<https://leetcode.com/problems/minimum-operations-to-make-the-array-k-increasing/description/>

You are given a **0-indexed** array arr consisting of n positive integers, and a positive integer k.

The array arr is called **K-increasing** if arr[i-k] <= arr[i] holds for every index i, where k <= i <= n-1.

For example, arr = [4, 1, 5, 2, 6, 2] is K-increasing for k = 2 because:

arr[0] <= arr[2] (4 <= 5)

arr[1] <= arr[3] (1 <= 2)

arr[2] <= arr[4] (5 <= 6)

arr[3] <= arr[5] (2 <= 2)

However, the same arr is not K-increasing for k = 1 (because arr[0] > arr[1]) or k = 3 (because arr[0] > arr[3]).

In one operation, you can choose an index i and change arr[i] into any positive integer.

Return the **minimum number of operations** required to make the array K-increasing for the given k.

**Example 1:**

**Input:** arr = [5,4,3,2,1], k = 1

**Output:** 4

**Explanation:**

For k = 1, the resultant array has to be non-decreasing.

Some of the K-increasing arrays that can be formed are [5,**6**,**7**,**8**,**9**], [**1**,**1**,**1**,**1**,1], [**2**,**2**,3,**4**,**4**]. All of them require 4 operations.

It is suboptimal to change the array to, for example, [**6**,**7**,**8**,**9**,**10**] because it would take 5 operations.It can be shown that we cannot make the array K-increasing in less than 4 operations.

**Example 2:**

**Input:** arr = [4,1,5,2,6,2], k = 2

**Output:** 0

**Explanation:**

This is the same example as the one in the problem description.

Here, for every index i where 2 <= i <= 5, arr[i-2] <=arr[i].Since the given array is already K-increasing, we do not need to perform any operations.

**Example 3:**

**Input:** arr = [4,1,5,2,6,2], k = 3

**Output:** 2

**Explanation:**

Indices 3 and 5 are the only ones not satisfying arr[i-3] <= arr[i] for 3 <= i <= 5.

One of the ways we can make the array K-increasing is by changing arr[3] to 4 and arr[5] to 5.

The array will now be [4,1,5,**4**,6,**5**].

Note that there can be other ways to make the array K-increasing, but none of them require less than 2 operations.

**Constraints:**

1 <= arr.length <= 10^5

1 <= arr[i], k <= arr.length

**Attempt 1: 2024-09-15**

**Solution 1: Brute Force + Longest Increasing Subsequence (30 min, purely based on Leetcode 300, 1964 solution, TLE 87/90)**

class Solution {

    public int kIncreasing(int[] arr, int k) {

        int len = arr.length;

        int count = 0;

        // We need to build k lists, find LIS for each list,

        // then sum up all remain elements besides LIS in

        // each list, the final count is minimum operations

        for(int i = 0; i < k; i++) {

            List<Integer> list = new ArrayList<>();

            int j = i;

            // Collect elements separated by k indices

            while(j < len) {

                list.add(arr[j]);

                j += k;

            }

            // Calculate the number of operations needed

            count += list.size() - lengthOfLIS(list);

        }

        return count;

    }

// The DP solution to find the LIS for given list same as L300

    private int lengthOfLIS(List<Integer> list) {

        int result = 1;

        int size = list.size();

        int[] dp = new int[size];

        Arrays.fill(dp, 1);

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

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

                if(list.get(j) <= list.get(i)) {

                    dp[i] = Math.max(dp[i], dp[j] + 1);

                }

            }

            result = Math.max(result, dp[i]);

        }

        return result;

    }

}

Time Complexity: O(n^2/k)

There are k subarrays, each of size approximately n/k, for each subarray,

we perform an O(n/k)^2 LIS calculation, thus, the total time complexity is: O(n^2/k)

Space Complexity: O(n)

