# LeetCode Training Day 26 Scheduling

Scheduling problems are part of Greedy. The idea is to pick up a job with most remaining count and schedule it, but you should follow a limitation, after certain number such job is scheduled, you need to cool it down without scheduling it in the next k units of time.

## 358. Rearrange String k Distance Apart

Hard

Given a string s and an integer k, rearrange s such that the same characters are **at least** distance k from each other. 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 a distance of 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 a distance of 2 from each other.

**Constraints:**

* 1 <= s.length <= 3 \* 105
* s consists of only lowercase English letters.
* 0 <= k <= s.length

### Analysis:

On each round you can scan 26 characters, select the one with most count, however when you select a character you need to assign a next start position as i + k, i is the current index in result, k is the waiting interval, it means that until the result index become i + k, you cannot select this character again.

At very beginning, every character can be started at 0.

We only have 26 characters, simply loop 26 in every round. Use heap also fine, but not necessarily.

/// <summary>

/// Leet code #358. Rearrange String k Distance Apart

///

/// 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:

/// s = "aabbcc", k = 3

/// Result: "abcabc"

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

///

/// Example 2:

/// s = "aaabc", k = 3

/// Answer: ""

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

///

/// Example 3:

/// s = "aaadbbcc", k = 2

/// Answer: "abacabcd"

/// Another possible answer is: "abcabcda"

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

/// </summary>

string LeetCodeGreedy::rearrangeString(string s, int k)

{

vector<pair<int, int>> char\_map(26, { 0, -1 });

string result(s.size(), 0);

for (char ch : s)

{

char\_map[ch - 'a'].first++;

}

for (size\_t i = 0; i < s.size(); i++)

{

pair<int, int> max\_pos = make\_pair(0, -1);

for (int j = 0; j < 26; j++)

{

if (char\_map[j].first > max\_pos.first && char\_map[j].second <= (int)i)

{

max\_pos = make\_pair(char\_map[j].first, j);

}

}

if (max\_pos.second == -1) return "";

result[i] = max\_pos.second + 'a';

char\_map[max\_pos.second].first--;

char\_map[max\_pos.second].second = i + k;

}

return result;

}

## 621. Task Scheduler

Medium

Given a characters array tasks, representing the tasks a CPU needs to do, where each letter represents a different task. Tasks could be done in any order. Each task is done in one unit of time. For each unit of time, the CPU could complete either one task or just be idle.

However, there is a non-negative integer n that represents the cooldown period between two **same tasks** (the same letter in the array), that is that there must be at least n units of time between any two same tasks.

Return *the least number of units of times that the CPU will take to finish all the given tasks*.

**Example 1:**

**Input:** tasks = ["A","A","A","B","B","B"], n = 2

**Output:** 8

**Explanation:**

A -> B -> idle -> A -> B -> idle -> A -> B

There is at least 2 units of time between any two same tasks.

**Example 2:**

**Input:** tasks = ["A","A","A","B","B","B"], n = 0

**Output:** 6

**Explanation:** On this case any permutation of size 6 would work since n = 0.

["A","A","A","B","B","B"]

["A","B","A","B","A","B"]

["B","B","B","A","A","A"]

...

And so on.

**Example 3:**

**Input:** tasks = ["A","A","A","A","A","A","B","C","D","E","F","G"], n = 2

**Output:** 16

**Explanation:**

One possible solution is

A -> B -> C -> A -> D -> E -> A -> F -> G -> A -> idle -> idle -> A -> idle -> idle -> A

**Constraints:**

* 1 <= task.length <= 104
* tasks[i] is upper-case English letter.
* The integer n is in the range [0, 100].

### Analysis:

Schedule the task with most remaining count if the start position is less than or equal to current index in result. If no task can be assigned, set idle.

Schedule a task and put next starting index as i + k, at very beginning every task can start at zero.

/// <summary>

/// Leet code #621. Task Scheduler

///

/// Given a char array representing tasks CPU need to do. It contains

/// capital letters A to Z where different letters represent different

/// tasks. Tasks could be done without original order. Each task could

/// be done in one interval. For each interval, CPU could finish one

/// task or just be idle.

/// However, there is a non-negative cooling interval n that means

/// between two same tasks, there must be at least n intervals that

/// CPU are doing different tasks or just be idle.

/// You need to return the least number of intervals the CPU will take

/// to finish all the given tasks.

///

/// Example 1:

/// Input: tasks = ['A','A','A','B','B','B'], n = 2

/// Output: 8

/// Explanation: A -> B -> idle -> A -> B -> idle -> A -> B.

