<https://leetcode.com/problems/task-scheduler/description/>

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]

**Attempt 1: 2023-11-05**

**Solution 1: Priority Queue + Harsh Table (30 min)**

**Style 1: Need StringBuilder, section length always fixed as n + 1 and need trim (same as L358/P14.11.Rearrange String K Distance Apart)**

class Solution {

public int leastInterval(char[] tasks, int n) {

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

for(char c : tasks) {

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

}

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();

// The strategy is we always try to fill in the fixed

// length('sectionLen') section with different characters,

// the fixed length based on given n + 1, is the CPU cycle

// length, if n is the cooldown period then after a task A

// there will be n more tasks

int sectionLen = n + 1;

while(!maxPQ.isEmpty()) {

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

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

// If no more character can be used to fill in the fixed

// length section, we have to fill in 'space' instead

if(maxPQ.isEmpty()) {

// Use 'space' instead of '#' to work with trim() method

//sb.append('#');

sb.append(" ");

} else {

char c = maxPQ.poll();

sb.append(c);

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

// Same strategy as L358/P14.11.Rearrange String K Distance Apart

// Use a temporary list to store the remained characters(unique)

// for next section to fill in

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

tmp.add(c);

}

}

}

// Add all remain characters(unique) value back to Priority Queue

// for next section to fill in

for(char c : tmp) {

maxPQ.offer(c);

}

}

// Have to trim the redundant 'spaces' at the end of result string

// e.g

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

// One possible solution is

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

// If no trim() after above logic, the string will be:

// ABCADEAFGA..A..A.. -> the redundant 2 spaces(.) at the end have to trim

return sb.toString().trim().length();

}

}

Time Complexity: O(N), not O(N\*logN) because logN for the PQ is log(26) since we are only pushing 26 tasks at worst. so it is constant

Space Complexity: O(N)

**Style 2: No need StringBuilder, section length will change and no need trim**

class Solution {

public int leastInterval(char[] tasks, int n) {

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

for(char c : tasks) {

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

}

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);

}

int totalLen = 0;

while(!maxPQ.isEmpty()) {

// The strategy is we always try to fill in the fixed

// length('sectionLen') section with different characters,

// the fixed length based on given n + 1, is the CPU cycle

// length, if n is the cooldown period then after a task A

// there will be n more tasks

int sectionLen = n + 1;

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

while(sectionLen > 0 && !maxPQ.isEmpty()) {

char c = maxPQ.poll();

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

totalLen++;

sectionLen--;

// Same strategy as L358/P14.11.Rearrange String K Distance Apart

// Use a temporary list to store the remained characters(unique)

// for next section to fill in

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

tmp.add(c);

}

}

// Add all remain characters(unique) value back to Priority Queue

// for next section to fill in

for(char c : tmp) {

maxPQ.offer(c);

}

// If the priority queue is empty than all tasks are completed and we don't need to include the idle time

if(maxPQ.isEmpty()) {

break;

}

// This counts the idle time

totalLen += sectionLen;

}

return totalLen;

}

}

**Refer to**

<https://leetcode.com/problems/task-scheduler/solutions/3280549/full-explanation-using-priority-queue-and-formula-based-approach/>

# **Intuition**

It took me a lot of time to figure this one out. The intuitive solution is to first count the occurence of each task and store it in a priority queue so that the highest frequency task can be done first and then other tasks can follow.But, we have to keep the cooldown period in mind.

# **Approach**

## **1. Using priority queue**

First count the number of occurrences of each task and store that in a map/vector.

Then push the count into a priority queue, so that the maximum frequency task can be accessed and completed first.

Then until all tasks are completed (i.e the priority queue is not empty):

initialize the cycle length as n+1. n is the cooldown period so the cycle will be of n+1 length.

Example: for ['A','A','A','B','B'] and n=2,

the tasks can occur in the following manner:

[A B idle]->[A B idle]->[A]. See here each cycle is n+1 length long, only then A can repeat itself.

for all elements in the priority queue, until the cycle length is exhausted, pop the elements out of the queue and if the task is occurring more than once then add it to the remaining array (which stores the remaining tasks).

This means that we are completing that task once in this cycle, so keep counting the time.

Then, add the occurrence of tasks back to the priority queue.

Add the idle time into the time count.

Idle time is the time that was needed in the cycle because no task was available. It is the remaining cycle length in our algorithm. Idle time should be only added if the priority queue is empty (i.e all tasks are completed).

class Solution {

public:

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

priority\_queue<int> pq;

vector<int>mp(26,0);

for(char i:tasks){

mp[i-'A']++; // count the number of times a task needs to be done

}

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

if(mp[i])

pq.push(mp[i]);

}

int time=0; // stores the total time taken

while(!pq.empty()){

vector<int>remain;

int cycle=n+1; // n+1 is the CPU cycle length, if n is the cooldown period then after a task A there will be n more tasks. Hence n+1.

while(cycle and !pq.empty()){

int max\_freq=pq.top(); // the task at the top should be first assigned the CPU as it has highest frequency

pq.pop();

if(max\_freq>1){ // task with more than one occurrence, the next occurrence will be done in the next cycle

remain.push\_back(max\_freq-1); // add it to remaining task list

}

++time;

--cycle;

}

for(int count:remain){

pq.push(count);

