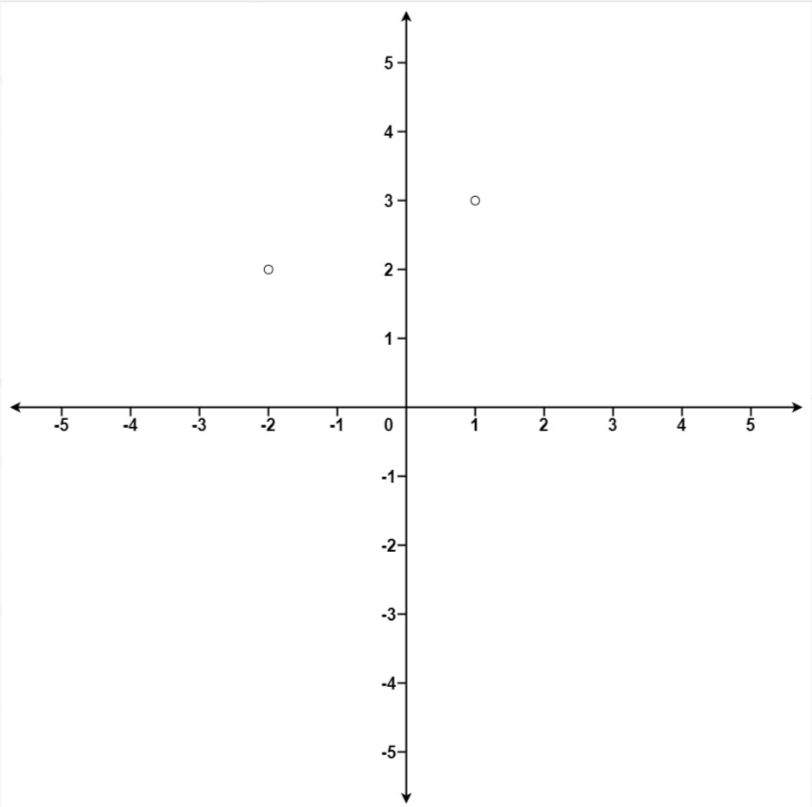
<https://leetcode.com/problems/k-closest-points-to-origin/>

Given an array of points where points[i] = [xi, yi] 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., √(x1 - x2)2 + (y1 - y2)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 <= 104
* -104 <= xi, yi <= 104

**Attempt 1: 2023-10-15**

**Solution 1: Sort Array (10 min)**

class Solution {

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

Arrays.sort(points, (p1, p2) -> (p1[0] \* p1[0] + p1[1] \* p1[1] - p2[0] \* p2[0] - p2[1] \* p2[1]));

int[][] result = new int[k][2];

for(int i = 0; i < k; i++) {

result[i] = points[i];

}

return result;

}

}

Time Complexity: O(NlogN) same as quick sort since Arrays.sort based on Tim Sort which is stable O(NlogN)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/k-closest-points-to-origin/solutions/221532/c-stl-quickselect-priority-queue-and-multiset/>

The simplest solution is to use partial\_sort or nth\_element to order the K closest points at the beginning of points. Here we need a custom comparator to compare the *closeness* of points. This solution is of O(nlogK) time. If we ignore the space of the output since that is inevitable, this solution is of O(1) space.

**Solution 2: Heap (10 min)**

**Style 1: MaxPQ**

class Solution {

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

PriorityQueue<int[]> maxPQ = new PriorityQueue<>((a, b) -> b[0] \* b[0] + b[1] \* b[1] - a[0] \* a[0] - a[1] \* a[1]);

for(int[] p : points) {

maxPQ.offer(p);

if(maxPQ.size() > k) {

maxPQ.poll();

}

}

int[][] result = new int[k][2];

while(k > 0) {

result[k - 1] = maxPQ.poll();

k--;

}

return result;

}

}

Time complexity: O(NlogK)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/k-closest-points-to-origin/solutions/221532/c-stl-quickselect-priority-queue-and-multiset/>

If you would not like to modify points, you may maintain the K closest points so far in a separate data structure. We can use a **max** heap to maintain the K closest points. A max heap has its largest element in the root. Each time we add a point to the heap, if its size exceeds K, we pop the root, which means we get rid of the farthest point and keep the closest ones. This solution is also of O(nlogK) time.

