2064. Minimized Maximum of Products Distributed to Any Store

You are given an integer `n` indicating there are `n` specialty retail stores. There are `m` product types of varying amounts, which are given as a **0-indexed** integer array `quantities`, where `quantities[i]` represents the number of products of the `ith` product type.

You need to distribute **all products** to the retail stores following these rules:

- A store can only be given **at most one product type** but can be given **any** amount of it.
- After distribution, each store will have been given some number of products (possibly `0`). Let `x` represent the maximum number of products given to any store. You want `x` to be as small as possible, i.e., you want to **minimize** the **maximum** number of products that are given to any store.

Return *the minimum possible* `x`.

Example 1:

```
Input: n = 6, quantities = [11,6]
Output: 3
Explanation: One optimal way is:
- The 11 products of type 0 are distributed to the first four stores in these
amounts: 2, 3, 3, 3
- The 6 products of type 1 are distributed to the other two stores in these
amounts: 3, 3
The maximum number of products given to any store is max(2, 3, 3, 3, 3, 3) = 3.
```

Example 2:

```
Input: n = 7, quantities = [15,10,10]
Output: 5
Explanation: One optimal way is:
- The 15 products of type 0 are distributed to the first three stores in these
amounts: 5, 5, 5
- The 10 products of type 1 are distributed to the next two stores in these
amounts: 5, 5
- The 10 products of type 2 are distributed to the last two stores in these
amounts: 5, 5
The maximum number of products given to any store is max(5, 5, 5, 5, 5, 5, 5) = 5.
```

Example 3:

```
Input: n = 1, quantities = [100000]
Output: 100000
Explanation: The only optimal way is:
- The 100000 products of type 0 are distributed to the only store.
The maximum number of products given to any store is max(100000) = 100000.
```

Constraints:

- `m == quantities.length`
- $1 <= m <= n <= 10^5$
- $1 <= quantities[i] <= 10^5$

2064. Minimized Maximum of Products Distributed to Any Store

```
/*
   Linear search
   Time complexity: O(m * hi)
   Space complexity: O(1)
   hi: maximum quantitites
   m: number of products
*/
class Solution {
   public:
      int minimizedMaximum(int n, std::vector<int>& quantities){
         // Get max quantity
         int hi=*std::max_element(quantities.begin(),quantities.end());

         // Linear search for the minimum quantity
         // that we can distribute to stores.
         for(int limit=1;limit<=hi;++limit){
            // Determine the maximum number of stores that we can
            // distribute to them `limit` amounts of product
            long long nb_stores=0;
            for(auto& qte: quantities){
               nb_stores+=(qte/limit+int(qte%limit!=0));
            }

            // if is possible to distribute `limit` product to
            // the stores, return that limit which represent
            // the best we can do
            if(nb_stores<=n) return limit;
         }

         return 0; // Never reached
      }
};
```

2064. Minimized Maximum of Products Distributed to Any Store

```
/*
   Binary search
   Time complexity: O(m log hi)
   Space complexity: O(1)
   hi: maximum quantitites
   m: number of products
*/
class Solution {
   public:
      int minimizedMaximum(int n, std::vector<int>& quantities){
         // Get max quantity
         int hi=*std::max_element(quantities.begin(),quantities.end());

         int lo=1;
         int ans=0;

         // Binary search for the minimum quantity
         // that we can distribute to stores.
         while(lo<=hi){
            int limit=(lo+hi)/2;
            // Determine the maximum number of stores that we can
            // distribute to them `limit` amounts of product
            long long nb_stores=0;
            for(auto& qte: quantities){
               nb_stores+=(qte/limit+int(qte%limit!=0));
            }

            // if is possible to distribute `limit` product to
            // the stores, return that limit which represent
            // the best we can do
            if(nb_stores<=n){
               ans=limit;
               hi=limit-1;
            }
            else lo=limit+1;
         }

         return ans; // Never reached
      }
};
```

Approach 2: Greedy Approach Using a Heap

Intuition

The key idea of this approach is to assign stores to product types in an optimal way, rather than assigning products to stores. Initially, each product type is assigned one store, which is guaranteed by the constraint $m \le n$. After this, we focus on which product types should receive additional stores. The algorithm greedily selects the product type i with the highest ratio of $quantity[i]$ to assigned_stores[i] , assigning the next available store to that product type.

