Orderly Queue

Problem Description

You are given a string s and an integer k. You can choose one of the first k letters of s and append it at the end of the string..

Return *the lexicographically smallest string you could have after applying the mentioned step any number of moves*.

**Example 1:**

**Input:** s = "cba", k = 1

**Output:** "acb"

**Explanation:**

In the first move, we move the 1st character 'c' to the end, obtaining the string "bac".

In the second move, we move the 1st character 'b' to the end, obtaining the final result "acb".

**Example 2:**

**Input:** s = "baaca", k = 3

**Output:** "aaabc"

**Explanation:**

In the first move, we move the 1st character 'b' to the end, obtaining the string "aacab".

In the second move, we move the 3rd character 'c' to the end, obtaining the final result "aaabc".

**Constraints:**

* 1 <= k <= s.length <= 1000
* s consist of lowercase English letters.

SOLUTION:

**Approach 1: Mathematical**

**Intuition**

Let's call the move that takes the kthk^{th}*kth* letter from the beginning and puts it on the end a "*k-kick*" move.

Examining 1-kick moves, they let us consider the string as a "necklace" that may be rotated freely, where each bead of the necklace corresponds to a letter in the string. (Formally, this is the equivalence class under 1-kick moves.)

Examining 2-kick moves (in the context of treating the string as a necklace), they allow us to swap the positions of two adjacent beads. Thus, with 2-kick moves, every permutation of necklace is possible. (To actually construct the necklace, we bring the second smallest bead to be after the smallest, then the third smallest to be after the second smallest, and so on.)

The previous insight may be difficult to find. Another strategy is to write a brute force program to examine the result of 2-kick moves - then we might notice that 2-kick moves allow any permutation of the string.

Yet another strategy might be to explicitly construct new moves based on previous moves. If we perform a 2-kick move followed by many 1-kick moves, we can transform a string like "xyzzzzzz" -> "xzzzzzzy" -> "yxzzzzzz", proving that we can swap the positions of any two adjacent letters.

**Algorithm**

If k = 1, only rotations of s are possible, and the answer is the lexicographically smallest rotation.

If k > 1, any permutation of s is possible, and the answer is the letters of s written in lexicographic order.

**Complexity Analysis**

* Time Complexity: O(N2)O(N^2)*O*(*N*2), where NN*N* is the length of s.
  + If k = 1, we need O(N)O(N)*O*(*N*) time to build each new string and O(N)O(N)*O*(*N*) time to check whether it's the lexicographically smallest string among the strings generated so far. In total, there are NN*N* such different strings to build and check. Therefore, the time complexity for this case is O(N2)O(N^2)*O*(*N*2).
  + If k > 1, we need to convert our given string to an array of characters (this costs O(N)O(N)*O*(*N*) time), then sort the newly obtained array (sorting takes O(Nlog⁡N)O(N \log N)*O*(*N*log*N*) time), and build the output string from the sorted array which takes O(N)O(N)*O*(*N*) time.
  + Thus, the worst-case scenario is when k is 1, so the overall time complexity of the solution is O(N2)O(N^2)*O*(*N*2).
* Space Complexity: O(N)O(N)*O*(*N*).
  + If k = 1, we need the space to store only two strings: the lexicographically smallest string found so far and a newly built string, that will be compared to the lexicographically smallest string. This requires O(N)O(N)*O*(*N*) space.
  + If k > 1, we need O(N)O(N)*O*(*N*) space to store the character array. Other than that, sorting the array requires O(log⁡N)O(\log N)*O*(log*N*) additional space for Java and O(N)O(N)*O*(*N*) additional space for Python.
  + Therefore, the overall space complexity of the solution is O(N)O(N)*O*(*N*).