<https://leetcode.com/problems/minimum-cost-to-hire-k-workers/description/>

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:

Every worker in the paid group should be paid in the ratio of their quality compared to other workers in the paid group.

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 <= 10^4

1 <= quality[i], wage[i] <= 10^4

**Attempt 1: 2023-03-16**

**Solution 1: Priority Queue (60 min)**

**Wrong Solution**

**Test out by**

Input

quality = [3,1,10,10,1]

wage = [4,8,2,2,7]

k = 3

Output = 23.00000

Expected = 30.66667

**Style 1: Pure array, NO 'Worker' class**

class Solution {

    public double mincostToHireWorkers(int[] quality, int[] wage, int k) {

        // Use a max heap to keep track of the highest quality workers.

        PriorityQueue<Integer> maxPQ = new PriorityQueue<>((a, b) -> b - a);

        int n = quality.length;

        // pair -> {quality, ratio(wage/quality)}

        double[][] pair = new double[n][2];

        for(int i = 0; i < n; i++) {

            pair[i] = new double[]{(double)quality[i], (double)(wage[i] / quality[i])};

        }

        // Sort by ratio(wage/quality) in ascending order

        // Different than L1383 or L2542 which sort in descending order

        Arrays.sort(pair, (a, b) -> Double.compare(a[1], b[1]));

        int qualitySum = 0;

        double result = Double.MAX\_VALUE;

        for(int i = 0; i < n; i++) {

            qualitySum += pair[i][0];

            maxPQ.offer((int)pair[i][0]);

            if(maxPQ.size() == k) {

                result = Math.min(result, qualitySum \* pair[i][1]);

                qualitySum -= maxPQ.poll();

            }

        }

        return result;

    }

}

**The result of "pair[i] = new double[]{(double)quality[i], (double)(wage[i] / quality[i])}" is wrong**

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

    pair[i] = new double[]{(double)quality[i], (double)(wage[i] / quality[i])};

}

**Style 2: With 'Worker' class**

class Solution {

    class Worker {

        int quality;

        double ratio;

        public Worker(int quality, int wage) {

            this.quality = quality;

// Below line error out -> should change to

// this.ratio = (double) wage / quality;

            this.ratio = (double) (wage / quality);

        }

    }

    public double mincostToHireWorkers(int[] quality, int[] wage, int k) {

        // Use a max heap to keep track of the highest quality workers.

        PriorityQueue<Integer> maxPQ = new PriorityQueue<>((a, b) -> b - a);

        int n = quality.length;

        // pair -> {quality, ratio(wage/quality)}

        Worker[] workers = new Worker[n];

        for(int i = 0; i < n; i++) {

            workers[i] = new Worker(quality[i], wage[i]);

        }

        // Sort by ratio(wage/quality) in ascending order

        // Different than L1383 or L2542 which sort in descending order

        Arrays.sort(workers, (a, b) -> Double.compare(a.ratio, b.ratio));

        int qualitySum = 0;

        double result = Double.MAX\_VALUE;

        for(int i = 0; i < n; i++) {

            qualitySum += workers[i].quality;

            maxPQ.offer(workers[i].quality);

            if(maxPQ.size() == k) {

                result = Math.min(result, qualitySum \* workers[i].ratio);

                qualitySum -= maxPQ.poll();

            }

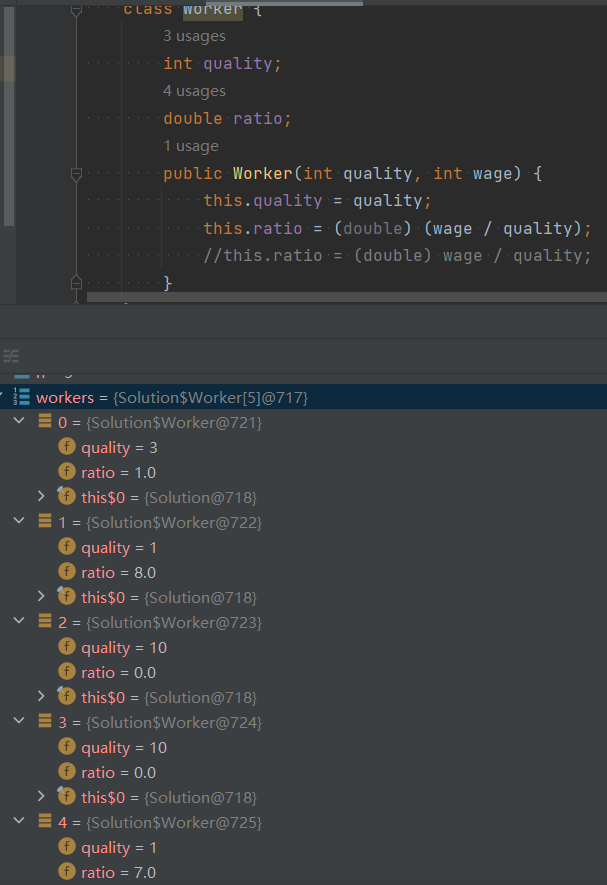
        }

        return result;