**Solution 2: Longest Increasing Subsequence + Greedy + Binary Search (Find Upper Boundary) (30 min)**

class Solution {

    public int kIncreasing(int[] arr, int k) {

        int len = arr.length;

        int count = 0;

        // We need to build k lists, find LIS for each list,

        // then sum up all remain elements besides LIS in

        // each list, the final count is minimum operations

        for(int i = 0; i < k; i++) {

            List<Integer> list = new ArrayList<>();

            int j = i;

            // Collect elements separated by k indices

            while(j < len) {

                list.add(arr[j]);

                j += k;

            }

            // Calculate the number of operations needed

            count += list.size() - lengthOfLIS(list);

        }

        return count;

    }

    // The Binary Search solution to find the LIS for given list same as L1964

    private int lengthOfLIS(List<Integer> list) {

        List<Integer> tmp = new ArrayList<>();

        int size = list.size();

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

            int num = list.get(i);

            int insert\_idx = binarySearch(tmp, num);

            // If insert\_idx is equal to the size of tmp, it means the current

            // obstacle is greater than all the obstacles encountered so far, so we

            // add it to the end of tmp.

            if(insert\_idx == tmp.size()) {

                tmp.add(num);

            // If insert\_idx is less than the size of tmp, it means the current

            // obstacle can replace an obstacle in the existing increasing subsequence.

            // We update the obstacle at index insert\_idx in tmp with the current

            // obstacle. (The update happens because we follow greedy thought try to make

            // all previous obstacles on the course as small as possible, which benefit

            // us to put more obstacles on this course)

            } else {

                tmp.set(insert\_idx, num);

            }

        }

        return tmp.size();

    }

    private int binarySearch(List<Integer> tmp, int target) {

        int lo = 0;

        int hi = tmp.size() - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(tmp.get(mid) > target) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // 'lo - 1' means upper boundary, '+ 1' means insert index of current

        // 'target' value, the insert index of current obstacle should one

        // right position than the last target value position

        return lo - 1 + 1;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(n)

**An example to explain the strategy**

arr = {4,1,5,2,6,2}

=============================================================

Example 1: when k = 2

4 1

5 2

6 2

A1 = {arr[0], arr[2], arr[4]}

arr[0] -> arr[2] -> arr[4] = 4 -> 5 -> 6

=> len(A1) - LIS(A1) = 3 - 3 = 0

A2 = {arr[1], arr[3], arr[5]}

arr[1] -> arr[3] -> arr[5] = 1 -> 2 -> 2

=> len(A2) - LIS(A2) = 3 - 3 = 0

Sum up: (len(A1) - LIS(A1)) + (len(A2) - LIS(A2)) = 0 + 0 = 0

=============================================================

Example 2: when k = 3

4 1 5

2 6 2

arr[0] -> arr[3] = 4 -> 2 x

arr[1] -> arr[4] = 1 -> 6

arr[2] -> arr[5] = 5 -> 2 x

A1 = {arr[0], arr[3]}

arr[0] -> arr[3] = 4 -> 2 x

=> len(A1) - LIS(A1) = 2 - 1 = 1

A2 = {arr[1], arr[4]}

arr[1] -> arr[4] = 1 -> 6

=> len(A2) - LIS(A2) = 2 - 2 = 0

A3 = {arr[2], arr[5]}

arr[2] -> arr[5] = 5 -> 2 x

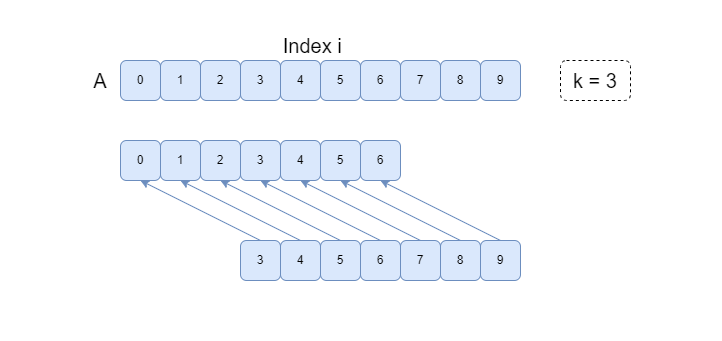
=> len(A3) - LIS(A3) = 2 - 1 = 1

Sum up: (len(A1) - LIS(A1)) + (len(A2) - LIS(A2)) + (len(A3) - LIS(A3)) = 1 + 0 + 1 = 2

