**Minimum Cost to Hire K Workers**

There are n workers. You are given two integer arrays quality and wage where quality[i] is the quality of the ith worker and wage[i] is the minimum wage expectation for the ith worker.

We want to hire exactly k workers to form a paid group. To hire a group of k workers, we must pay them according to the following rules:

1. Every worker in the paid group should be paid in the ratio of their quality compared to other workers in the paid group.
2. Every worker in the paid group must be paid at least their minimum wage expectation.

Given the integer k, return *the least amount of money needed to form a paid group satisfying the above conditions*. Answers within 10-5 of the actual answer will be accepted.

**Example 1:**

**Input:** quality = [10,20,5], wage = [70,50,30], k = 2

**Output:** 105.00000

**Explanation:** We pay 70 to 0th worker and 35 to 2nd worker.

**Example 2:**

**Input:** quality = [3,1,10,10,1], wage = [4,8,2,2,7], k = 3

**Output:** 30.66667

**Explanation:** We pay 4 to 0th worker, 13.33333 to 2nd and 3rd workers separately.

**Constraints:**

* n == quality.length == wage.length
* 1 <= k <= n <= 104
* 1 <= quality[i], wage[i] <= 104

/\*\*

\* @param {number[]} quality

\* @param {number[]} wage

\* @param {number} k

\* @return {number}

\*/

var mincostToHireWorkers = function(quality, wage, k) {

};

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.

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;

}

}

**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.

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());

}

}

**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*).