Since we need to repeatedly access the product type with the highest ratio and update the ratios as stores are assigned, a priority queue (max-heap) is useful for efficiently managing these operations.

Consider an arbitrary distribution of stores to products, represented as $[s_0, s_1, s_2, \ldots, s_{m-1}]$, where s_i denotes the number of stores assigned to the $i-th$ product type. The specific indices of stores assigned or the order of assignment don't affect the result.

To minimize the load on any single store, the products of type i should be distributed as evenly as possible across its s_i assigned stores. This ensures that each store handling products of type i will have no more than $\lceil \frac{quantities_i}{s_i} \rceil$ products

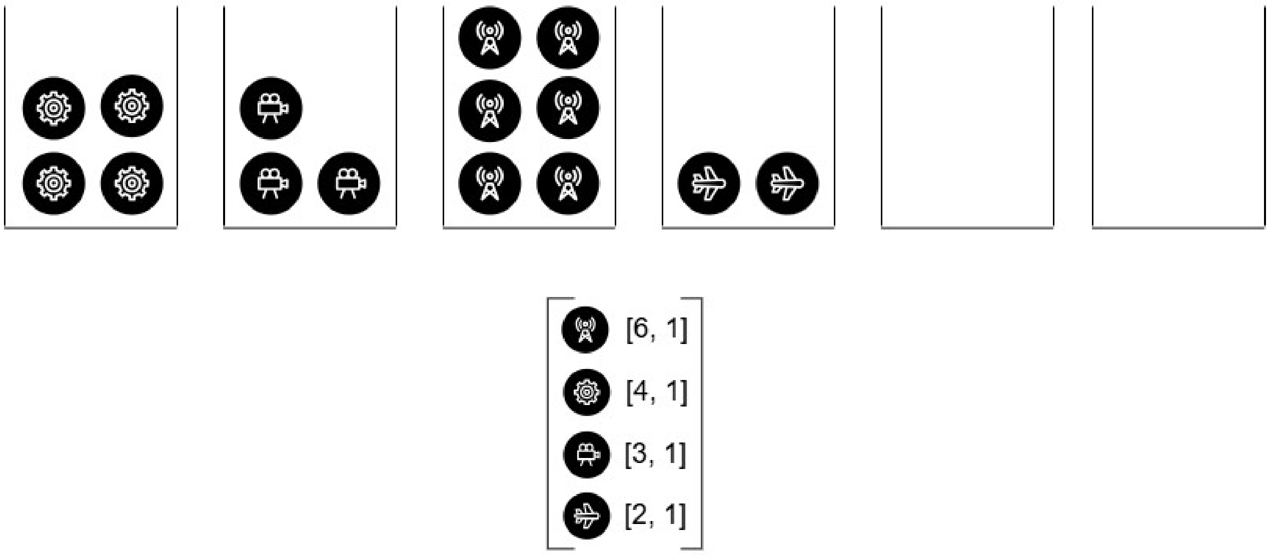
Thus, our objective is to minimize the maximum number of products any store receives. The function should return:

$$f(i) = max_{i \in [0, m-1]} \lceil \frac{quantities_i}{s_i} \rceil$$

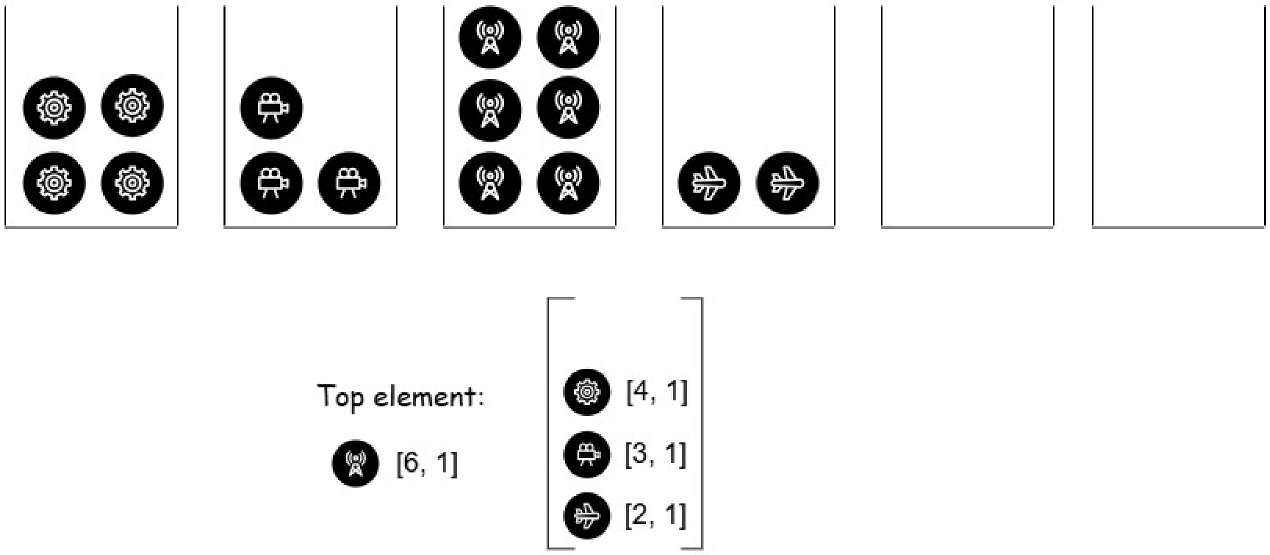
Now, consider the greedy approach: If at any point in the algorithm, we fail to assign the next available store to the product type with the highest ratio $quantity[i]$ to assigned_stores[i] , that ratio will remain the largest, leading to a non-optimal distribution. This would cause the highest ratio to dominate, violating our goal of minimizing the maximum number of products per store.

To gain a better understanding of the algorithm, let's revisit our initial example with $n=6$ and $quantities=[4,3,6,2]$.

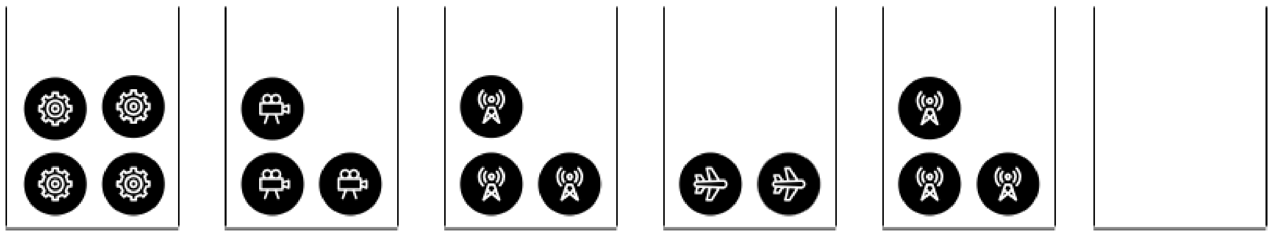
1. Begin by assigning a store to each product type.

