Rearrange String k Distance Apart

Given a non-empty string **str** and an integer **k**, rearrange the string such that the same characters are at least distance **k** from each other.

All input strings are given in lowercase letters. If it is not possible to rearrange the string, return an empty string `""`.

Example 1:

```
str = "aabbcc", k = 3

Result: "abcabc"

The same letters are at least distance 3 from each other.
```

Example 2:

```
str = "aaabc", k = 3

Answer: ""

It is not possible to rearrange the string.
```

Example 3:

```
str = "aaadbbcc", k = 2

Answer: "abacabcd"

Another possible answer is: "abcabcda"

The same letters are at least distance 2 from each other.
```

Credits:

Special thanks to @elmirap for adding this problem and creating all test cases.

Solution 1

key point: using cache during processing heap data.

new version:

```
class Solution {
public:
    string rearrangeString(string str, int k) {
        if(k == 0) return str;
        int length = (int)str.size();

        string res;
        unordered_map<char, int> dict;
        priority_queue<pair<int, char>> pq;

        for(char ch : str) dict[ch]++;
        for(auto it = dict.begin(); it != dict.end(); it++){
            pq.push(make_pair(it->second, it->first));
        }

        while(!pq.empty()){
            vector<pair<int, char>> cache; //store used char during one while loo
p
            int count = min(k, length); //count: how many steps in a while loop
            for(int i = 0; i < count; i++){
                if(pq.empty()) return "";
                auto tmp = pq.top();
                pq.pop();
                res.push_back(tmp.second);
                if(--tmp.first > 0) cache.push_back(tmp);
                length--;
            }
            for(auto p : cache) pq.push(p);
        }
        return res;
    }
};
```

old version:

```
class Solution {
    struct mycompare{
        bool operator()(pair<int, char>& p1, pair<int, char>& p2){
            if(p1.first == p2.first) return p1.second > p2.second;
            return p1.first < p2.first;
        }
    };
public:
    string rearrangeString(string str, int k) {
        if(k == 0) return str;
        unordered_map<char, int> dict;
        for(char ch : str) dict[ch]++;
        int left = (int)str.size();
        priority_queue<pair<int, char>, vector<pair<int, char>>, mycompare > pq;
        for(auto it = dict.begin(); it != dict.end(); it++){
            pq.push(make_pair(it->second, it->first));
        }
        string res;

        while(!pq.empty()){
            vector<pair<int, char>> cache;
            int count = min(k, left);
            for(int i = 0; i < count; i++){
                if(pq.empty()) return "";
                auto tmp = pq.top();
                pq.pop();
                res.push_back(tmp.second);
                if(--tmp.first > 0) cache.push_back(tmp);
                left--;
            }
            for(auto p : cache){
                pq.push(p);
            }
        }
        return res;
    }
};
```

written by sxycwzwzq original link here

Solution 2

This is a greedy problem.
Every time we want to find the best candidate: which is the character with the largest remaining count. Thus we will be having two arrays.
One count array to store the remaining count of every character. Another array to keep track of the most left position that one character can appear. We will iterated through these two array to find the best candidate for every position. Since the array is fixed size, it will take constant time to do this.
After we find the candidate, we update two arrays.

```
public class Solution {
    public String rearrangeString(String str, int k) {
        int length = str.length();
        int[] count = new int[26];
        int[] valid = new int[26];
        for(int i=0;i<length;i++){
            count[str.charAt(i)-'a']++;
        }
        StringBuilder sb = new StringBuilder();
        for(int index = 0;index<length;index++){
            int candidatePos = findValidMax(count, valid, index);
            if( candidatePos == -1) return "";
            count[candidatePos]--;
            valid[candidatePos] = index+k;
            sb.append((char)('a'+candidatePos));
        }
        return sb.toString();
    }

   private int findValidMax(int[] count, int[] valid, int index){
       int max = Integer.MIN_VALUE;
       int candidatePos = -1;
       for(int i=0;i<count.length;i++){
           if(count[i]>0 && count[i]>max && index>=valid[i]){
               max = count[i];
               candidatePos = i;
           }
       }
       return candidatePos;
   }
}
```

At first I was considering using PriorityQueue and referring to this post:
c++ unordered*map priority*queue solution using cache
I have doubts for this solution. If we have "abba", k=2; It seems we might end up with "abba" as the result. Since in the second while loop, I'm not sure 'a' or 'b' will be polled out first.
In Java, when two keys in PriorityQueue have same value, there is no guarantee on the order poll() will return. But I'm not sure how heap is implemented in C++.

written by OrlandoChen0308 original link here

Solution 3

```
import java.util.SortedSet;
import java.util.TreeSet;
public class Solution {
    public String rearrangeString(String str, int k) {
        if(k < 2) return str;
      int[] times = new int[26];
      for(int i = 0; i < str.length(); i++){
          ++times[str.charAt(i) - 'a'];
      }
      SortedSet<int[]> set = new TreeSet<int[]>(new Comparator<int[]>(){
          @Override
          public int compare(int[] a, int[] b){
              return a[0] == b[0] ? Integer.compare(a[1], b[1]) : Integer.compare
(b[0], a[0]);
          }
      });
      for(int i = 0; i < 26; i++){
          if(times[i] != 0){
            set.add(new int[]{times[i], i});
          }
      }
      int cycles = 0;
      int cur = cycles;
      Iterator<int[]> iter = set.iterator();
      char[] res = new char[str.length()];
      while(iter.hasNext()){
          int[] e = iter.next();
          for(int i = 0; i < e[0]; i++){
              res[cur] = (char)('a'+e[1]);
              if(cur > 0 && res[cur] == res[cur-1])
                return "";
              cur += k;
              if(cur >= str.length()){
                  cur = ++cycles;
              }
          }
      }
      return new String(res);
    }
}
```

written by fatalme original link here