<https://leetcode.ca/all/358.html>

Given a non-empty string **s** 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:

Input: s = "aabbcc", k = 3

Output: "abcabc"

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

Example 2:

Input: s = "aaabc", k = 3

Output: ""

Explanation: It is not possible to rearrange the string.

Example 3:

Input: s = "aaadbbcc", k = 2

Output: "abacabcd"

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

**Attempt 1: 2023-11-04**

**Solution 1: Priority Queue + Harsh Table (120 min)**

**Style 1: HashMap for frequency**

import java.util.\*;

public class Solution {

// 这道题给了我们一个字符串str，和一个整数k，让我们对字符串str重新排序，使得其中相同的字符之间的距离不小于k，

// 这道题的难度标为Hard，看来不是省油的灯。的确，这道题的解法用到了哈希表，堆，和贪婪算法。这道题我最开始想

// 的算法没有通过OJ的大集合超时了，下面的方法是参考网上大神的解法，发现十分的巧妙。我们需要一个哈希表来建立

// 字符和其出现次数之间的映射，然后需要一个堆来保存这每一对映射，按照出现次数来排序。然后如果堆不为空我们就

// 开始循环，我们找出k和str长度之间的较小值，然后从0遍历到这个较小值，对于每个遍历到的值，如果此时堆为空了，

// 说明此位置没法填入字符了，返回空字符串，否则我们从堆顶取出一对映射，然后把字母加入结果res中，此时映射的

// 个数减1，如果减1后的个数仍大于0，则我们将此映射加入临时集合v中，同时str的个数len减1，遍历完一次，

// 我们把临时集合中的映射对由加入堆中，参见代码如下：

public String rearrangeString(String str, int k) {

// Initialize the counter for each character

Map<Character, Integer> freq = new HashMap<>();

for(char c : str.toCharArray()) {

freq.put(c, freq.getOrDefault(c, 0) + 1);

}

// Sort the chars by frequency, if same frequency then compare alphabetical order

PriorityQueue<Character> maxPQ = new PriorityQueue<>((a, b) -> freq.get(a) == freq.get(b) ? a.compareTo(b) : freq.get(b) - freq.get(a));

for(char c : freq.keySet()) {

maxPQ.offer(c);

}

StringBuilder sb = new StringBuilder();

int len = str.length();

while(!maxPQ.isEmpty()) {

// 为何需要定义临时list ？

// 所有的剩余字符(全部减掉1个频次)都必须同时在下一轮中加入，而不是在本轮中再次加入

// 举个例子aaabbcc，a->3，b->2，c->2，第一轮过后a->2，b->2，c->2，如果直接

// 放入加入频次减1的a会导致a依然排在b，c之前，而根据均匀使用的原则，我们是希望先把

// a，b，c先分别使用一次的，而不是怼着a用，即希望重新排序时得到的是abc...而非

// aabc...，所以可以使用一个临时list来保存本轮已经均匀使用后剩下的所有字母

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

// 为何需要在k和len之间的大小 ？

// 为了对付aaadbbcc -> abc|abc|ad中最后的ad部分，此时k = 3,

// 但是len变为2，len小于3，必须按照len的长度而不再是k来填充

int sectionLen = Math.min(len, k);

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

if(maxPQ.isEmpty()) {

return "";

}

char c = maxPQ.poll();

sb.append(c);

freq.put(c, freq.get(c) - 1);

if(freq.get(c) > 0) {

tmp.add(c);

}

len--;

}

// 在前一轮对sectionLen个位置的填充结束后利用之前定义的临时list中收集的剩余字符进行新一轮填充

for(char c : tmp) {

maxPQ.offer(c);

}

}

return sb.toString();

}

public static void main(String[] args) {

Solution so = new Solution();

String s = "aaadbbcc";

//String s = "aabbcc";

//String s = "aaaabbcc";

String result = so.rearrangeString(s, 3);

System.out.println(result);

}

}

Time Complexity:O(nlog26) => O(n), 在这题heap存储的元素为26，因为所有字符都是lowercase

Space Complexity:O(n)

**Style 2: int[] array for frequency**

**第一个是统计frequency用hash table vs int[]的区别，事实证明array的确要比hash table快一点（80+ms vs 110+ms）。无论input string是不是只包含小写字母，都可以用array来替代hash table，只是size大点小点的关系。**

**之前onsite面试的时候就已经被面试官不止一次的指出这个问题，能用array的时候就别用hash table.**

import java.util.\*;

public class Solution {

// 这道题给了我们一个字符串str，和一个整数k，让我们对字符串str重新排序，使得其中相同的字符之间的距离不小于k，

// 这道题的难度标为Hard，看来不是省油的灯。的确，这道题的解法用到了哈希表，堆，和贪婪算法。这道题我最开始想

// 的算法没有通过OJ的大集合超时了，下面的方法是参考网上大神的解法，发现十分的巧妙。我们需要一个哈希表来建立

// 字符和其出现次数之间的映射，然后需要一个堆来保存这每一对映射，按照出现次数来排序。然后如果堆不为空我们就

// 开始循环，我们找出k和str长度之间的较小值，然后从0遍历到这个较小值，对于每个遍历到的值，如果此时堆为空了，

// 说明此位置没法填入字符了，返回空字符串，否则我们从堆顶取出一对映射，然后把字母加入结果res中，此时映射的

// 个数减1，如果减1后的个数仍大于0，则我们将此映射加入临时集合v中，同时str的个数len减1，遍历完一次，

// 我们把临时集合中的映射对由加入堆中，参见代码如下：

public String rearrangeString(String str, int k) {

// Initialize the counter for each character

int[] freq = new int[26];

for(char c : str.toCharArray()) {

freq[c - 'a']++;