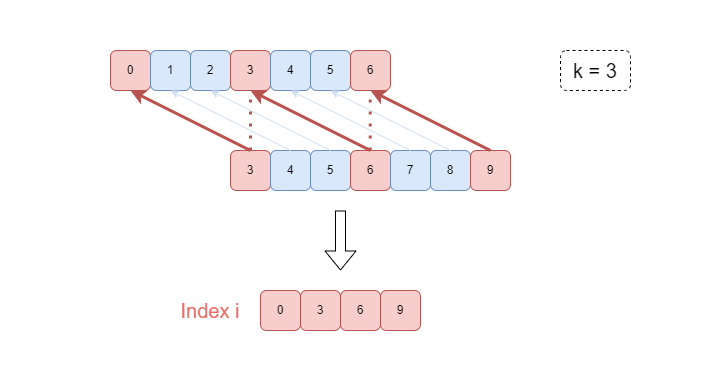
**Refer to**

<https://leetcode.com/problems/minimum-operations-to-make-the-array-k-increasing/solutions/1635026/python-explanation-with-pictures-lis/>

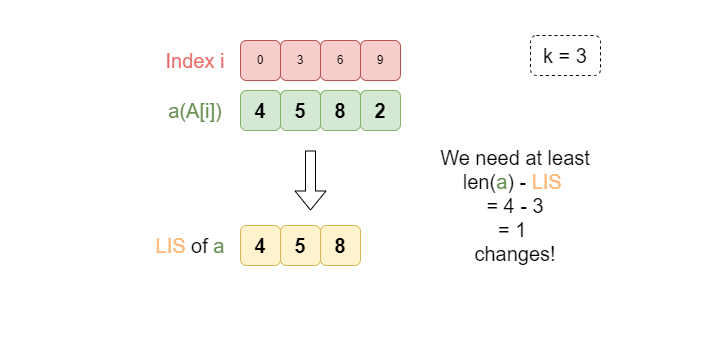
Given k = 3 for example. Let's stay that we start with index = 0, the first condition is **A[0] <= A[3]**,



We also need to guarantee **A[3] <= A[6]**, **A[6] <= A[9]**, etc, if such larger indexs exist. In a word, we need to make this subarray **a = [A[0], A[3], A[6], ... ]** of unique indexes **non-decreasing**.



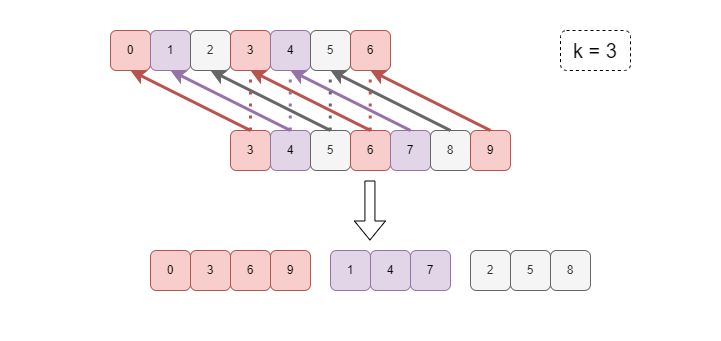
This sub-problem turns into finding the Largest Increasing Subsequence (LIS) of **a**, then the minimum number of changes we shall make equals to **len(a) - LIS(a)**



Since the final array is k-increasing, thus we have to compare at most k subarrays. [A[0], A[k], A[2k], ...], [A[1], A[k + 1], A[2k + 1], ...], ..., [A[k - 1], A[2k - 1], A[3k - 1], ...].

