Solution Review: Problem Challenge 1

We'll cover the following

Rearrange String K Distance Apart (hard)

Given a string and a number 'K', find if the string can be rearranged such that the same characters are at least 'K' distance apart from each other.

Example 1:

```
Input: "mmpp", K=2
Output: "mpmp" or "pmpm"
Explanation: All same characters are 2 distance apart.
```

Example 2:

```
Input: "Programming", K=3
Output: "rgmPrgmiano" or "gmringmrPoa" or "gmrPagimnor" and a few more
Explanation: All same characters are 3 distance apart.
```

Example 3:

```
Input: "aab", K=2
Output: "aba"
Explanation: All same characters are 2 distance apart.
```

Example 4:

```
Input: "aappa", K=3
Output: ""
Explanation: We cannot find an arrangement of the string where any two 'a' are 3 distance apart.
```

Solution

This problem follows the Top 'K' Numbers pattern and is quite similar to Rearrange String. The only difference is that in the 'Rearrange String' the same characters need not be adjacent i.e., they should be at least '2' distance apart (in other words, there should be at least one character between two same characters), while in the current problem, the same characters should be 'K' distance apart.

Following a similar approach, since we were inserting a character back in the heap in the next iteration, in this problem, we will re-insert the character after 'K' iterations. We can keep track of previous characters in a queue to insert them back in the heap after 'K' iterations.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```python
from heapq import *
from collections import deque


def reorganize_string(str, k):
  if k <= 1:
    return str

  charFrequencyMap = {}
  for char in str:
    charFrequencyMap[char] = charFrequencyMap.get(char, 0) + 1

  maxHeap = []
  # add all characters to the max heap
  for char, frequency in charFrequencyMap.items():
    heappush(maxHeap, (-frequency, char))

  queue = deque()
  resultString = []
  while maxHeap:
    frequency, char = heappop(maxHeap)
    # append the current character to the result string and decrement its count
    resultString.append(char)
    # decrement the frequency and append to the queue
    queue.append((char, frequency+1))
    if len(queue) == k:
      char, frequency = queue.popleft()
      if -frequency > 0:
        heappush(maxHeap, (frequency, char))

  # if we were successful in appending all the characters to the result string, return it
```

Run | Save | Reset

Time complexity

The time complexity of the above algorithm is $O(N * logN)$ where 'N' is the number of characters in the input string.

Space complexity

The space complexity will be $O(N)$, as in the worst case, we need to store all the 'N' characters in the HashMap.