# 题目

我们有一个由平面上的点组成的列表 points。需要从中找出 K 个距离原点 (0, 0) 最近的点。

（这里，平面上两点之间的距离是欧几里德距离。）

你可以按任何顺序返回答案。除了点坐标的顺序之外，答案确保是唯一的。

**示例 1：**

**输入：**points = [[1,3],[-2,2]], K = 1

**输出：**[[-2,2]]

**解释：**

(1, 3) 和原点之间的距离为 sqrt(10)，

(-2, 2) 和原点之间的距离为 sqrt(8)，

由于 sqrt(8) < sqrt(10)，(-2, 2) 离原点更近。

我们只需要距离原点最近的 K = 1 个点，所以答案就是 [[-2,2]]。

**示例 2：**

**输入：**points = [[3,3],[5,-1],[-2,4]], K = 2

**输出：**[[3,3],[-2,4]]

（答案 [[-2,4],[3,3]] 也会被接受。）

**提示：**

1 <= K <= points.length <= 10000

-10000 < points[i][0] < 10000

-10000 < points[i][1] < 10000

# 分析

## 方法一：排序

**思路：**

将每个点到原点的欧几里得距离的平方从小到大排序后，取出前k个即可。

**代码：**

class Solution {

public:

vector<vector<int>> kClosest(vector<vector<int>>& points, int k) {

sort(points.begin(), points.end(), [](const vector<int>& u, const vector<int>& v) {

return u[0] \* u[0] + u[1] \* u[1] < v[0] \* v[0] + v[1] \* v[1];

});

return {points.begin(), points.begin() + k};

}

};

**复杂度分析：**

时间复杂度：O(nlogn)，其中n是数组points的长度。算法的时间复杂度即排序的时间复杂度。

空间复杂度：O(logn)，排序所需额外的空间复杂度为O(logn)。

## 方法二：堆/优先队列

**思路（LeetCode）：**

我们可以使用一个大根堆实时维护前k个最小的距离平方。

首先我们将前k个点的编号（为了方便最后直接得到答案）以及对应的距离平方放入大根堆中，随后从第 k+1个点开始遍历：如果当前点的距离平方比堆顶的点的距离平方要小，就把堆顶的点弹出，再插入当前的点。当遍历完成后，所有在大根堆中的点就是前 k个距离最小的点。

不同的语言提供的堆的默认情况不一定相同。在 C++ 语言中，堆（即优先队列）为大根堆，但在 Python 语言中，堆为小根堆，因此我们需要在小根堆中存储（以及比较）距离平方的相反数。

**代码：**

class Solution {

public:

struct cmp {

bool operator () (vector<int> & a, vector<int> & b) {

//不是operator，这是个函数

return pow(long(a[0]),2)+pow(long(a[1]),2) <

pow(long(b[0]),2)+pow(long(b[1]),2);

//该函数作用即如果a的平方和小于b的平方和返回true，否则false

}

}; //结构体后面需要有“;”

vector<vector<int>> kClosest(vector<vector<int>>& points, int K) {

vector<vector<int>> re;

if (points.empty()) return re;

priority\_queue<vector<int>, vector<vector<int>>, cmp> que;

for (int i=0; i<points.size(); i++) {

que.push(points[i]);

//入堆的时候自动根据compare函数确定是否插入

if (que.size()>K) que.pop();

}

while (!que.empty()) {

re.push\_back(que.top());

que.pop();

}

return re;

}

};

另一种写法：

class Solution {

public:

struct compare{

bool operator() (vector<int> &a, vector<int> &b){

return pow(long(a[0]),2)+pow(long(a[1]),2) < pow(long(b[0]),2)+pow(long(b[1]),2);

}

};

public:

vector<vector<int>> kClosest(vector<vector<int>>& points, int K) {

std::priority\_queue<vector<int>,vector<vector<int>>,compare> pri\_queue;

vector<vector<int>> tmpArr;

if(points.empty())

return tmpArr;

int i=0;

for(;i<points.size();i++)

{

pri\_queue.push(points.at(i));

if(i>=K)

{

pri\_queue.pop();

}

}

while(!pri\_queue.empty())

{

tmpArr.push\_back(pri\_queue.top());

pri\_queue.pop();

}

return tmpArr;

}

};

Leetcode官方写法：

class Solution {

public:

vector<vector<int>> kClosest(vector<vector<int>>& points, int k) {

priority\_queue<pair<int, int>> q;

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

q.emplace(points[i][0] \* points[i][0] + points[i][1] \* points[i][1], i);

}

int n = points.size();

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

int dist = points[i][0] \* points[i][0] + points[i][1] \* points[i][1];

if (dist < q.top().first) {

q.pop();

q.emplace(dist, i);

}

}

vector<vector<int>> ans;

while (!q.empty()) {

ans.push\_back(points[q.top().second]);

q.pop();

}

return ans;

}

};

**复杂度分析：**

时间复杂度：O(nlogk)，其中n是数组points 的长度。由于大根堆维护的是前k个距离最小的点，因此弹出和插入操作的单次时间复杂度均为O(logk)。在最坏情况下，数组里n个点都会插入，因此时间复杂度为O(nlogk)。

空间复杂度：O(k)，因为大根堆中最多有k个点。