<https://leetcode.com/problems/find-the-most-competitive-subsequence/description/>

Given an integer array nums and a positive integer k, return the most competitive subsequence of nums of size k.

An array's subsequence is a resulting sequence obtained by erasing some (possibly zero) elements from the array.

We define that a subsequence a is more competitive than a subsequence b (of the same length) if in the first position where a and b differ, subsequence a has a number less than the corresponding number in b. For example, [1,3,4] is more competitive than [1,3,5] because the first position they differ is at the final number, and 4 is less than 5.

**Example 1:**

**Input:** nums = [3,5,2,6], k = 2

**Output:** [2,6]

**Explanation:** Among the set of every possible subsequence: {[3,5], [3,2], [3,6], [5,2], [5,6], [2,6]}, [2,6] is the most competitive.

**Example 2:**

**Input:** nums = [2,4,3,3,5,4,9,6], k = 4

**Output:** [2,3,3,4]

**Constraints:**

1 <= nums.length <= 10^5

0 <= nums[i] <= 10^9

1 <= k <= nums.length

**Attempt 1: 2024-09-24**

**Solution 1: Monotonic Increasing Stack (30 min)**

class Solution {

    public int[] mostCompetitive(int[] nums, int k) {

        int len = nums.length;

        int[] result = new int[k];

        Stack<Integer> stack = new Stack<>();

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

            // While the stack is not empty, and the current element is smaller than

            // the top element of the stack, and there are enough elements remaining

            // to form a sequence of length k we pop the stack since the current element

            // gives a more competitive sequence.

            // The '- 1' in 'len - i - 1' is important, need to exclude current number

            // at index 'i', because current number will be used to replace the one

            // pop out from stack (if pop out triggered), one in, one out, cancel out

            // each other on count perspective, the total count of numbers after current

            // number should no less than 'k - stack.size()', in case each of them used

            // to fill in stack to meet 'k' size result requirement

            while(!stack.isEmpty() && nums[stack.peek()] > nums[i] && k - stack.size() <= len - i - 1) {

                stack.pop();

            }

            // If the stack size is less than k, we can push the current element onto the stack.

            if(stack.size() < k) {

                stack.push(i);

            }

        }

        // Pop elements from the stack to fill the answer array in reverse order

        // since the elements in deque are in reverse order of the required sequence.

        for(int i = k - 1; i >= 0; i--) {

            result[i] = nums[stack.pop()];

        }

        return result;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/find-the-most-competitive-subsequence/solutions/1029198/c-o-n-solution-with-detailed-explanation/>

**Intuition**

By taking a close look at the following test cases, we can understand the problem better:

Consider :

[3,5,2,6] and k=2

Answer= [2,6]

[3,5,2,6] and k=3

Answer= [3,2,6]

In the first case, our answer began with 2 which was the smallest number available in the array. However, in the second case we could not start our answer with 2 as we need a total of k=3 numbers in the answer. Thus we begin our answer with the smallest number we can find, which **also** has enough numbers after it to contribute to the final answer.

**Algorithm**

For each arr[i] we have 2 options: either it will be a part of our ans or it won't be. Thus there's a 'dilemna' we have to resolve for each arr[i].

We shall use a stack, which will at max contain k elements at a time. These will be the elements whose dilemna is unresolved. At the end, this stack will contain our final answer.

Start traversing the arr. PUSH the arr[i] in the stack if stack.size() < k. We will resolve this pushed arr[i] 's dilemna at the time of POP operation (explained next).

POP **all** the elements whose dilemna is resolved ie they will NOT be a part of the ans. This will be the case when current arr[i] is smaller than stack.top(), and arr[i] **also** has sufficient number of elements after it to form the final answer.

On following the above steps, at the end our stack will contain the final answer. Convert it to an array and return it.

**Complexity**

Time Complexity will be O(n) and space complexity will be O(k) as the stack will at max contain k elements.

**Follow up**

Instead of using a stack and then storing it to vector and returning, we can simply use a vector from the very beginning. Use vector.back() to access 'top' element and vector.pop\_back() for popping. At the end simply return this vector we were using like a stack.

vector<int> mostCompetitive(vector<int>& arr, int k){

int n= arr.size();

stack<int> s; //stores the elements which have a dilemna. In the end, the elements who remain form the answer.