Notice that the changes on one "series" of indexes don't interference with other series, so we can safely calculate the number of changes for every series without affecting the answer.

Then the answer is the sum of **len(a) - LIS(a)** for each subarray.



**python**

def kIncreasing(self, A: List[int], k: int) -> int:

def LIS(arr):

ans = []

for a in arr:

idx = bisect.bisect\_right(ans, a)

if idx == len(ans):

ans.append(a)

else:

ans[idx] = a

return len(ans)

ans, n = 0, len(A)

for start in range(k):

cur, idx = [], start

while idx < n:

cur.append(A[idx])

idx += k

ans += len(cur) - LIS(cur)

return ans

**Refer to chatGPT**

class Solution {

public int kIncreasing(int[] A, int k) {

int ans = 0;

int n = A.length;

// Iterate over each group of elements based on the starting index

for (int start = 0; start < k; start++) {

List<Integer> cur = new ArrayList<>();

int idx = start;

// Collect elements separated by k indices

while (idx < n) {

cur.add(A[idx]);

idx += k;

}

// Calculate the number of operations needed

ans += cur.size() - LIS(cur);

}

return ans;

}

// Function to calculate the length of the longest non-decreasing subsequence

private int LIS(List<Integer> arr) {

List<Integer> ans = new ArrayList<>();

for (int a : arr) {

int idx = upperBound(ans, a); // Find the right place using binary search

if (idx == ans.size()) {

ans.add(a); // Append to the end if it's greater than all elements

} else {

ans.set(idx, a); // Replace the element at the found position

}

}

return ans.size(); // The size of the list is the length of the LNDS

}

// Custom binary search to find the first element greater than 'target'

private int upperBound(List<Integer> list, int target) {

int low = 0, high = list.size();

while (low < high) {

int mid = low + (high - low) / 2;

if (list.get(mid) <= target) {

low = mid + 1;

} else {

high = mid;

}

}

return low;

}

}

**Refer to**

<https://algo.monster/liteproblems/2111>

**Problem Description**

You are provided with an array arr of n positive integers and a positive integer k. An array is defined as K-increasing if for every index i such that k <= i <= n-1, the inequality arr[i-k] <= arr[i] is satisfied. In other words, for any element in the array, if you move k steps backward, you should not find a larger number.

For example, the array [4, 1, 5, 2, 6, 2] is K-increasing when k=2 because every element is greater than or equal to the element which is 2 places before it. However, it is not K-increasing for k=1 because 4 (element at index 0) is greater than 1 (element at index 1).

The task is to convert the array into a K-increasing array by performing the minimum number of operations. In one operation, you can choose an index i and change arr[i] to any positive integer.

The goal is to find out the minimum number of such operations needed to make the array K-increasing given the value of k.

**Intuition**

The solution utilizes a dynamic programming approach with a twist. The idea is that if you divide the array into k subarrays, where each subarray contains elements that are k indices apart in the original array, you'll notice that for the array to be K-increasing overall, each of these subarrays must be non-decreasing.

Here's the intuition broken down step by step:

Subarray Division: Consider the arr=[4, 1, 5, 2, 6, 2] and k=2; we have two subarrays [4, 5, 6] and [1, 2, 2]. If each of these subarrays is non-decreasing, the original array is K-increasing.

Longest Increasing Subsequence (LIS): For each subarray, we want to keep it non-decreasing with the minimum number of changes. To achieve this, we need to find the length of the Longest Increasing Subsequence (LIS) of the subarray. The reason behind this is that elements in the LIS do not need to be changed, as they already contribute to making the subarray non-decreasing.

Operations Count: Once we have the LIS length, the number of operations required to make the subarray non-decreasing is the total number of elements minus the LIS length. This is because we can keep the LIS as is and change the other elements to fit around it.