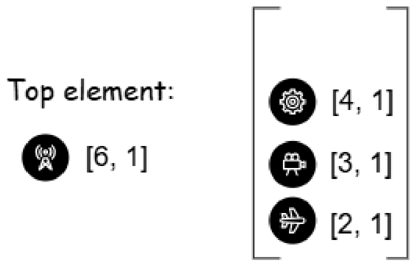


2. Pop the first element of the priority queue: ● [6, 1] .

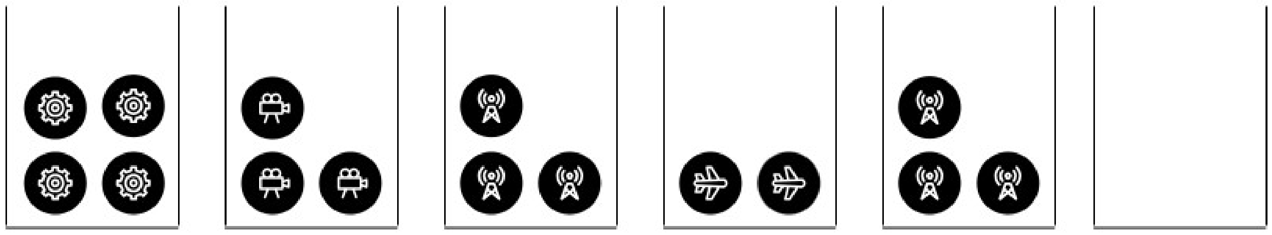


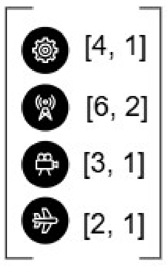
3. Assign store #5 to the product type ●. Split products of product type ● evenly between stores #3 and #5.