}

if(pq.empty())break; // if the priority queue is empty than all tasks are completed and we don't need to include the idle time

time+=cycle; // this counts the idle time

}

return time;

}

};

**Solution 2: Greedy + Math (30 min)**

class Solution {

public int leastInterval(char[] tasks, int n) {

int maxFreqCount = 0;

int maxFreq = 0;

int[] freq = new int[26];

for(char c : tasks) {

freq[c - 'A']++;

if(freq[c - 'A'] == maxFreq) {

maxFreqCount++;

} else if(freq[c - 'A'] > maxFreq) {

maxFreq = freq[c - 'A'];

maxFreqCount = 1;

}

}

int cooldown\_interval\_count = maxFreq - 1;

int cooldown\_interval\_len = n + 1 - maxFreqCount;

int total\_empty\_slots = cooldown\_interval\_count \* cooldown\_interval\_len;

int available\_tasks = tasks.length - maxFreq \* maxFreqCount;

int idles = Math.max(0, total\_empty\_slots - available\_tasks);

return idles + tasks.length;

}

}

Time Complexity: O(N)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/task-scheduler/solutions/104500/Java-O(n)-time-O(1)-space-1-pass-no-sorting-solution-with-detailed-explanation/>

The key is to find out how many idles do we need.Let's first look at how to arrange them. it's not hard to figure out that we can do a "greedy arrangement": always arrange task with most frequency first.E.g. we have following tasks : 3 A, 2 B, 1 C. and we have n = 2. According to what we have above, we should first arrange A, and then B and C. Imagine there are "slots" and we need to arrange tasks by putting them into "slots". Then A should be put into slot 0, 3, 6 since we need to have at least n = 2 other tasks between two A. After A put into slots, it looks like this:

A ? ? A ? ? A

"?" is "empty" slots.

Now we can use the same way to arrange B and C. The finished schedule should look like this:

A B C A B # A

"#" is idle

Now we have a way to arrange tasks. But the problem only asks for number of CPU intervals, so we don't need actually arrange them. Instead we only need to get the total idles we need and the answer to problem is just number of idles + number of tasks. Same example: 3 A, 2 B, 1 C, n = 2. After arranging A, we have:

A ? ? A ? ? A

We can see that A separated slots into (count(A) - 1) = 2 parts, each part has length n. With the fact that A is the task with most frequency, it should need more idles than any other tasks. **In this case if we can get how many idles we need to arrange A, we will also get number of idles needed to arrange all tasks.** Calculating this is not hard, we first get number of parts separated by A: partCount = count(A) - 1; then we can know number of empty slots: emptySlots = partCount \* n; we can also get how many tasks we have to put into those slots: availableTasks = tasks.length - count(A). Now if we have emptySlots > availableTasks which means we have not enough tasks available to fill all empty slots, we must fill them with idles. Thus we have **idles = max(0, emptySlots - availableTasks);**

Almost done. One special case: what if there are more than one task with most frequency? OK, let's look at another example: 3 A 3 B 2 C 1 D, n = 3

Similarly we arrange A first:

A ? ? ? A ? ? ? A

Now it's time to arrange B, we find that we have to arrange B like this:

A B ? ? A B ? ? A B

we need to put every B right after each A. Let's look at this in another way, think of sequence "A B" as a special task "X", then we got:

X ? ? X ? ? X

Comparing to what we have after arranging A:

A ? ? ? A ? ? ? A

The only changes are length of each parts (from 3 to 2) and available tasks. By this we can get more general equations:

**partCount = count(A) - 1**

**emptySlots = partCount \* (n - (count of tasks with most frequency - 1))**

**availableTasks = tasks.length - count(A) \* count of tasks with most frequency**

**idles = max(0, emptySlots - availableTasks)**

**result = tasks.length + idles**

What if we have more than n tasks with most frequency and we got emptySlot negative? Like 3A, 3B, 3C, 3D, 3E, n = 2. In this case seems like we can't put all B C S inside slots since we only have n = 2.Well partCount is actually the "minimum" length of each part required for arranging A. You can always make the length of part longer.

E.g. 3A, 3B, 3C, 3D, 2E, n = 2.

You can always first arrange A, B, C, D as:

A B C D | A B C D | A B C D

in this case you have already met the "minimum" length requirement for each part (n = 2), you can always put more tasks in each part if you like: e.g.

A B C D E | A B C D E | A B C D

emptySlots < 0 means you have already got enough tasks to fill in each part to make arranged tasks valid. But as I said you can always put more tasks in each part once you met the "minimum" requirement.

To get count(A) and count of tasks with most frequency, we need to go through inputs and calculate counts for each distinct char. This is O(n) time and O(26) space since we only handle upper case letters. All other operations are O(1) time O(1) space which give us total time complexity of O(n) and space O(1)

public class Solution {

public int leastInterval(char[] tasks, int n) {

int[] counter = new int[26];

int max = 0;

int maxCount = 0;

for(char task : tasks) {

counter[task - 'A']++;

if(max == counter[task - 'A']) {

maxCount++;

}

else if(max < counter[task - 'A']) {

max = counter[task - 'A'];

maxCount = 1;

}

}

int partCount = max - 1;

int partLength = n - (maxCount - 1);

int emptySlots = partCount \* partLength;

int availableTasks = tasks.length - max \* maxCount;

int idles = Math.max(0, emptySlots - availableTasks);

return tasks.length + idles;

}

}