///

/// Note:

/// 1.The number of tasks is in the range [1, 10000].

/// 2.The integer n is in the range [0, 100].

/// </summary>

int LeetCodeGreedy::leastInterval(vector<char>& tasks, int n)

{

int result = 0;

unordered\_map<char, int> task\_count;

priority\_queue<pair<int, int>> task\_queue;

for (char task : tasks) task\_count[task]++;

for (auto &itr : task\_count)

{

task\_queue.push(make\_pair(0, itr.second));

}

while (!task\_queue.empty())

{

result++;

// get top task;

pair<int, int> task = task\_queue.top();

// if closed task still not ready to schedule, we push a idle;

if (-task.first < result)

{

task\_queue.pop();

task.second--;

if (task.second != 0)

{

task\_queue.push(make\_pair(task.first - n - 1, task.second));

}

}

}

return result;

}

## 1405. Longest Happy String

Medium

A string s is called **happy** if it satisfies the following conditions:

* s only contains the letters 'a', 'b', and 'c'.
* s does not contain any of "aaa", "bbb", or "ccc" as a substring.
* s contains **at most** a occurrences of the letter 'a'.
* s contains **at most** b occurrences of the letter 'b'.
* s contains **at most** c occurrences of the letter 'c'.

Given three integers a, b, and c, return *the****longest possible happy****string*. If there are multiple longest happy strings, return *any of them*. If there is no such string, return *the empty string*"".

A **substring** is a contiguous sequence of characters within a string.

**Example 1:**

**Input:** a = 1, b = 1, c = 7

**Output:** "ccaccbcc"

**Explanation:** "ccbccacc" would also be a correct answer.

**Example 2:**

**Input:** a = 7, b = 1, c = 0

**Output:** "aabaa"

**Explanation:** It is the only correct answer in this case.

**Constraints:**

* 0 <= a, b, c <= 100
* a + b + c > 0

### Analysis:

Schedule most frequent character if no violation.

/// <summary>

/// Leet code #1405. Longest Happy String

///

/// Medium

///

/// A string is called happy if it does not have any of the strings

/// 'aaa', 'bbb' or 'ccc' as a substring.

///

/// Given three integers a, b and c, return any string s, which satisfies

/// following conditions:

///

/// s is happy and longest possible.

/// s contains at most a occurrences of the letter 'a', at most b

/// occurrences of the letter 'b' and at most c occurrences of the

/// letter 'c'.

/// s will only contain 'a', 'b' and 'c' letters.

/// If there is no such string s return the empty string "".

///

/// Example 1:

/// Input: a = 1, b = 1, c = 7

/// Output: "ccaccbcc"

/// Explanation: "ccbccacc" would also be a correct answer.

///

/// Example 2:

/// Input: a = 2, b = 2, c = 1

/// Output: "aabbc"

///

/// Example 3:

/// Input: a = 7, b = 1, c = 0

/// Output: "aabaa"

/// Explanation: It's the only correct answer in this case.

///

/// Constraints:

/// 1. 0 <= a, b, c <= 100

/// 2. a + b + c > 0

/// </summary>

string LeetCodeGreedy::longestDiverseString(int a, int b, int c)

{

vector<int> char\_count = { a, b, c };

string result;

bool dup = false;

for (int i = 0; i < a + b + c; i++)

{

int index = -1;

for (size\_t j = 0; j < 3; j++)

{

if (char\_count[j] == 0) continue;

if ((index == -1) || (char\_count[j] > char\_count[index]))

{

if (result.empty() || (result.back() != 'a' + j) || (!dup))

{

index = j;

}

}

}

if (index == -1) break;

char ch = 'a' + index;

if (!result.empty() && result.back() == ch) dup = true;

else dup = false;

char\_count[ch - 'a']--;

result.push\_back(ch);

}

return result;

}

## 2182. Construct String With Repeat Limit

Medium

You are given a string s and an integer repeatLimit. Construct a new string repeatLimitedString using the characters of s such that no letter appears **more than** repeatLimit times **in a row**. You do **not** have to use all characters from s.

Return *the****lexicographically largest***repeatLimitedString *possible*.

A string a is **lexicographically larger** than a string b if in the first position where a and b differ, string a has a letter that appears later in the alphabet than the corresponding letter in b. If the first min(a.length, b.length) characters do not differ, then the longer string is the lexicographically larger one.

