`y_i`) coordinate. These points are located on a two-dimensional plane. Additionally, we're given an integer `k`. Our task is to find the `k` points from the array that are closest to the origin point (0, 0) based on the Euclidean distance. The Euclidean distance between two points `(x1, y1)` and `(x2, y2)` on the X-Y plane is the square root of `(x1-x2)^2 + (y1-y2)^2`. The solution should return these `k` closest points in any order, and it is noted that the points returned will be unique in terms of their position in the output list.

Intuition

To solve this problem, we need to determine how far each point in the array is from the origin. In geometrical terms, we are interested in the magnitude of the vector from the origin to the point, which corresponds to the Euclidean distance. However, since we only care about the relative distances for sorting (finding the smallest distances) and not the exact distances themselves, we can avoid calculating the square root to simplify the comparison. By comparing the squares of the distances, we can maintain the same ordering as we would if we used the actual distances.

Here is the thought process behind the solution:

1. We recognize that we need to sort the points by their distance to the origin.
2. To avoid unnecessary computation, we can compare the squared distances instead of the actual distances.
3. We define a custom sorting key that calculates `x^2 + y^2` for each point, where `x` and `y` are the coordinates of that point.
4. Using the `sort()` function of a list in Python, we sort all points based on their squared distances from the origin.
5. Finally, we return the first `k` elements of the sorted array, as these will be the `k` closest points to the origin.

By applying this approach, we achieve a manageable solution that is easy to implement and understand, using built-in Python functionalities for sorting with a custom key.

Solution Approach

The implementation of the solution makes use of Python's built-in sort function and the concept of lambda functions for the custom sorting key.

1. The lambda function is employed as the sorting key for the `sort()` method. This function takes a point `p` as input and returns `p[0] * p[0] + p[1] * p[1]`. This expression equates to the square of the distance from the point to the origin, omitting the square root for the reasons described earlier.
2. The algorithm:
   - Accepts the list of points and the integer `k`.
   - Applies the `sort()` function on the list of points. Instead of sorting by a single element, it sorts by the value returned by the lambda function for each element, which represents the squared distance of each point to the origin.
   - The sorting process rearranges the items of the list in place, meaning that after the `sort()` method is called, the original list is modified to be in sorted order.
3. After sorting the full list, a slice of the first `k` elements (`points[:k]`) is returned. In Python, slicing a list returns a new list containing the specified range of elements, so in this case, it gives us the closest `k` points.
4. The use of the `sort()` function and a slicing operation makes the solution concise and effective. The Python `sort()` method is an efficient, typically O(n log n) algorithm for sorting lists.
5. The space complexity is kept to O(1) since the sorting is done in place and only the resulting list slice of size `k` is returned.

In conclusion, the solution leverages the efficiency of Python's sorting algorithm and a concise expression of the distance calculation to produce a clean and efficient algorithm for finding the `k` closest points to the origin.

Example Walkthrough

Let's illustrate the solution approach with a simple example. Imagine we have the following points and we are tasked with finding the `k = 2` closest points to the origin `(0, 0)`:

```
points = [[1, 3], [2, -2], [5, 4], [-3, 3]]
```

Following the thought process behind the solution:

1. We calculate the squared distance from the origin for each point which gives us:
   - For point `[1, 3]`: `1^2 + 3^2 = 1 + 9 = 10`
   - For point `[2, -2]`: `2^2 + (-2)^2 = 4 + 4 = 8`
   - For point `[5, 4]`: `5^2 + 4^2 = 25 + 16 = 41`
   - For point `[-3, 3]`: `(-3)^2 + 3^2 = 9 + 9 = 18`
2. We sort the points based on their squared distances, without computing the square root:

   ```
   Sorted points based on squared distances: [[2, -2], [1, 3], [-3, 3], [5, 4]]
   Sorted squared distances: [8, 10, 18, 41]
   ```

3. We employ a lambda function in Python to serve as a custom sorting key: `lambda p: p[0]*p[0] + p[1]*p[1]`.
4. We apply this lambda function within the `sort()` method to our `points` list to rearrange the list based on the calculated squared distances:

   ```
   points.sort(key=lambda p: p[0]*p[0] + p[1]*p[1])
   ```

5. After sorting, `points` looks like this: `[[2, -2], [1, 3], [-3, 3], [5, 4]]`.
6. We return the first `k` elements of the sorted array to get the `k` closest points. Since `k = 2`, we return the first two points:

   ```
   closest_points = points[:k]  # closest_points = [[2, -2], [1, 3]]
   ```

With this example, we have effectively walked through the solution approach described in the problem content. We sorted the points by their squared distance from the origin, avoided unnecessary square root computation, and returned the `k` closest points by slicing the sorted array without altering the original array's order.

Python Solution

```python
from typing import List

class Solution:
    def kClosest(self, points: List[List[int]], k: int) -> List[List[int]]:
        # Sort the given list of points by their Euclidean distance
        # from the origin (0, 0) without actually computing the square root.
        # The lambda function computes the squared distance.
        points.sort(key=lambda point: point[0]**2 + point[1]**2)

        # Return the first k points from the sorted list.
        return points[:k]
```

Java Solution

```java
import java.util.Arrays;

class Solution {
    // This method finds the k closest points to the origin
    public int[][] kClosest(int[][] points, int k) {
        // Sort the array of points based on their Euclidean distance from the origin
        Arrays.sort(points, (point1, point2) -> {
            // Calculate the squared distance for the first point from the origin
            int distance1 = point1[0] * point1[0] + point1[1] * point1[1];
            // Calculate the squared distance for the second point from the origin
            int distance2 = point2[0] * point2[0] + point2[1] * point2[1];
            // Compare the two distances
            return distance1 - distance2;
        });

        // Return the first k elements of the sorted array, which are the k closest to the origin
        return Arrays.copyOfRange(points, 0, k);
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Method to find the 'k' closest points to the origin (0, 0) in a 2D plane
    std::vector<std::vector<int>> kClosest(std::vector<std::vector<int>>& points, int k) {
        // Sort the 'points' array using a custom comparator
        std::sort(points.begin(), points.end(), [](const std::vector<int>& pointA, const std::vector<int>& pointB) {
            // Calculate the squared Euclidean distance from the origin for pointA
            int distA = pointA[0] * pointA[0] + pointA[1] * pointA[1];
            // Calculate the squared Euclidean distance from the origin for pointB
            int distB = pointB[0] * pointB[0] + pointB[1] * pointB[1];
            // Compare the squared distances to sort the points by distance
            return distA < distB;
        });

        // Construct a vector containing the first 'k' elements from the sorted 'points' array
        return std::vector<std::vector<int>>(points.begin(), points.begin() + k);
    }
};
```

Typescript Solution

```typescript
function kClosest(points: number[][], k: number): number[][] {
    // A comparison function for the .sort() method, which will sort the points
    // based on their distance squared from the origin in ascending order.
    const compareDistance = (pointA: number[], pointB: number[]) => {
        // Calculate the distance squared of pointA from the origin.
        const distanceA = pointA[0] ** 2 + pointA[1] ** 2;
        // Calculate the distance squared of pointB from the origin.
        const distanceB = pointB[0] ** 2 + pointB[1] ** 2;
        // The return value determines the order of sorting.
        return distanceA - distanceB;
    };

    // Sort the array of points with the custom compare function.
    const sortedPoints = points.sort(compareDistance);

    // Return the first 'k' elements from the sorted array of points.
    return sortedPoints.slice(0, k);
}
```

Time and Space Complexity

Time Complexity: The time complexity of the code is `O(n log n)`, where `n` is the number of points. This arises from the use of the `.sort()` method, which has `O(n log n)` complexity for sorting the list of points based on their distance from the origin calculated by the key function (`lambda p: p[0] * p[0] + p[1] * p[1]`).

Space Complexity: The space complexity of the code is `O(n)`. While the sorting is done in-place and does not require additional space proportional to the input, the sorted list of points is stored in the same space that was taken by the input list. Hence, the space complexity is linear with respect to the size of the input list of points.