**Style 2: MinPQ**

class Solution {

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

PriorityQueue<int[]> minPQ = new PriorityQueue<>((a, b) -> a[0] \* a[0] + a[1] \* a[1] - b[0] \* b[0] - b[1] \* b[1]);

for(int[] p : points) {

minPQ.offer(p);

}

int[][] result = new int[k][2];

int i = 0;

while(i < k) {

result[i++] = minPQ.poll();

}

return result;

}

}

Time complexity: O(N + NlogK)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/k-closest-points-to-origin/solutions/221532/c-stl-quickselect-priority-queue-and-multiset/>

We can also use a min heap. A min heap has the smallest element in the root. We add all the points to the heap, and then pop the first K ones, we are just the closest ones. This makes the code shorter. Now this one is of O(n + Klogn) time. The n part is on adding all points to the heap (building a min heap for all the points) and the Klogn part is on fetching the top K points from the heap.

**Solution 3: Quick Select (120 min)**

**注意：L973不能直接套用L215的Quick Select的模版，因为L215中只需要找到第k个数，不关心除了第k个数以为的数字的排序，L215中的做法是一种极端简化的Quick Select模版，而L973中的情况就不同，我们除了需要求出第k个数之外还需要知道前面排序后的k - 1个数，所以实际上需要Quick Sort中的Quicksort Partitioning模版，属于复杂的Quick Select形态，因为本质上已经是Quick Sort的接近完全体**

**Style 1: Standard Quick Sort**

class Solution {

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

int n = points.length;

int[][] result = new int[k][2];

quickSort(points, 0, n - 1);

for(int i = 0; i < k; i++) {

result[i] = points[i];

}

return result;

}

private void quickSort(int[][] points, int left, int right) {

// End of recursion reached?

if(left >= right) {

return;

}

int pivotPos = partition(points, left, right);

quickSort(points, left, pivotPos - 1);

quickSort(points, pivotPos + 1, right);

}

private int partition(int[][] points, int left, int right) {

int pivot = distance(points[right]);

int i = left;

int j = right - 1;

while(i < j) {

// Find the first element >= pivot

while(distance(points[i]) < pivot) {

i++;

}

// Find the last element < pivot

while(j > left && distance(points[j]) >= pivot) {

j--;

}

// If the greater element is left of the lesser element, switch them

if(i < j) {

swap(points, i, j);

i++;

j--;

}

}

// i == j means we haven't checked this index yet.

// Move i right if necessary so that i marks the start of the right array.

if(i == j && distance(points[i]) < pivot) {

i++;

}

// Move pivot element to its final position

if(distance(points[i]) != pivot) {

swap(points, i, right);

}

return i;

}

private int distance(int[] p) {

return p[0] \* p[0] + p[1] \* p[1];

}

private void swap(int[][] points, int i, int j) {

int[] tmp = points[i];

points[i] = points[j];

points[j] = tmp;

}

}

Time complexity: O(N) ~ O(N^2)

Space Complexity: O(1)

Theoretically, the average time complexity is O(N) , but just like quick sort, in the worst case, this solution would be degenerated to O(N^2), and practically, the real time it takes on leetcode is 15ms.

**Refer to Quicksort Partitioning**

public class QuicksortSimple {

public void sort(int[] elements) {

quicksort(elements, 0, elements.length - 1);

}

private void quicksort(int[] elements, int left, int right) {

// End of recursion reached?

if (left >= right) {

return;

}

int pivotPos = partition(elements, left, right);

quicksort(elements, left, pivotPos - 1);

quicksort(elements, pivotPos + 1, right);

}

public int partition(int[] elements, int left, int right) {

int pivot = elements[right];

int i = left;

int j = right - 1;

while (i < j) {

// Find the first element >= pivot

while (elements[i] < pivot) {

i++;

}

// Find the last element < pivot

while (j > left && elements[j] >= pivot) {

j--;

}

// If the greater element is left of the lesser element, switch them

if (i < j) {

ArrayUtils.swap(elements, i, j);

i++;

j--;

}

}

// i == j means we haven't checked this index yet.

// Move i right if necessary so that i marks the start of the right array.

if (i == j && elements[i] < pivot) {

i++;

}

// Move pivot element to its final position

if (elements[i] != pivot) {

ArrayUtils.swap(elements, i, right);

}

return i;

}

}

**Style 2: Customized Quick Select (With QuickSort partition + swap elements to sort)**

class Solution {

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

int n = points.length;

// k closest points will be select from sorted array [0, k - 1]

// the kth closest point index is k - 1

quickSelect(points, k - 1, 0, n - 1);

int[][] result = new int[k][2];

for(int i = 0; i < k; i++) {

result[i] = points[i];

}

return result;

}

private void quickSelect(int[][] points, int k, int left, int right) {

if(left >= right) {

return;

}

int pivotIdx = right;

int pivot = distance(points[pivotIdx]);

int i = left;

int j = right - 1;

while(i < j) {

// Find the first element >= pivot

while(distance(points[i]) < pivot) {

i++;

}

// Find the last element < pivot

while(j > left && distance(points[j]) >= pivot) {

j--;

}

// If the greater element is left of the lesser element, switch them

if(i < j) {

swap(points, i, j);

i++;

j--;

}

}

// i == j means we haven't checked this index yet.

