Problem 973: K Closest Points to Origin

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Given an array of

points

where

points[i] = [x

i

, y

i

]

represents a point on the

X-Y

plane and an integer

k

, return the

k closest points to the origin $(0, 0)$.

The distance between two points on the X-Y plane is the Euclidean distance (i.e., $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$).

You may return the answer in

any order

. The answer is

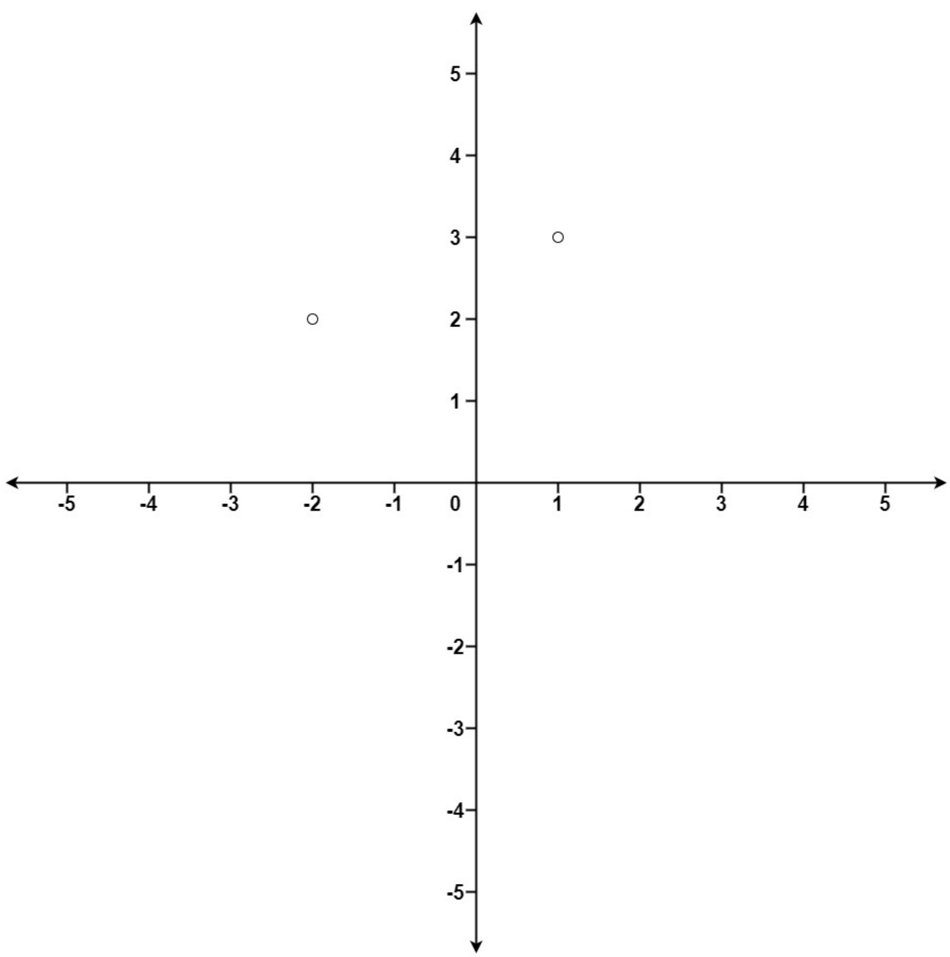
guaranteed

to be

unique

(except for the order that it is in).

Example 1:



Input:

points = [[1,3],[-2,2]], k = 1

Output:

[[-2,2]]

Explanation:

The distance between (1, 3) and the origin is sqrt(10). The distance between (-2, 2) and the origin is sqrt(8). Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin. We only want the closest k = 1 points from the origin, so the answer is just [[-2,2]].

Example 2:

Input:

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

Output:

[[3,3],[-2,4]]

Explanation:

The answer [[-2,4],[3,3]] would also be accepted.

Constraints:

1 <= k <= points.length <= 10

4

-10

4

<= x

i

, y

i

<= 10

4

Code Snippets

C++:

```
class Solution {
public:
vector<vector<int>> kClosest(vector<vector<int>>& points, int k) {

}
};
```

Java:

```
class Solution {
public int[][] kClosest(int[][] points, int k) {

}
}
```

Python3:

```
class Solution:
def kClosest(self, points: List[List[int]], k: int) -> List[List[int]]:
```

Python:

```
class Solution(object):
def kClosest(self, points, k):
"""
:type points: List[List[int]]
:type k: int
:rtype: List[List[int]]
```

```
"""
```

JavaScript:

```
/**
* @param {number[][]} points
* @param {number} k
* @return {number[][]}
*/
var kClosest = function(points, k) {

};
```

TypeScript:

```
function kClosest(points: number[][], k: number): number[][] {

};
```

C#:

```
public class Solution {
public int[][] KClosest(int[][] points, int k) {

}
}
```