    }

}

**The result of "this.ratio = (double) (wage / quality)" is wrong**



**Correct Solution**

**Style 1: Pure array, NO 'Worker' class**

**Only one line change:**

**pair[i] = new double[]{(double)quality[i], (double)(wage[i] / quality[i])} -> pair[i] = new double[]{(double)quality[i], (double)wage[i] / quality[i]}**

class Solution {

    public double mincostToHireWorkers(int[] quality, int[] wage, int k) {

        // Use a max heap to keep track of the highest quality workers.

        PriorityQueue<Integer> maxPQ = new PriorityQueue<>((a, b) -> b - a);

        int n = quality.length;

        // pair -> {quality, ratio(wage/quality)}

        double[][] pair = new double[n][2];

        for(int i = 0; i < n; i++) {

            pair[i] = new double[]{(double)quality[i], (double)wage[i] / quality[i]};

        }

        // Sort by ratio(wage/quality) in ascending order

        // Different than L1383 or L2542 which sort in descending order

        Arrays.sort(pair, (a, b) -> Double.compare(a[1], b[1]));

        int qualitySum = 0;

        double result = Double.MAX\_VALUE;

        for(int i = 0; i < n; i++) {

            qualitySum += pair[i][0];

            maxPQ.offer((int)pair[i][0]);

            if(maxPQ.size() == k) {

                result = Math.min(result, qualitySum \* pair[i][1]);

                qualitySum -= maxPQ.poll();

            }

        }

        return result;

    }

}

Time Complexity: O(n log n + n log k)

Space Complexity: O(n + k)

**Style 2: With 'Worker' class**

**Only one line change:**

**"this.ratio = (double) (wage / quality)" -> "this.ratio = (double) wage / quality"**

class Solution {

    class Worker {

        int quality;

        double ratio;

        public Worker(int quality, int wage) {

            this.quality = quality;

            //this.ratio = (double) (wage / quality);

            this.ratio = (double) wage / quality;

        }

    }

    public double mincostToHireWorkers(int[] quality, int[] wage, int k) {

        // Use a max heap to keep track of the highest quality workers.

        PriorityQueue<Integer> maxPQ = new PriorityQueue<>((a, b) -> b - a);

        int n = quality.length;

        // pair -> {quality, ratio(wage/quality)}

        Worker[] workers = new Worker[n];

        for(int i = 0; i < n; i++) {

            workers[i] = new Worker(quality[i], wage[i]);

        }

        // Sort by ratio(wage/quality) in ascending order

        // Different than L1383 or L2542 which sort in descending order

        Arrays.sort(workers, (a, b) -> Double.compare(a.ratio, b.ratio));

        int qualitySum = 0;

        double result = Double.MAX\_VALUE;

        for(int i = 0; i < n; i++) {

            qualitySum += workers[i].quality;

            maxPQ.offer(workers[i].quality);

            if(maxPQ.size() == k) {

                result = Math.min(result, qualitySum \* workers[i].ratio);

                qualitySum -= maxPQ.poll();

            }

        }

        return result;