// Move i right if necessary so that i marks the start of the right array.

if(i == j && distance(points[i]) < pivot) {

i++;

}

// Move pivot element to its final position

if(distance(points[i]) != pivot) {

swap(points, i, right);

}

// Notice that input k is (k - 1)

// i is now where [pivotIdx] is

if(i > k) {

quickSelect(points, k, left, i - 1); // Continue on left

}

if(i < k) {

quickSelect(points, k, i + 1, right); // Continue on right

}

// i == k, done

}

private int distance(int[] p) {

return p[0] \* p[0] + p[1] \* p[1];

}

private void swap(int[][] points, int i, int j) {

int[] tmp = points[i];

points[i] = points[j];

points[j] = tmp;

}

}

Time complexity: O(N) ~ O(N^2)

Space Complexity: O(1)

Theoretically, the average time complexity is O(N) , but just like quick sort, in the worst case, this solution would be degenerated to O(N^2), and practically, the real time it takes on leetcode is 15ms.

**Refer to**

<https://leetcode.com/problems/k-closest-points-to-origin/solutions/411088/java-solutions-with-exp-comments-sorting-heap-quickselect/>

<https://leetcode.com/problems/k-closest-points-to-origin/solutions/258395/quickselect-thinking-process/>

**和自己的做法区别在于答案里把第一个位置points[lo]设置为pivot，但是自己的做法里将最后一个位置points[right](也就对应这里的points[hi])设置为pivot，另外，实际上并不需要random一个pivot再换到第一个位置，也就是说下面的一段没有必要：**

Random rand = new Random();

int randIdx = lo + rand.nextInt(hi - lo + 1); // lo + (0 ~ #element)

// place the key to the beginning

swap(points, lo, randIdx);

**完整代码**

class Solution {

// quickSelect

public int[][] kClosest(int[][] points, int K) {

int n = points.length;

quickSelect(points, K - 1, 0, n - 1); // index from 0

int[][] result = new int[K][];

for (int i = 0; i < K; ++i) {

result[i] = points[i];

}

return result;

}

// find the k-th element (from 0 ~ hi - 1)

private void quickSelect(int[][] points, int k, int lo, int hi) {

if (lo == hi) {

return;

}

Random rand = new Random();

int randIdx = lo + rand.nextInt(hi - lo + 1); // lo + (0 ~ #element)

// place the key to the beginning

swap(points, lo, randIdx);

int key = lo;

int i = lo, j = hi + 1; // one index offset

while (true) {

while (dis(points[++i]) < dis(points[key])) { // move i

if (i == hi) break;

}

while (dis(points[--j]) > dis(points[key])) { // move j

if (j == lo) break;

}

if (i >= j) break;

swap(points, i, j);

}

swap(points, key, j); // put [key] to the correct place [<key] [key] [>key]

// notice that k = K - 1

// j is now where [key] is

if (j > k) quickSelect(points, k, lo, j - 1); // left

if (j < k) quickSelect(points, k, j + 1, hi); // right

// if j == k, finish.

}

private void swap(int[][] points, int i, int j) {

int[] temp = points[i];

points[i] = points[j];

points[j] = temp;

}

private int dis(int[] point) {

return point[0] \* point[0] + point[1] \* point[1];

}

}

**[Java] Three solutions to this classical K-th problem.**

**Refer to**

<https://leetcode.com/problems/k-closest-points-to-origin/solutions/220235/java-three-solutions-to-this-classical-k-th-problem/>

This is a very classical problem, so-called K-th problem. Here I will share some summaries and some classical solutions to this kind of problem.

**I.** The very naive and simple solution is sorting the all points by their distance to the origin point directly, then get the top k closest points. We can use the sort function and the code is very short.

**Theoretically**, the time complexity is **O(NlogN)**, **practically**, the real time it takes on leetcode is **104ms**.

The **advantages** of this solution are **short**, intuitive and easy to implement.

The **disadvantages** of this solution are not very efficient and have to know all of the points previously, and it is unable to deal with real-time(online) case, it is an **off-line** solution.

The short code shows as follows:

public int[][] kClosest(int[][] points, int K) {

Arrays.sort(points, (p1, p2) -> p1[0] \* p1[0] + p1[1] \* p1[1] - p2[0] \* p2[0] - p2[1] \* p2[1]);

return Arrays.copyOfRange(points, 0, K);

}

**II.** The second solution is based on the first one. We don't have to sort all points. Instead, we can maintain a **max-heap** with size K. Then for each point, we add it to the heap. Once the size of the heap is greater than K, we are supposed to extract one from the max heap to ensure the size of the heap is always K. Thus, the max heap is always maintain top K smallest elements from the first one to current one. Once the size of the heap is over its maximum capacity, it will exclude the maximum element in it, since it can not be the proper candidate anymore.