Summing Up: Since our original array is divided into k subarrays, we apply the LIS strategy for each subarray and sum up the operations required. This will give us the total minimum number of operations needed to make the entire array K-increasing.

The provided solution uses a helper function lis(arr) which calculates the required operations for a given subarray by finding the length of the LIS using binary search (bisect\_right). Then it uses list comprehension to sum up the operations for each subarray, which are slices of the original array (arr[i::k] for i in range(k)).

This approach is efficient because it reduces the problem to k individual LIS problems and avoids unnecessary changes to elements that are already part of the LIS, hence minimizing the number of operations.

**Solution Approach**

The key to implementing the solution is understanding the concept of Longest Increasing Subsequence (LIS) within the context of the given array and how it applies to each of the k subsequences.

The provided Python solution includes a nested function lis, which is the implementation of the LIS algorithm. This is not the traditional dynamic programming solution for LIS with O(n^2) complexity but a more efficient version that uses binary search (bisect\_right from the bisect module) and has a time complexity of O(n log n) for each subsequence.

Here's the breakdown of the algorithms and data structures used in the solution:

Nested Function lis(arr):

This function computes the length of the LIS for a given subarray.

It initializes an empty list t that will store the last element of the smallest increasing subsequence of each length found so far.

The function iterates over each element x in the subarray:

Using binary search (bisect\_right), it finds the position idx in t where x could be placed to either extend an existing subsequence or replace an element to create a potential new subsequence.

If idx is equal to the length of t, it means x is larger than any element in t, and we can append x to t, effectively extending the longest subsequence seen so far.

Otherwise, we replace the element at t[idx] with x, as x might be the start of a new potential subsequence or a smaller end element for a subsequence of length idx.

After processing all elements in the subarray, the length of LIS is obtained by subtracting the length of t from the length of the subarray (len(arr) - len(t)).

Combining the Results:

The main part of the solution is a single line that sums up the operation counts for each k subsequence: sum(lis(arr[i::k]) for i in range(k)).

It iterates over each start index from 0 to k-1 and takes every k-th element from the original array using slicing arr[i::k]. This yields k subsequences that must each individually be non-decreasing to satisfy the K-increasing property.

For each subsequence, it applies the lis function to find the number of operations needed, which is then accumulated to get the total minimum number of operations required for the entire array.

By applying the LIS algorithm separately to each of the k subsequences, the solution effectively translates the problem of making an array K-increasing into multiple independent subproblems. Each subproblem aims to minimize the adjustments within its subsequence, and the sum of the solutions to these subproblems is the minimum number of operations needed for the whole array.

**Example Walkthrough**

Let's consider a small example array arr = [3, 9, 4, 6, 7, 5] and k = 3. According to the given solution approach, we need to divide this array into k subarrays where each subarray contains elements that are k indices apart in the original array. Let's do this step by step:

Subarray Division:

With k = 3, we divide the original array into 3 subarrays: [3, 6], [9, 7], [4, 5].

These subarrays are created by taking every third element from the original array starting from indices 0, 1, and 2 respectively.

Longest Increasing Subsequence (LIS):

We then need to determine the LIS for each subarray:

For the first subarray [3, 6], the LIS is [3, 6] itself, and the length is 2.

For the second subarray [9, 7], since 7 is not larger than 9, the LIS is [7], and the length is 1.

For the third subarray [4, 5], the LIS is [4, 5], and the length is 2.

Operations Count:

We calculate the number of operations required to make each subarray non-decreasing:

For [3, 6], no operations are needed as it is already non-decreasing.

For [9, 7], we need 2 - 1 = 1 operation, changing 9 to a number not greater than 7 (e.g., 7 or any smaller number).

For [4, 5], no operations are needed as it is already non-decreasing.

Summing Up:

Sum up the operations required: 0 (for the first subarray) + 1 (for the second subarray) + 0 (for the third subarray) = 1.

Therefore, we need a minimum of 1 operation to make the entire array k-increasing.