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

//n-i-1 elements are left in the array after ith index and k-s.size() is the no. of elements still required for our answer

while(!s.empty() && arr[i]<s.top() && (n-i-1 >= k-s.size())){

s.pop();

}

//push elements if stack has less than k elements (these elements are the ones whose dilemna has not been resolved yet)

if(s.size()<k){

s.push(arr[i]);

}

}

//stack s contains our answer now, converting to a vector

vector<int> ans;

while(!s.empty()){

ans.push\_back(s.top());

s.pop();

}

reverse(ans.begin(), ans.end());

return ans;

}

**Refer to**

<https://algo.monster/liteproblems/1673>

**Problem Description**

In this problem, we are given an array nums of integers and another integer k. Our task is to find the most competitive subsequence of the array nums that has a size of k. A subsequence here means a sequence that can be derived by deleting any number of elements (including zero) from the array without changing the order of the remaining elements.

A subsequence is considered more competitive if at the first point of difference between two subsequences, the number in the more competitive one is smaller. For example, the subsequence [1,3,4] is more competitive than [1,3,5] because, at their first differing position, 4 is smaller than 5.

To put it another way, we are looking for the smallest lexicographically subsequence of length k from the given array, where "smallest" refers to the subsequence that would come first if you listed all possible subsequences of the given length in numerical order.

**Intuition**

The intuition behind approaching this problem lies in maintaining a stack that can help us keep track of potential candidates for the most competitive subsequence. Here's the thought process:

We move through the array and for each element, we decide if it should be included in the stack (which represents the current most competitive subsequence we have built up).

We apply two checks while considering to push an element onto the stack: a. If the current element is smaller than the top of the stack (last element in our current subsequence), it could lead to a more competitive subsequence. So we might want to remove the larger elements from the top of the stack. b. However, we can only remove elements from the stack if we are sure that we can still complete our subsequence of size k. In other words, there must be enough elements left in the array to fill the stack after popping. This is guaranteed when len(stk) + n - i > k where stk is the stack, n is the number of elements in nums, and i is the current index.

After considering the above two points, if our stack size is less than k, we push the current element onto the stack as a potential candidate for the most competitive sequence. We keep doing this until we have scanned through all elements in the array.

After the loop finishes, the stack contains the most competitive subsequence of size k.

The stack effectively stores the smallest numbers seen so far and pops the larger ones that could be replaced by the following smaller numbers to maintain the competitiveness of the sequence. By the time the scanning of the array is complete, we have the most competitive subsequence.

**Solution Approach**

The solution uses a stack as a data structure to implement a greedy approach. Here's a step-by-step breakdown of the algorithm:

Initialize an empty list called stk which will be treated like a stack. This will be used to store the elements of the most competitive subsequence.

Determine the length of the input array nums and store it in a variable n.

Loop through each element v and its index i in the array using enumerate(nums).

Inside the loop, there is a while loop with a compound condition that is used to decide whether the top element should be popped from the stack:

The stack should not be empty (there's an element to potentially pop).

The current element v is less than the last element in the stack (stk[-1]). This check is important because a smaller element makes a subsequence more competitive.

There are still enough elements left in nums (after ith index) to fill the stack to a length of k (the condition len(stk) + n - i > k ensures this).

If the above conditions are met, the top element of the stack is popped because keeping it would make our subsequence less competitive. We continue popping until the conditions fail, meaning we either exhaust the stack or the current element v is not smaller than the last in stk or there aren't enough elements remaining to reach length k.

After the while loop, a check ensures that we only add elements to stk if its length is less than k. Otherwise, our subsequence would exceed the desired length.

We append the current element v to the stack if the length of the stack is less than k.

After iterating over all elements in nums, the list stk will contain exactly k elements, which forms the most competitive sequence possible, and thus is returned.

During each iteration, the algorithm greedily tries to make the subsequence as small as possible by comparing the current number with the ones already in stk. This use of a stack allows the algorithm to keep the subsequence always in a competitive state by ensuring that each new added number would not make it any worse compared to other potential subsequences.

The main algorithmic patterns used here are:

Greedy algorithm: Making local optimal choices at each step with the hope of finding a global optimum, i.e., the most competitive subsequence.

Stack: Providing a way to keep track of elements and efficiently facilitate the necessary add and remove operations as per competitive conditions.

**Example Walkthrough**

Let's illustrate the solution approach with a small example. Suppose we have the following array nums and integer k:

nums = [3, 5, 2, 6]

k = 2

We want to find the most competitive subsequence of size k from nums.

Following the solution approach:

Initialize an empty stack stk.

Determine the length of nums, which is n = 4.

Begin iterating through nums:

At index i = 0, v = 3, there is nothing to pop because stk is empty. We add 3 to stk.

