**K Closest Points to Origin**

Question

We have a list of points on the plane.  Find the K closest points to the origin (0, 0).

(Here, the distance between two points on a plane is the Euclidean distance.)

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]]

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

**Note:**

1. 1 <= K <= points.length <= 10000
2. -10000 < points[i][0] < 10000
3. -10000 < points[i][1] < 10000

# **Solution**

#### **Approach 1: Greedy**

**Intuition**

At least one worker will be paid their minimum wage expectation. If not, we could scale all payments down by some factor and still keep everyone earning more than their wage expectation.

**Algorithm**

For each captain worker that will be paid their minimum wage expectation, let's calculate the cost of hiring K workers where each point of quality is worth wage[captain] / quality[captain] dollars. With this approach, the remaining implementation is straightforward.

Note that this algorithm would not be efficient enough to pass larger test cases.

#### Coding Solution

Java

|  |
| --- |
| class Solution {  public double mincostToHireWorkers(int[] quality, int[] wage, int K) {  int N = quality.length;  double ans = 1e9;  for (int captain = 0; captain < N; ++captain) {  // Must pay at least wage[captain] / quality[captain] per qual  double factor = (double) wage[captain] / quality[captain];  double prices[] = new double[N];  int t = 0;  for (int worker = 0; worker < N; ++worker) {  double price = factor \* quality[worker];  if (price < wage[worker]) continue;  prices[t++] = price;  }  if (t < K) continue;  Arrays.sort(prices, 0, t);  double cand = 0;  for (int i = 0; i < K; ++i)  cand += prices[i];  ans = Math.min(ans, cand);  }  return ans;  }  } |

Python

|  |
| --- |
| class Solution(object):  def mincostToHireWorkers(self, quality, wage, K):  from fractions import Fraction  ans = float('inf')  N = len(quality)  for captain in xrange(N):  # Must pay at least wage[captain] / quality[captain] per qual  factor = Fraction(wage[captain], quality[captain])  prices = []  for worker in xrange(N):  price = factor \* quality[worker]  if price < wage[worker]: continue  prices.append(price)  if len(prices) < K: continue  prices.sort()  ans = min(ans, sum(prices[:K]))  return float(ans) |

**Complexity Analysis**

* Time Complexity: O(N^2 \log N)*O*(*N*2log*N*), where N*N* is the number of workers.
* Space Complexity: O(N)*O*(*N*).

#### **Approach 2: Heap**

**Intuition**

As in Approach #1, at least one worker is paid their minimum wage expectation.

Additionally, every worker has some minimum ratio of dollars to quality that they demand. For example, if wage[0] = 100 and quality[0] = 20, then the ratio for worker 0 is 5.0.

The key insight is to iterate over the ratio. Let's say we hire workers with a ratio R or lower. Then, we would want to know the K workers with the lowest quality, and the sum of that quality. We can use a heap to maintain these variables.

**Algorithm**

Maintain a max heap of quality. (We're using a minheap, with negative values.) We'll also maintain sumq, the sum of this heap.

For each worker in order of ratio, we know all currently considered workers have lower ratio. (This worker will be the 'captain', as described in Approach #1.) We calculate the candidate answer as this ratio times the sum of the smallest K workers in quality.

Coding Solution

Java

|  |
| --- |
| class Solution {  public double mincostToHireWorkers(int[] quality, int[] wage, int K) {  int N = quality.length;  Worker[] workers = new Worker[N];  for (int i = 0; i < N; ++i)  workers[i] = new Worker(quality[i], wage[i]);  Arrays.sort(workers);  double ans = 1e9;  int sumq = 0;  PriorityQueue<Integer> pool = new PriorityQueue();  for (Worker worker: workers) {  pool.offer(-worker.quality);  sumq += worker.quality;  if (pool.size() > K)  sumq += pool.poll();  if (pool.size() == K)  ans = Math.min(ans, sumq \* worker.ratio());  }  return ans;  }  }  class Worker implements Comparable<Worker> {  public int quality, wage;  public Worker(int q, int w) {  quality = q;  wage = w;  }  public double ratio() {  return (double) wage / quality;  }  public int compareTo(Worker other) {  return Double.compare(ratio(), other.ratio());  }  } |

Python

|  |
| --- |
| class Solution(object):  def mincostToHireWorkers(self, quality, wage, K):  from fractions import Fraction  workers = sorted((Fraction(w, q), q, w)  for q, w in zip(quality, wage))  ans = float('inf')  pool = []  sumq = 0  for ratio, q, w in workers:  heapq.heappush(pool, -q)  sumq += q  if len(pool) > K:  sumq += heapq.heappop(pool)  if len(pool) == K:  ans = min(ans, ratio \* sumq)  return float(ans) |

**Complexity Analysis**

* Time Complexity: O(N \log N)*O*(*N*log*N*), where N*N* is the number of workers.
* Space Complexity: O(N)*O*(*N*).