This walkthrough illustrates the solution steps using a simple example, showcasing how to divide the original problem into smaller ones by finding the LIS of the subarrays and then deducing the minimum number of operations needed to achieve a k-increasing array.

**Solution Implementation**

class Solution {

public int kIncreasing(int[] arr, int k) {

int n = arr.length; // Get the length of the array

int ans = 0; // Initialize answer to 0

// Iterate over the first k elements

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

List<Integer> subsequence = new ArrayList<>(); // List to hold subsequences

// Populate the subsequences with elements spaced k apart

for (int j = i; j < n; j += k) {

subsequence.add(arr[j]);

}

// Increment the answer by the number of modifications needed

// to make the subsequence strictly increasing

ans += leastIncrementsNeeded(subsequence);

}

return ans; // Return the total number of modifications for all subsequences

}

// Determine the least number of increments needed to make the given list strictly increasing

private int leastIncrementsNeeded(List<Integer> arr) {

List<Integer> temp = new ArrayList<>(); // Temporary list to hold the longest increasing subsequence

for (int x : arr) {

int idx = findInsertionIndex(temp, x);

// If the element is greater than all elements in temp, add it to the end

// Otherwise, replace the first element that is greater or equal to x

if (idx == temp.size()) {

temp.add(x);

} else {

temp.set(idx, x);

}

}

// The difference between the list size and temp size is the number of increments needed

return arr.size() - temp.size();

}

// Binary search to find the rightmost index to insert the element

private int findInsertionIndex(List<Integer> arr, int x) {

int left = 0; // Left pointer for the binary search

int right = arr.size(); // Right pointer for the binary search

// Perform the binary search

while (left < right) {

// Compute mid-point, equivalent to (left + right) / 2 but avoids potential overflow

int mid = (left + right) >> 1;

// If current element is greater, ignore the right half

if (arr.get(mid) > x) {

right = mid;

} else { // Otherwise, ignore the left half

left = mid + 1;

}

}

return left; // Return the computed index

}

}

**Time and Space Complexity**

The given code implements a function that, for a given list arr and an integer k, finds the minimum number of elements to change to ensure that every k-th subsequence of the list is non-decreasing.

**Time Complexity:**

To analyze the time complexity, let’s break down the process:

The lis function is called k times, once for each of the k subsequences.

Inside the lis function, there's a loop that goes through the elements of a subsequence (which has a length of about n/k, where n is the length of arr).

In the worst case, bisect\_right performs a binary search, which has a time complexity of O(log m) where m is the size of the temporary list t.

The size of t can grow up to the size of the subsequence being considered, in the worst case approximated to n/k.

Putting it all together:

Single call to lis: O((n/k) \* log(n/k))

lis called k times: O(k \* (n/k) \* log(n/k)) = O((n \* log(n/k)))

Since we have a log term that depends on n/k, the overall time complexity isn't perfectly linear with respect to n. However, as k increases, the time complexity approaches O(n log n) since the subsequences processed by each call get shorter.

**Space Complexity:**

The space complexity can be evaluated by considering:

The temporary list t used inside the lis function, which holds the elements of the longest increasing subsequence (LIS) within a k-th subsequence;

t's size is at most n/k for a given k-th subsequence; However, t is reused for each subsequence and does not grow with k.

There are no additional data structures that grow with the size of the input, other than the input itself and the function call stack.

Hence, the space complexity is O(n/k), which simplifies to O(n) because we keep a single t for each subsequence.

**Please note:** since the actual maximum length of t can vary depending on the input arr, the space complexity in practice can be less than O(n) if the subsequences have a strong increasing trend, but in the worst case, it is O(n).

**Refer to**

[L300.Longest Increasing Subsequence](note://20C954956F6E4AA6A3575EA3146A1FCA)

[L1964.Find the Longest Valid Obstacle Course at Each Position (Ref.L300,L2111)](note://WEB19b46f112c3b422b47c007e35e1ef021)