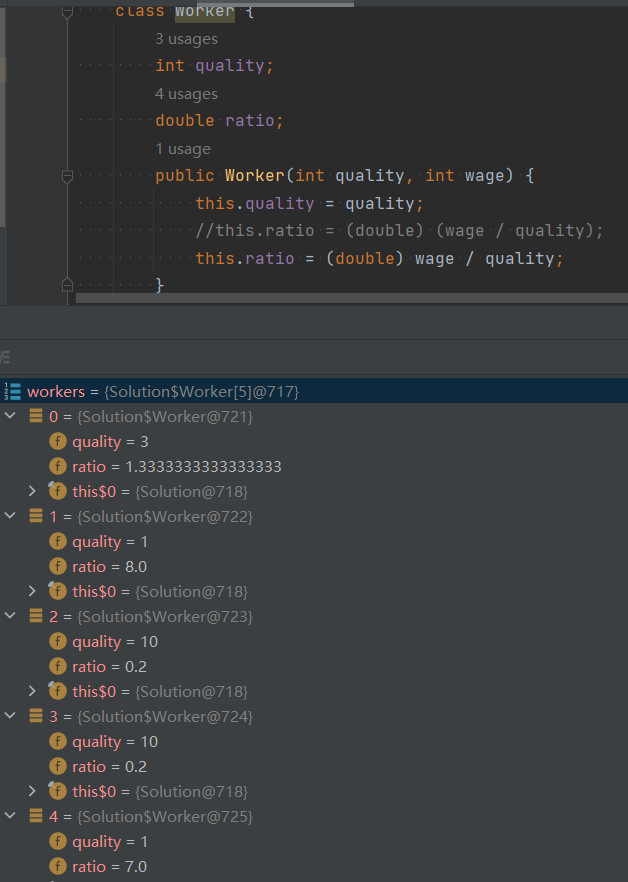
    }

}

Time Complexity: O(n log n + n log k)

Space Complexity: O(n + k)

**The result of "this.ratio = (double) wage / quality" is correct**



**Refer to**

<https://leetcode.com/problems/minimum-cost-to-hire-k-workers/solutions/141768/detailed-explanation-o-nlogn/>

Let's read description first and figure out the two 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."**

So for any two workers in the paid group,

we have wage[i] : wage[j] = quality[i] : quality[j]

So we have wage[i] : quality[i] = wage[j] : quality[j]

We pay wage to every worker in the group with the same ratio compared to his own quality.

**"2. Every worker in the paid group must be paid at least their minimum wage expectation."**

**For every worker, he has an expected ratio of wage compared to his quality.**

So to minimize the total wage, we want a small ratio.

So we sort all workers with their expected ratio, and pick up K first worker.

Now we have a minimum possible ratio for K worker and we their total quality.

As we pick up next worker with bigger ratio, we increase the ratio for whole group.

Meanwhile we remove a worker with highest quality so that we keep K workers in the group.

We calculate the current ratio \* total quality = total wage for this group.

We redo the process and we can find the minimum total wage.

Because workers are sorted by ratio of wage/quality.

For every ratio, we find the minimum possible total quality of K workers.

**Time Complexity**

O(NlogN) for sort.

O(NlogK) for priority queue.

public double mincostToHireWorkers(int[] q, int[] w, int K) {

double[][] workers = new double[q.length][2];

for (int i = 0; i < q.length; ++i)

workers[i] = new double[]{(double)(w[i]) / q[i], (double)q[i]};

Arrays.sort(workers, (a, b) -> Double.compare(a[0], b[0]));

double res = Double.MAX\_VALUE, qsum = 0;

PriorityQueue<Double> pq = new PriorityQueue<>();

for (double[] worker: workers) {

qsum += worker[1];

pq.add(-worker[1]);

if (pq.size() > K) qsum += pq.poll();

if (pq.size() == K) res = Math.min(res, qsum \* worker[0]);

}

return res;

}

**FAQ:**

**Question:** "However, it is possible that current worker has the highest quality, so you removed his quality in the last step, which leads to the problem that you are "using his ratio without him".

**Answer:** It doesn't matter. The same group will be calculated earlier with smaller ratio.

And it doesn't obey my logic here: For a given ratio of wage/quality, find minimum total wage of K workers.

<https://leetcode.com/problems/minimum-cost-to-hire-k-workers/solutions/141768/detailed-explanation-o-nlogn/comments/148422>

Just to rephrase the solution the way I understood it:

1- **expect[i] = wage[i]/quality[i]**: meaning i-th worker claims expect[i] money per each unit of its quality. Therefore, if **expect[i] > expect[j]**, that means if we pay j-th worker quality[j]\*expect[i] he/she would be more than happy and it's more than its minimal requested wage.

2- Therefore, for k workers sorted by their expect values, if we pay each worker **q[i]\*expect[k]**, both rules are satisfied. The total needed money = **(sum(q\_1 + q\_2 + ... + q\_k) \* expect[k])**. Note that this is the minimum money for this k workers, since you have to pay the k-th worker at least **q[k]\*expect[k]**.

3- To recap, we sort workers based on their expect values. Say we are at worker i and want to form a k-group and we already know it would cost **sum\*expect[i]**. To pay the minimum money we should minimum the sum, which can be found using a maxHeap (to replace the max value with a smaller one) to keep the smaller **q's** as we move forward.

**Refer to**

<https://algo.monster/liteproblems/857>

**Problem Description**

In this problem, you are responsible for hiring a group of n workers. Each worker has two associated parameters: quality and wage. The quality of a worker represents a numerical value for the worker's skill level, while the wage indicates the minimum salary that the worker expects.