At index i = 1, v = 5, there is nothing to pop because 5 is not less than the last element in stk, which is 3. Since len(stk) < k, we add 5 to stk.

At index i = 2, v = 2, v is less than the last element in stk (which is 5) and len(stk) + (n - i) > k (2 elements in stk + 2 elements left in nums > 2). So we pop 5 from stk and add 2.

At index i = 3, v = 6, there is nothing to pop since 6 is greater than the last element in stk (which is 2). However, len(stk) is already k, so we don’t add 6 to stk.

After the loop finishes, stk = [2, 3] which forms the most competitive subsequence because it is lexicographically the smallest subsequence of length k that we can form from nums.

The resulting most competitive subsequence is [2, 3]. The stack has helped us maintain the potential candidates for the subsequence efficiently, ensuring at each step that any larger numbers that could be replaced by smaller ones and do not contribute to the competitiveness are removed.

**Solution Implementation**

class Solution {

public int[] mostCompetitive(int[] nums, int k) {

// Use a deque as a stack to maintain the most competitive sequence.

Deque<Integer> stack = new ArrayDeque<>();

int n = nums.length;

// Iterate over the array elements.

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

// While the stack is not empty, and the current element is smaller than the top element of the stack,

// and there are enough elements remaining to form a sequence of length k

// we pop the stack since the current element gives a more competitive sequence.

while (!stack.isEmpty() && stack.peek() > nums[i] && stack.size() + n - i > k) {

stack.pop();

}

// If the stack size is less than k, we can push the current element onto the stack.

if (stack.size() < k) {

stack.push(nums[i]);

}

}

// Create an array to store the most competitive sequence.

int[] answer = new int[stack.size()];

// Pop elements from the stack to fill the answer array in reverse order

// since the elements in deque are in reverse order of the required sequence.

for (int i = answer.length - 1; i >= 0; --i) {

answer[i] = stack.pop();

}

// Return the array containing the most competitive sequence

return answer;

}

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the code is O(n), where n is the length of the input list nums. Even though there is a nested loop caused by the while loop within the for loop, each element is pushed to and popped from the stack at most once. This guarantees that each operation is bounded by 2n (each element pushed and popped at most once), which simplifies to O(n).

**Space Complexity**

The space complexity is O(k) since we are using a stack to keep track of the most competitive subsequence, which, according to the problem, can have a maximum length of k. No other data structures are used that grow with the size of the input.

**Refer to chatGPT**

LeetCode 1673 is titled "Find the Most Competitive Subsequence." The task is to find the most competitive subsequence of length k from an array. A subsequence is considered competitive if it has the smallest lexicographical order among all possible subsequences of the same length.

**Problem:**

Given an integer array nums and an integer k, return the most competitive subsequence of nums of length k.

**Approach:**

We can solve this problem using a **greedy approach** with a **monotonic stack**. The key idea is to maintain a stack where we keep elements in an increasing order while ensuring that we can still collect enough elements to form the subsequence of length k.

The algorithm works as follows:

Initialize an empty stack.

Iterate through the elements of nums.

For each element:

Pop elements from the stack if the current element is smaller than the top of the stack, and if there are enough elements left to form a subsequence of length k.

Push the current element to the stack.

At the end, the stack will contain the most competitive subsequence of length k.

public class Solution {

public int[] mostCompetitive(int[] nums, int k) {

// Stack to maintain the most competitive subsequence

Stack<Integer> stack = new Stack<>();

int n = nums.length;

// Iterate through the array

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

// While the stack is not empty and the top element is greater than the current element

// and we can still pick enough elements to fill the subsequence

while (!stack.isEmpty() && stack.peek() > nums[i] && stack.size() + (n - i) > k) {

stack.pop();

}

// Push the current element to the stack if the stack has less than k elements

if (stack.size() < k) {

stack.push(nums[i]);

}

}

// Convert the stack into an array of length k

int[] result = new int[k];

for (int i = k - 1; i >= 0; i--) {

result[i] = stack.pop();

}

return result;

}

}

**Explanation:**

Stack: Used to maintain the most competitive subsequence.

While loop: Removes elements from the stack if a smaller element comes along, and there's still room to complete the subsequence.

Result Array: At the end, the stack contains the result, which is converted to an array.

**Refer to**

[L84.Largest Rectangle in Histogram](note://E2FC22D9715540C5A8F06F60FBBE886F)

[L402.Remove K Digits](note://09B83C91043B44D5BC818F72BEE20307)