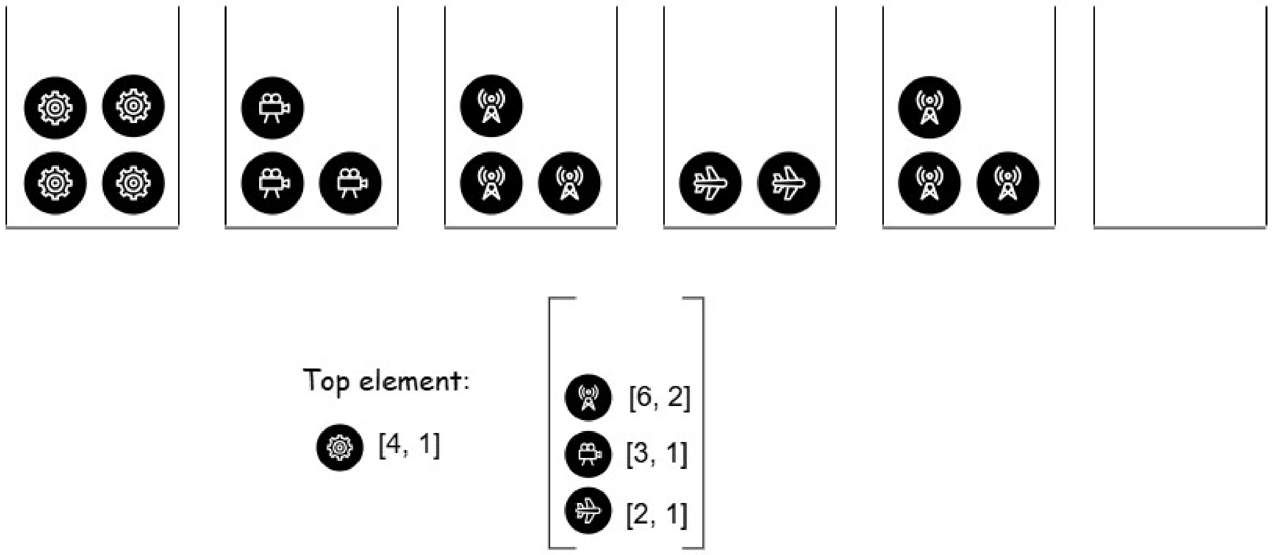


4. Push element ● [6, 1 + 1] back into the priority queue.

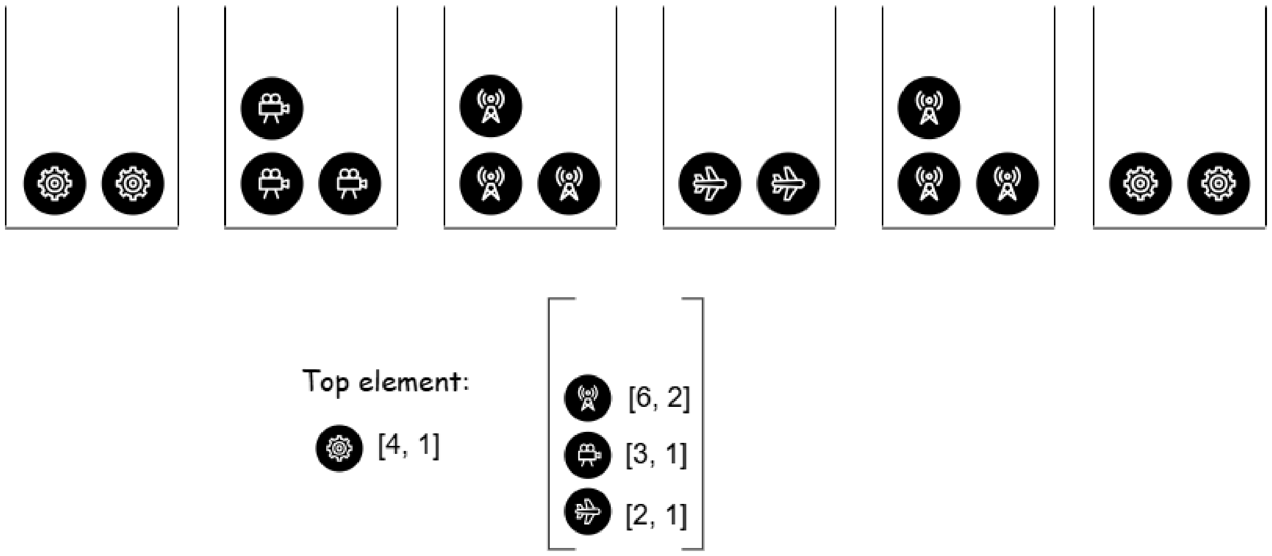




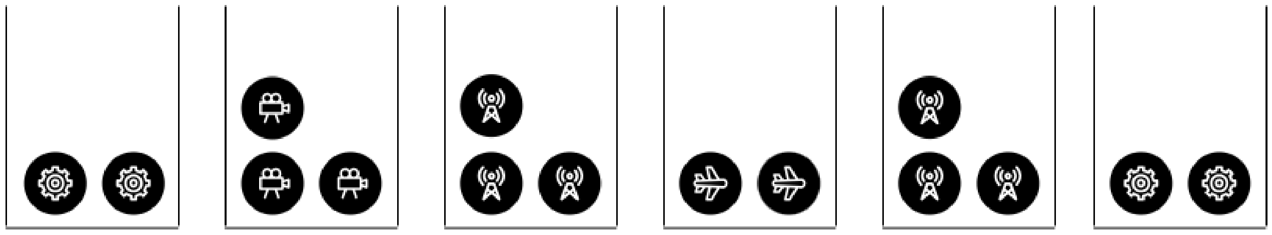
5. Pop the first element of the priority queue: [4, 1] .