To form a paid group, exactly k workers need to be hired under two constraints:

The payment of each worker must be proportional to their quality relative to the other workers in the group. This means if one worker has twice the quality compared with another worker in the group, they should also be paid twice as much.

No worker can be paid less than their minimum wage expectation.

The goal is to calculate the minimum amount of money required to hire exactly k workers such that both of the aforementioned conditions are satisfied.

**Intuition**

The solution involves the concept of 'efficiency', which in this context is defined as the ratio of wage to quality for each worker. The intuition is to first sort the workers by their efficiency. This allows us to consider the workers in order of how much wage they require per unit quality.

Once sorted, the algorithm uses a min-heap to keep track of the k workers with the largest qualities, since any payment must be at least as large as the k-th worker's quality times the efficiency. By iterating through the sorted workers and maintaining the sum of the largest k qualities, when the k-th worker is added to the heap, the smallest ratio that satisfies all the workers' minimum wage expectations can be calculated. This is done by multiplying the total sum of the chosen qualities by the current worker's efficiency.

During iteration, the algorithm keeps updating the answer to record the least amount of money needed to form a paid group. This is achieved by maintaining a total quality sum and updating it as the group changes. When a new worker is added and the group exceeds k workers, the worker with the highest quality (and therefore the one contributing most to the total wage) is removed from the heap and the total quality sum. This step ensures that the maintained group always has k workers. The answer is updated whenever a potentially lower total wage for the group is found.

The use of a heap allows efficient removal of the worker with the highest quality in O(log k) time and helps to maintain the total quality sum and minimum possible wage for the group at each step.

**Solution Approach**

The implementation of the solution provided in Python involves a sorting step and the use of a min-heap. The overall approach follows an efficient algorithm to minimize the total wage based on the ratio of wage to quality defined as efficiency. Here's a step-by-step explanation of how the solution works:

Sorting: We start by computing the efficiency for each worker, which is the ratio of wage[i] to quality[i]. Then, we sort the workers based on this efficiency in ascending order. This ensures that we consider workers in increasing order of how much they expect to be paid relative to their quality.

Initializing the Heap: We use a min-heap (in Python, a min-heap can be utilized by pushing negative values into it) to keep track of the k largest qualities encountered thus far in the sorted array. A heap is advantageous here because it allows us to efficiently retrieve and remove the worker with the highest quality, which in turn affects the total payment calculation.

Iterative Calculation: Next, we iterate through the sorted workers and do the following:

Add the current worker's quality to the total quality sum (tot).

Push the negative of the current worker's quality to the heap to maintain the k largest qualities.

If the heap size exceeds k, it means we have more than k workers in the current group, so we pop from the heap (which removes the worker with the highest quality from our consideration) and reduce our total quality sum accordingly.

If the heap size is exactly k, we calculate the current cost to hire k workers based on the worker with the current efficiency and update the answer with the minimum cost observed so far. This is done by ans = min(ans, w / q \* tot), where w / q represents the efficiency of the last worker added to meet the group size of k.

Returning the Result: After going through all workers, the variable ans stores the minimum amount of money needed to hire k workers while satisfying the constraints. We return ans as the solution to the problem.

The key patterns used in this solution include sorting based on a custom comparator (efficiency), and utilizing a heap for dynamic subset selection - in this case, maintaining a subset of workers of size k with the greatest quality.

Algorithm Complexity: The time complexity of this algorithm is mainly governed by the sorting step, which is O(n log n), where n is the number of workers. Operating on the heap takes O(log k) time for each insertion and deletion, and since we go through the workers once, the overall complexity for heap operations is O(n log k). Hence, the total time complexity is O(n log n + n log k). The space complexity is O(n) to store the sorted efficiencies and the heap.

**Example Walkthrough**

Let's walk through the solution with a small example.

Suppose we have 4 workers, each with the following quality and wage expectations:

Workers: 1 2 3 4

Quality: 10 20 15 30

Wage: 60 120 90 180

We want to hire k = 2 workers while meeting the conditions specified in the problem.

Step 1: Sorting First, we calculate the efficiency wage[i] / quality[i] for each worker:

Efficiency: 6 6 6 6

Notice that in this case, all workers have the same efficiency, meaning any order keeps them sorted by efficiency. However, this won't always be the case.

Step 2: Initializing the Heap Now we initialize a min-heap to keep track of the largest qualities as we iterate through the workers. Initially, the heap is empty.

Step 3: Iterative Calculation We iterate through the sorted workers, keeping track of the total quality (tot) and potential total wage (ans).