C:

```
/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** kClosest(int** points, int pointsSize, int* pointsColSize, int k, int*
returnSize, int** returnColumnSizes) {

}
```

Go:

```
func kClosest(points [][]int, k int) [][]int {

}
```

Kotlin:

```
class Solution {
fun kClosest(points: Array<IntArray>, k: Int): Array<IntArray> {

}
}
```

Swift:

```
class Solution {
func kClosest(_ points: [[Int]], _ k: Int) -> [[Int]] {

}
}
```

Rust:

```
impl Solution {
pub fn k_closest(points: Vec<Vec<i32>>, k: i32) -> Vec<Vec<i32>> {

}
}
```

Ruby:

```
# @param {Integer[][]} points
# @param {Integer} k
# @return {Integer[][]}
def k_closest(points, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $points
```

```
* @param Integer $k
* @return Integer[][]
*/
function kClosest($points, $k) {

}
}
```

Dart:

```
class Solution {
List<List<int>> kClosest(List<List<int>> points, int k) {

}
}
```

Scala:

```
object Solution {
def kClosest(points: Array[Array[Int]], k: Int): Array[Array[Int]] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec k_closest(points :: [[integer]], k :: integer) :: [[integer]]
def k_closest(points, k) do

end
end
```

Erlang:

```
-spec k_closest(Points :: [[integer()]], K :: integer()) -> [[integer()]].
k_closest(Points, K) ->
.
```

Racket:

```
(define/contract (k-closest points k)
(-> (listof (listof exact-integer?)) exact-integer? (listof (listof
exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: K Closest Points to Origin
* Difficulty: Medium
* Tags: array, math, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
vector<vector<int>> kClosest(vector<vector<int>>& points, int k) {

}
};
```

Java Solution:

```
/**
* Problem: K Closest Points to Origin
* Difficulty: Medium
* Tags: array, math, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public int[][] kClosest(int[][] points, int k) {
```

```
}
}
```

Python3 Solution:

```
"""
Problem: K Closest Points to Origin
Difficulty: Medium
Tags: array, math, sort, queue, heap

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def kClosest(self, points: List[List[int]], k: int) -> List[List[int]]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def kClosest(self, points, k):
"""
:type points: List[List[int]]
:type k: int
:rtype: List[List[int]]
"""
```

JavaScript Solution:

```
/**
* Problem: K Closest Points to Origin
* Difficulty: Medium
* Tags: array, math, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
/**
* @param {number[][]} points
* @param {number} k
* @return {number[][]}
*/
var kClosest = function(points, k) {

};
```

TypeScript Solution:

```
/**
* Problem: K Closest Points to Origin
* Difficulty: Medium
* Tags: array, math, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function kClosest(points: number[][], k: number): number[][] {

};
```

C# Solution:

```
/*
* Problem: K Closest Points to Origin
* Difficulty: Medium
* Tags: array, math, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int[][] KClosest(int[][] points, int k) {
```

```
}
}
```

C Solution:

```
/*
* Problem: K Closest Points to Origin
* Difficulty: Medium
* Tags: array, math, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** kClosest(int** points, int pointsSize, int* pointsColSize, int k, int*
returnSize, int** returnColumnSizes) {

}
```

Go Solution:

```
// Problem: K Closest Points to Origin
// Difficulty: Medium
// Tags: array, math, sort, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func kClosest(points [][]int, k int) [][]int {

}
```

Kotlin Solution:

```
class Solution {
fun kClosest(points: Array<IntArray>, k: Int): Array<IntArray> {

}
}
```

Swift Solution:

```
class Solution {
func kClosest(_ points: [[Int]], _ k: Int) -> [[Int]] {

}
}
```

Rust Solution:

```
// Problem: K Closest Points to Origin
// Difficulty: Medium
// Tags: array, math, sort, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn k_closest(points: Vec<Vec<i32>>, k: i32) -> Vec<Vec<i32>> {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} points
# @param {Integer} k
# @return {Integer[][]}
def k_closest(points, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $points
* @param Integer $k
* @return Integer[][]
*/
function kClosest($points, $k) {

}
}
```

Dart Solution:

```
class Solution {
List<List<int>> kClosest(List<List<int>> points, int k) {

}
}
```

Scala Solution:

```
object Solution {
def kClosest(points: Array[Array[Int]], k: Int): Array[Array[Int]] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec k_closest(points :: [[integer]], k :: integer) :: [[integer]]
def k_closest(points, k) do

end
end
```

Erlang Solution:

```
-spec k_closest(Points :: [[integer()]], K :: integer()) -> [[integer()]].
k_closest(Points, K) ->
.
```

Racket Solution:

```
(define/contract (k-closest points k)
(-> (listof (listof exact-integer?)) exact-integer? (listof (listof
exact-integer?)))
)
```