3066. Minimum Operations to Exceed Threshold Value II

You are given a **0-indexed** integer array `nums`, and an integer `k`.

In one operation, you will:

- Take the two smallest integers `x` and `y` in `nums`.
- Remove `x` and `y` from `nums`.
- Add `min(x, y) * 2 + max(x, y)` anywhere in the array.

Note that you can only apply the described operation if `nums` contains at least two elements.

Return *the **minimum** number of operations needed so that all elements of the array are greater than or equal to* `k`.

Example 1:

```
Input: nums = [2,11,10,1,3], k = 10
Output: 2
Explanation: In the first operation, we remove elements 1 and 2, then add 1 * 2 + 2 to nums. nums becomes equal to [4, 11, 10, 3].
In the second operation, we remove elements 3 and 4, then add 3 * 2 + 4 to nums. nums becomes equal to [10, 11, 10].
At this stage, all the elements of nums are greater than or equal to 10 so we can stop.
It can be shown that 2 is the minimum number of operations needed so that all elements of the array are greater than or equal to 10.
```

Example 2:

```
Input: nums = [1,1,2,4,9], k = 20
Output: 4
Explanation: After one operation, nums becomes equal to [2, 4, 9, 3].
After two operations, nums becomes equal to [7, 4, 9].
After three operations, nums becomes equal to [15, 9].
After four operations, nums becomes equal to [33].
At this stage, all the elements of nums are greater than 20 so we can stop.
It can be shown that 4 is the minimum number of operations needed so that all elements of the array are greater than or equal to 20.
```

Constraints:

- `2 <= nums.length <=` $2*10^5$
- `1 <= nums[i] <=` 10^9
- `1 <= k <=` 10^9
- The input is generated such that an answer always exists. That is, there exists some sequence of operations after which all elements of the array are greater than or equal to `k`.

Overview

We are given an array `nums` and an integer `k`. We repeatedly have to apply the following operation until all elements of `nums` are greater than or equal to `k`:

1. Remove the two smallest numbers `x` and `y` from `nums` from the array.
2. Add a new element `min(x, y) * 2 + max(x, y)` back into `nums`. The placement of this element doesn't matter.

We have to find out how many of the above operations are needed to make all elements in `nums` greater than or equal to `k`.

> Note that the described operation can only be applied if `nums` contains at least two elements.

Approach: Priority Queue

Intuition

For a straightforward approach, we can simulate the operations by maintaining a list that holds the current elements of $nums$. Then, we can scan through all elements of $nums$ in this list and take out the two smallest integers. If these integers are not greater than or equal to k, then we know we have to keep applying the operation, so we can append $min(x, y) * 2 + max(x, y)$ to our list. We can maintain a counter and repeat this operation until the two smallest integers are greater than or equal to k (if the two smallest integers are greater than or equal to k, then so are the rest of the elements, and we can stop applying the operations).

However, this simulation is time-consuming. Scanning through $nums$ and finding the two smallest integers before each operation takes $O(N)$ time. To find the two smallest integers more efficiently, we can use a priority queue (min-heap) instead of a list.

In a min heap, the smallest element is at the top of the tree and can be removed in $O(logN)$ time. Thus, for each operation, we can remove from the top of the heap twice to get the two smallest integers x and y, and then add back into our heap $min(x, y) * 2 + max(x, y)$. Note that adding elements into our heap also takes $O(logN)$ time. Thus, using a heap will improve our operation time from $O(N)$ to $O(logN)$.

Furthermore, checking for our stopping condition is also quicker. With a min heap, we can access the smallest element in $O(1)$ time. Until this smallest element is greater than or equal to k, we know we have to keep applying the operation.

3066. Minimum Operations to Exceed Threshold Value II

```
/*
   Min heap
   Time complexity: O(nlogn)
   Space complexity: O(n)
*/
typedef std::vector<int> vi;
typedef  long long ll;
typedef std::vector<ll> vll;

class Solution {
   public:
      int minOperations(vi& nums, int k){
         if(nums.size()<2) return 0;

         std::priority_queue<ll,vll,std::greater<ll>> min_heap;
         for(auto& e: nums) min_heap.push(e);

         int ans=0;
         while(min_heap.size()>=2 && min_heap.top()<k){
            ll x=min_heap.top();
            min_heap.pop();
            ll y=min_heap.top();
            min_heap.pop();
            min_heap.push(2*std::min(x,y)*1ll+std::max(x,y)*1ll);
            ans++;
         }
         return ans;
      }
};
```

Complexity Analysis

Let N be the size of `nums`.

- Time Complexity: $O(NlogN)$

  In the worst case, we have to apply N operations because each operation reduces the heap size by 1 (removing two elements and adding one). Each heap operation takes $O(logN)$ time, resulting in an overall time complexity of $O(NlogN)$.

- Space Complexity: $O(N)$

  At the start, our heap contains all elements from `nums`, so the space complexity is $O(N)$.