For worker 1, quality = 10. We add it to tot and push -10 to the heap.

For worker 2, quality = 20. We add it to tot, giving us tot = 30, and push -20 to the heap.

The heap size is now equal to k, so we calculate the total payment for these k workers using the current efficiency, which is 6 \* tot = 180. We set ans = 180.

To consider other possible combinations since the heap size now exceeds k, we must pop from the heap:

Since the heap is a min-heap with negatives, when we pop, we're removing the worker with the highest quality, in this case -10 (worker 1), leaving the heap with -20 and tot = 20.

We then move to worker 3, with quality = 15. Adding to tot gives us 35, and we push -15 to the heap.

The heap size again exceeds k, so we pop the largest quality, which is now -20 (worker 2), and tot becomes 15 (as we keep worker 3 and the new worker 4 in our consideration).

We must update ans if it gets lower. Worker 4 gets added with an efficiency of 6 and quality = 30. The new tot is 45, and ans becomes min(180, 6 \* 45) = 270.

We proceed like this until all workers have been considered.

Step 4: Returning the Result After evaluating all possible groupings of k workers, we find that the minimum total payment we can get away with to hire k = 2 workers while meeting the constraints is ans = 180.

This is a very simplified example for clarity, where efficiencies were all equal. In practice, efficiencies would vary, and the sorting step would play a critical role in determining which workers could potentially be hired together within budget constraints.

**Java Solution**

import java.util.Arrays;

import java.util.PriorityQueue;

class Solution {

    // This class represents workers with their quality and wage ratio

    class Worker {

        double wageQualityRatio;

        int quality;

        Worker(int quality, int wage) {

            this.quality = quality;

            this.wageQualityRatio = (double) wage / quality;

        }

    }

    public double mincostToHireWorkers(int[] quality, int[] wage, int k) {

        int numWorkers = quality.length;

        Worker[] workers = new Worker[numWorkers];

        // Create Worker instances combining both quality and wage.

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

            workers[i] = new Worker(quality[i], wage[i]);

        }

        // Sort the workers based on their wage-to-quality ratio.

        Arrays.sort(workers, (a, b) -> Double.compare(a.wageQualityRatio, b.wageQualityRatio));

        // Use a max heap to keep track of the highest quality workers.

        PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

        double minCost = Double.MAX\_VALUE; // Initialize minimum cost to a large number.

        int totalQuality = 0; // Tracks the total quality of hired workers.

        // Iterate through the sorted workers by increasing wage-to-quality ratio.

        for (Worker worker : workers) {

            // Add the current worker's quality to the total.

            totalQuality += worker.quality;

            // Add the current worker's quality to the max heap.

            maxHeap.offer(worker.quality);

            // If we have enough workers (exactly k), we can try to form an answer.

            if (maxHeap.size() == k) {

                // Calculate the cost based on current worker's wage-to-quality ratio.

                minCost = Math.min(minCost, totalQuality \* worker.wageQualityRatio);

                // Remove the worker with the highest quality (to maintain only k workers).

                totalQuality -= maxHeap.poll();

            }

        }

        return minCost; // Return the minimum cost found.

    }

}

[Time and Space Complexity](https://algo.monster/problems/runtime_summary)

**Time Complexity**

The time complexity of the code consists of several components:

Sorting: The list t is sorted based on the ratio of wage to quality using Python's built-in sort function, which has a time complexity of O(n log n) where n is the number of workers.

Heap Operations: For each worker in the sorted list, the algorithm performs heap push (heappush) and possibly heap pop (heappop) operations when the heap size is equal to k. Pushing onto the heap has a time complexity of O(log k) and popping from the heap also has a complexity of O(log k).

Given there are n workers, and therefore n heappush operations, and at most n heappop operations when the heap size reaches k, the heap operations give us a total time complexity of O(n log k).

So, the overall time complexity is the sum of the sorting complexity and the heap operations complexity: O(n log n + n log k).

**Space Complexity**

The space complexity is determined by the extra space used by the heap and the sorted list:

Sorted List: The sorted list t takes O(n) space since it contains a tuple for each worker.

Heap: The heap h can store up to k elements, so it has a space complexity of O(k).

Therefore, the total space complexity is the sum of the space complexities for the sorted list and the heap, which is O(n + k).

**Refer to**

[L1383.Maximum Performance of a Team (Ref.L2542)](note://WEBf5367f64c9712bbc2f91577b02b69305)

[L2542.Maximum Subsequence Score (Ref.L1383,L857)](note://WEBcc07e9686d4ff6360b7baba2fd021136)