**Theoretically**, the time complexity is **O(NlogK)**, but **practically**, the real time it takes on leetcode is **134ms**.

The **advantage** of this solution is it can deal with **real-time(online) stream data**. It does not have to know the size of the data previously.

The **disadvantage** of this solution is it is not the most efficient solution.

The short code shows as follows:

public int[][] kClosest(int[][] points, int K) {

PriorityQueue<int[]> pq = new PriorityQueue<int[]>((p1, p2) -> p2[0] \* p2[0] + p2[1] \* p2[1] - p1[0] \* p1[0] - p1[1] \* p1[1]);

for (int[] p : points) {

pq.offer(p);

if (pq.size() > K) {

pq.poll();

}

}

int[][] res = new int[K][2];

while (K > 0) {

res[--K] = pq.poll();

}

return res;

}

**III.** The last solution is based on quick sort, we can also call it **quick select**. In the quick sort, we will always choose a pivot to compare with other elements. After one iteration, we will get an array that all elements smaller than the pivot are on the left side of the pivot and all elements greater than the pivot are on the right side of the pviot (assuming we sort the array in ascending order). So, inspired from this, each iteration, we choose a pivot and then find the position **p** the pivot should be. Then we compare **p** with the **K**, if the **p** is smaller than the **K**, meaning the all element on the left of the pivot are all proper candidates but it is not adequate, we have to do the same thing on right side, and vice versa. If the **p** is exactly equal to the **K**, meaning that we've found the K-th position. Therefore, we just return the first K elements, since they are not greater than the pivot.

**Theoretically**, the average time complexity is **O(N)** , but just like quick sort, in the worst case, this solution would be degenerated to **O(N^2)**, and **practically**, the real time it takes on leetcode is **15ms**.

The **advantage** of this solution is it is very efficient.

The **disadvantage** of this solution are it is neither an online solution nor a stable one. And the K elements closest are **not sorted** in ascending order.

The short code shows as follows:

public int[][] kClosest(int[][] points, int K) {

int len = points.length, l = 0, r = len - 1;

while (l <= r) {

int mid = helper(points, l, r);

if (mid == K) break;

if (mid < K) {

l = mid + 1;

} else {

r = mid - 1;

}

}

return Arrays.copyOfRange(points, 0, K);

}

private int helper(int[][] A, int l, int r) {

int[] pivot = A[l];

while (l < r) {

while (l < r && compare(A[r], pivot) >= 0) r--;

A[l] = A[r];

while (l < r && compare(A[l], pivot) <= 0) l++;

A[r] = A[l];

}

A[l] = pivot;

return l;

}

private int compare(int[] p1, int[] p2) {

return p1[0] \* p1[0] + p1[1] \* p1[1] - p2[0] \* p2[0] - p2[1] \* p2[1];

}

**The average time complexity of quick sort is O(nlgn), so why third solution has O(n) time complexity?**

**Refer to**

<https://leetcode.com/problems/k-closest-points-to-origin/solutions/220235/java-three-solutions-to-this-classical-k-th-problem/comments/262781>

**Because in the Quick Sort, you have to take care of two sides of the pivot. But in Quick Select, you only focus on the side the target object should be. So, in optimal case, the running time of Quick Sort is (n + 2 \* (n/2) + 4 \* (n/4)...), it has logn iterations. Instead, the running time of Quick Select would be (n + n/2 + n/4 +...) it has logn iterations as well. Therefore, the running time of Quick Sort is O(nlogn), instead, the running time of Quick Select is O(n)**

<https://leetcode.com/problems/k-closest-points-to-origin/solutions/220235/java-three-solutions-to-this-classical-k-th-problem/comments/300485>

**The difference located in the recursion part. In quick sort, we need to do recursion in each part, but in quick select, we can throw the one of the part and do the recursion in the rest of the part. Therefore, the time complexity of quick select is T(t) = 2*T(t) - T(t) = c*(2n + n + n/2 + n/4 + ... + 2) - c \*(n + n/2 + n/4 + n/8 + ... + 1) = 2cn -1 = O(n)**

<https://leetcode.com/problems/k-closest-points-to-origin/solutions/220235/java-three-solutions-to-this-classical-k-th-problem/comments/315591>

**Average case: pivot is generally near the mid of l and r; therefore we have n + (n/2 + n/4 +...) <= n + (n) = 2 \* n = O(n).**

**As for worst case: K = n, and points is already sorted by the distance to origin. @Frimish's code will cost n - 1 + ... + 1 = (n - 1) \* n / 2 = O(n^2) time.**