}

// Sort the chars by frequency, if same frequency then compare alphabetical order

PriorityQueue<int[]> maxPQ = new PriorityQueue<>((a, b) -> a[1] == b[1] ? a[0] - b[0] : b[1] - a[1]);

for(int i = 0; i < 26; i++) {

if(freq[i] > 0) {

maxPQ.offer(new int[]{i, freq[i]});

}

}

StringBuilder sb = new StringBuilder();

int len = str.length();

while(!maxPQ.isEmpty()) {

// 为何需要定义临时list ？

// 所有的剩余字符(全部减掉1个频次)都必须同时在下一轮中加入，而不是在本轮中再次加入

// 举个例子aaabbcc，a->3，b->2，c->2，第一轮过后a->2，b->2，c->2，如果直接

// 放入加入频次减1的a会导致a依然排在b，c之前，而根据均匀使用的原则，我们是希望先把

// a，b，c先分别使用一次的，而不是怼着a用，即希望重新排序时得到的是abc...而非

// aabc...，所以可以使用一个临时list来保存本轮已经均匀使用后剩下的所有字母

List<int[]> tmp = new ArrayList<>();

// 为何需要在k和len之间的大小 ？

// 为了对付aaadbbcc -> abc|abc|ad中最后的ad部分，此时k = 3,

// 但是len变为2，len小于3，必须按照len的长度而不再是k来填充

int sectionLen = Math.min(len, k);

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

if(maxPQ.isEmpty()) {

return "";

}

int[] pair = maxPQ.poll();

sb.append((char)(pair[0] + 'a'));

freq[pair[0]]--;

if(freq[pair[0]] > 0) {

tmp.add(new int[] {pair[0], freq[pair[0]]});

}

len--;

}

// 在前一轮对sectionLen个位置的填充结束后利用之前定义的临时list中收集的剩余字符进行新一轮填充

for(int[] t : tmp) {

maxPQ.offer(t);

}

}

return sb.toString();

}

public static void main(String[] args) {

Solution so = new Solution();

String s = "aaadbbcc";

//String s = "aabbcc";

//String s = "aaaabbcc";

String result = so.rearrangeString(s, 3);

System.out.println(result);

}

}

Time Complexity:O(nlog26) => O(n), 在这题heap存储的元素为26，因为所有字符都是lowercase

Space Complexity:O(n)

**Refer to**

<https://www.youtube.com/watch?v=28ASDBKFTxw>

**Refer to**

<https://grandyang.com/leetcode/358/>

这道题给了我们一个字符串str，和一个整数k，让我们对字符串str重新排序，使得其中相同的字符之间的距离不小于k，这道题的难度标为Hard，看来不是省油的灯。的确，这道题的解法用到了哈希表，堆，和贪婪算法。这道题我最开始想的算法没有通过OJ的大集合超时了，下面的方法是参考网上大神的解法，发现十分的巧妙。我们需要一个哈希表来建立字符和其出现次数之间的映射，然后需要一个堆来保存这每一对映射，按照出现次数来排序。然后如果堆不为空我们就开始循环，我们找出k和str长度之间的较小值，然后从0遍历到这个较小值，对于每个遍历到的值，如果此时堆为空了，说明此位置没法填入字符了，返回空字符串，否则我们从堆顶取出一对映射，然后把字母加入结果res中，此时映射的个数减1，如果减1后的个数仍大于0，则我们将此映射加入临时集合v中，同时str的个数len减1，遍历完一次，我们把临时集合中的映射对由加入堆中，参见代码如下：

class Solution {

public:

string rearrangeString(string str, int k) {

if (k == 0) return str;

string res;

int len = (int)str.size();

unordered\_map<char, int> m;

priority\_queue<pair<int, char>> q;

for (auto a : str) ++m[a];

for (auto it = m.begin(); it != m.end(); ++it) {

q.push({it->second, it->first});

}

while (!q.empty()) {

vector<pair<int, int>> v;

int cnt = min(k, len);

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

if (q.empty()) return "";

auto t = q.top(); q.pop();

res.push\_back(t.second);

if (--t.first > 0) v.push\_back(t);

--len;

}

for (auto a : v) q.push(a);

}

return res;

}

};

**Refer to**

<https://tenderleo.gitbooks.io/leetcode-solutions-/content/GoogleHard/358.html>

public class Solution {

public String rearrangeString(String str, int k) {

if(k == 0) return str;

int len = str.length();

Map<Character, Integer> counts = new HashMap<>();

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

char ch = str.charAt(i);

int n =1;

if(counts.containsKey(ch)){

n = counts.get(ch)+1;

}

counts.put(ch, n);

}

PriorityQueue<Pair> pq = new PriorityQueue<>(10, new Comparator<Pair>(){

@Override

public int compare(Pair p1, Pair p2){

if(p1.cnt != p2.cnt) return p2.cnt - p1.cnt;

else return p2.ch - p1.ch; // to ensure the order of the chars with same count, they should show up in same order.

}

});

for(Map.Entry<Character, Integer> entry : counts.entrySet()){

pq.offer(new Pair(entry.getKey(), entry.getValue()));// pick the most show-up char first.

}

StringBuilder sb = new StringBuilder();

while(!pq.isEmpty()){

List<Pair> tmp = new ArrayList<>();// this is avoid you pick up same char in the same k-segment.

int d = Math.min(k, len);

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

if(pq.isEmpty()) return "";

Pair p = pq.poll();

sb.append(p.ch);

if(--p.cnt > 0) tmp.add(p);

len--;

}

for(Pair p : tmp) pq.offer(p);

}

return sb.toString();

}

class Pair{

char ch;

int cnt;

Pair(char c, int t){

ch = c;

cnt = t;

}

};

}