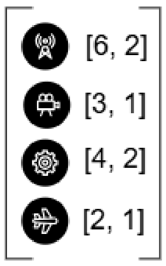


6. Assign store #6 to the product type . Split products of product type evenly between stores #1 and #6.

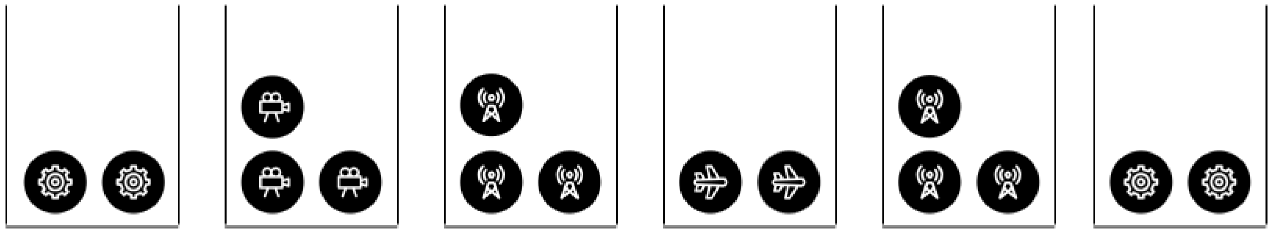


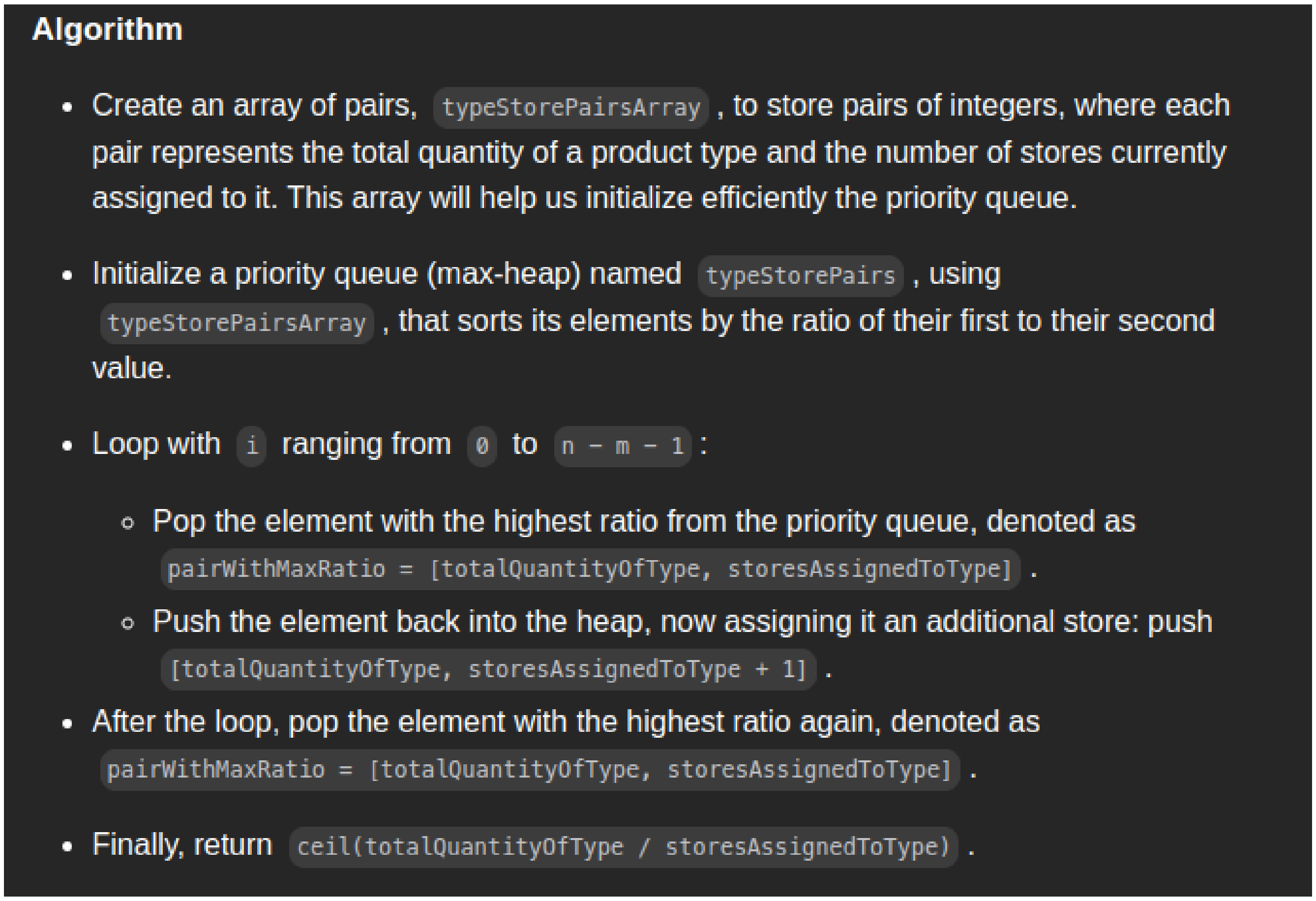
7. Push element ● [4, 1 + 1] back into the priority queue.