**Example 1:**

**Input:** s = "cczazcc", repeatLimit = 3

**Output:** "zzcccac"

**Explanation:** We use all of the characters from s to construct the repeatLimitedString "zzcccac".

The letter 'a' appears at most 1 time in a row.

The letter 'c' appears at most 3 times in a row.

The letter 'z' appears at most 2 times in a row.

Hence, no letter appears more than repeatLimit times in a row and the string is a valid repeatLimitedString.

The string is the lexicographically largest repeatLimitedString possible so we return "zzcccac".

Note that the string "zzcccca" is lexicographically larger but the letter 'c' appears more than 3 times in a row, so it is not a valid repeatLimitedString.

**Example 2:**

**Input:** s = "aababab", repeatLimit = 2

**Output:** "bbabaa"

**Explanation:** We use only some of the characters from s to construct the repeatLimitedString "bbabaa".

The letter 'a' appears at most 2 times in a row.

The letter 'b' appears at most 2 times in a row.

Hence, no letter appears more than repeatLimit times in a row and the string is a valid repeatLimitedString.

The string is the lexicographically largest repeatLimitedString possible so we return "bbabaa".

Note that the string "bbabaaa" is lexicographically larger but the letter 'a' appears more than 2 times in a row, so it is not a valid repeatLimitedString.

**Constraints:**

* 1 <= repeatLimit <= s.length <= 105
* s consists of lowercase English letters.

### Analysis:

Schedule highest character until reach limit.

/// <summary>

/// Leet Code 2158. Amount of New Area Painted Each Day

///

/// Hard

///

/// There is a long and thin painting that can be represented by a number

/// line. You are given a 0-indexed 2D integer array paint of length n,

/// where paint[i] = [starti, endi]. This means that on the ith day you

/// need to paint the area between starti and endi.

///

/// Painting the same area multiple times will create an uneven painting

/// so you only want to paint each area of the painting at most once.

///

/// Return an integer array worklog of length n, where worklog[i] is the

/// amount of new area that you painted on the ith day.

///

/// Example 1:

/// Input: paint = [[1,4],[4,7],[5,8]]

/// Output: [3,3,1]

/// Explanation:

/// On day 0, paint everything between 1 and 4.

/// The amount of new area painted on day 0 is 4 - 1 = 3.

/// On day 1, paint everything between 4 and 7.

/// The amount of new area painted on day 1 is 7 - 4 = 3.

/// On day 2, paint everything between 7 and 8.

/// Everything between 5 and 7 was already painted on day 1.

/// The amount of new area painted on day 2 is 8 - 7 = 1.

///

/// Example 2:

/// Input: paint = [[1,4],[5,8],[4,7]]

/// Output: [3,3,1]

/// Explanation:

/// On day 0, paint everything between 1 and 4.

/// The amount of new area painted on day 0 is 4 - 1 = 3.

/// On day 1, paint everything between 5 and 8.

/// The amount of new area painted on day 1 is 8 - 5 = 3.

/// On day 2, paint everything between 4 and 5.

/// Everything between 5 and 7 was already painted on day 1.

/// The amount of new area painted on day 2 is 5 - 4 = 1.

///

/// Example 3:

/// Input: paint = [[1,5],[2,4]]

/// Output: [4,0]

/// Explanation:

/// On day 0, paint everything between 1 and 5.

/// The amount of new area painted on day 0 is 5 - 1 = 4.

/// On day 1, paint nothing because everything between 2 and 4 was already

/// painted on day 0.

/// The amount of new area painted on day 1 is 0.

///

/// Constraints:

/// 1. 1 <= paint.length <= 10^5

/// 2. paint[i].length == 2

/// 3. 0 <= starti < endi <= 5 \* 10^4

/// </summary>

vector<int> LeetCodeGreedy::amountPainted(vector<vector<int>>& paint)

{

map<int, int> painted;

painted[INT\_MAX] = 0;

vector<int> result(paint.size());

for (size\_t i = 0; i < paint.size(); i++)

{

int length = 0;

int prev = paint[i][0];

auto itr = painted.lower\_bound(paint[i][0]);

int start = itr->second;

while (itr->first <= paint[i][1])

{

if (itr->second == 0)

{

length += itr->first - prev;

}

prev = itr->first;

auto next\_itr = next(itr);

painted.erase(itr);

itr = next\_itr;

}

if (itr->second == 0)

{

length += paint[i][1] - prev;

painted[paint[i][1]] = 1;

}

if (start == 0)

{

painted[paint[i][0]] = 0;

}

result[i] = length;

}

return result;

}