8. We have achieved the optimal distribution.





Algorithm

- Create an array of pairs, `typeStorePairsArray`, to store pairs of integers, where each pair represents the total quantity of a product type and the number of stores currently assigned to it. This array will help us initialize efficiently the priority queue.
- Initialize a priority queue (max-heap) named `typeStorePairs`, using `typeStorePairsArray`, that sorts its elements by the ratio of their first to their second value.
- Loop with `i` ranging from `0` to `n - m - 1`:
    - Pop the element with the highest ratio from the priority queue, denoted as `pairWithMaxRatio = [totalQuantityOfType, storesAssignedToType]`.
    - Push the element back into the heap, now assigning it an additional store: push `[totalQuantityOfType, storesAssignedToType + 1]`.
- After the loop, pop the element with the highest ratio again, denoted as `pairWithMaxRatio = [totalQuantityOfType, storesAssignedToType]`.
- Finally, return `ceil(totalQuantityOfType / storesAssignedToType)`.

```
class Solution {
public:
    int minimizedMaximum(int n, vector<int>& quantities) {
        int m = quantities.size();

        // Define a custom comparator for the priority queue
        // It sorts the pairs based on the ratio of their first to their second
        // element
        auto compareTypeStorePairs = [](pair<int, int>& a, pair<int, int>& b) {
            return (long long)a.first * b.second <
                   (long long)a.second * b.first;
        };

        // Helper array - useful for the efficient initialization of the
        // priority queue
        vector<pair<int, int>> typeStorePairsArray;

        // Push all product types to the array, after assigning one store to
        // each of them
        for (int i = 0; i < m; i++) {
            typeStorePairsArray.push_back({quantities[i], 1});
        }

        // Initialize the priority queue
        priority_queue<pair<int, int>, vector<pair<int, int>>,
                       decltype(compareTypeStorePairs)>
            typeStorePairs(typeStorePairsArray.begin(),
                           typeStorePairsArray.begin() + m);

        // Iterate over the remaining n - m stores.
        for (int i = 0; i < n - m; i++) {
            // Pop first element
            pair<int, int> pairWithMaxRatio = typeStorePairs.top();
            int totalQuantityOfType = pairWithMaxRatio.first;
            int storesAssignedToType = pairWithMaxRatio.second;
            typeStorePairs.pop();

            // Push same element after assigning one more store to its product
            // type
            typeStorePairs.push(
                {totalQuantityOfType, storesAssignedToType + 1});
        }

        // Pop first element
        pair<int, int> pairWithMaxRatio = typeStorePairs.top();
        int totalQuantityOfType = pairWithMaxRatio.first;
        int storesAssignedToType = pairWithMaxRatio.second;

        // Return the maximum minimum ratio
        return ceil((double)totalQuantityOfType / storesAssignedToType);
    }
};
```

Complexity Analysis

- Time complexity: $O(m+(n-m)logm)$

  We first iterate over the $quantities$ array, pushing each value as the first element of a pair into the helper array. This operation takes $O(m)$ time.

  We then initialize a priority queue (heap) using the elements from the array. Building the heap takes $O(m)$ time because heapify is performed in linear time.

  After that, we enter a second loop that runs $n-m$ times. In each iteration, we perform one pop and one push operation on the priority queue. Both operations take $O(logm)$ time, so this loop has a total time complexity of $O((n-m)logm)$.

  Combining the time complexities of the initialization, heap construction, and store allocation, the overall time complexity of the algorithm is: $O(m+(n-m)logm)$.

- Space complexity: $O(m)$

  The priority queue has a size of `m` since each value of the $quantities$ array is inserted as the first element of exactly one $typeSortPair$.