621. Task Scheduler Heap (Priority Queue) Medium Hash Table Counting) Sorting <u>Array</u>

Problem Description

by a CPU in any sequence, where each task takes one unit of time to complete. However, there is a constraint: the CPU must wait for a cooldown period of n units of time between performing the same type of task. During the cooldown period, the CPU can either perform a different task or remain idle if no other tasks are available. The goal is to find the minimum number of units of time the CPU will require to complete all tasks while adhering to the cooldown constraint.

In this problem, we are given an array of tasks where each type of task is represented by a letter. These tasks can be performed

Intuition

The key problem is to design a task schedule that minimizes idle time while respecting the cooldown constraint between the same tasks. To find the solution, we must first identify the task that occurs the most frequently, as it dictates the minimum time needed to complete all tasks.

Our solution approach begins by using a counter to count the occurrence of each task and find the maximum frequency (x). The

idea is to first schedule the most frequent tasks, ensuring that there are n units of idle time between them. Once these are

scheduled, we can fill in the remaining time with different tasks. We calculate idle time using (x - 1) * (n + 1), where (x - 1) represents the number of gaps between the most frequent tasks, and (n + 1) represents the size of each gap including the task itself.

Then, we must account for tasks that occur as frequently as the most frequent task. We do this by adding s, which is the number of tasks that have the maximum frequency. This ensures that if there are multiple tasks with the same maximum frequency, we

have enough time slots to accommodate them without additional idle time.

Finally, we have two scenarios to consider:

The calculated time is greater than the total number of tasks. This means that even after scheduling the most frequent tasks

with their cooldowns, we are able to accommodate all other tasks within the idle time slots. In this case, the calculated time is

The calculated time is less than or equal to the number of tasks. This means that the tasks are diverse enough that we do not

each one needs to be scheduled.

s = sum(v == x for v in cnt.values())

appears three times, B twice, and C only once.

The length of the original task list is 6.

The scheduler might operate like this:

Place task A

Place task A

• Idle slot

Idle slot

slots account for the time between completing instances of task A.

Calculating Total Time: Lastly, we must determine the actual total time:

x = max(cnt.values())

the result.

- need idle time, and the total number of tasks is the minimum time needed to complete them all. In this case, the number of tasks is the result. The max function is used to choose the larger of the two scenarios above, which gives us the minimum units of time to finish all
- the tasks. The solution approach involves the following steps detailed with the algorithms, data structures, and patterns used: Counting Task Frequencies: Using Python's Counter class, we construct a frequency map which counts the number of

occurrences for each task type. This helps us to quickly determine which tasks are the most frequent and how many times

cnt = Counter(tasks)

backbone of our scheduling strategy. The maximum value in our frequency map gives us this information.

unit of the cooldown period of the max frequency task without violating the cooldown requirement.

Identifying the Maximum Frequency: We need to know the frequency of the most common task because it will form the

Calculating Idle Time Slots: Our goal is to minimize the CPU's idle time while respecting the cooldown period. The number of idle slots needed is based on the maximum frequency and the cooldown period. We calculate this with (x - 1) * (n + 1), where (x - 1) gives us the number of idle slots needed, assuming all slots are filled with the most frequent task and its subsequent cooldown periods. Adding 1 accounts for the most frequent task itself being placed in the schedule.

Determining the Number of Maximum Frequent Tasks: Since there might be multiple tasks that have the same maximum

frequency, we calculate how many there are because they won't contribute to idle time—they can all be placed in the final

Calculating Total Time: Finally, we need to determine the total time which is the maximum of the length of the original task list or the calculated idle time slots plus the slots needed for the most frequent tasks. return max(len(tasks), (x - 1) * (n + 1) + s)

Using this strategy efficiently arranges tasks and minimizes the idle time. Thus, the max function is of particular importance here

because it ensures that if no idle time is needed (i.e., there are enough less frequent tasks to fit in the cooldown periods), the

total number of tasks is used. Otherwise, the calculated idle time and task slots indicate the CPU's total operating time.

Example Walkthrough

In this example, let's consider an array of tasks represented as ['A', 'A', 'A', 'B', 'B', 'C'] and a cooldown period n = 2.

Counting Task Frequencies: We count the occurrences of each task which gives us {'A': 3, 'B': 2, 'C': 1}. Task A

Determining the Number of Maximum Frequent Tasks: In this case, there is only one task (A) that has the maximum

Calculating Idle Time Slots: With a cooldown period n = 2, we have (3 - 1) * (2 + 1) = 2 * 3 = 6 idle slots at first. These

frequency, so s = 1.

Identifying the Maximum Frequency: The most common task is A, with a maximum frequency of x = 3.

- Calculated time considering idle slots and the most frequent task is 6 (idle slots calculated in step 3) + 1 (task A itself) = 7.We take the maximum: max(6, 7) = 7.
- Idle slot Idle slot
- Place task A Place tasks B and C (in the first two idle slots)

def leastInterval(self, tasks: List[str], n: int) -> int:

Count how many tasks have the maximum frequency

min_length = idle_time + max_freq_tasks_count

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

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

for (char task : tasks) {

int tasksWithMaxCount = 0;

for (int count : taskCount) {

if (count == maxCount) {

// Calculate the least interval

++tasksWithMaxCount;

// Count the tasks and find out the task with the maximum count

task -= 'A'; // Convert char to an index between 0 and 25

++taskCount[task]; // Increment the count for this task

return max(static_cast<int>(tasks.size()), minSlotsRequired);

// Count how many tasks have the same count as maxCount

vector<int> taskCount(26); // Initialize a vector to keep count of each task

int maxCount = 0; // Variable to keep track of the maximum count of a single task

// Second part: Add the number of tasks that have the highest frequency (taskWithMaxCount)

// The result is the maximum between the actual size of tasks and the minimum slots required

int minSlotsRequired = (maxCount - 1) * (coolingPeriod + 1) + tasksWithMaxCount;

maxCount = max(maxCount, taskCount[task]); // Update maxCount if current task's count is greater

max_freq_tasks_count = sum(freq == max_freq for freq in task_counts.values())

Calculate the number of idle states needed, which is defined by the formula:

// Counts of each task where index 0 represents 'A', 1 represents 'B', and so on.

// Loop over the tasks to count them and find the task with maximum frequency.

Count the occurrences of each task

max_freq = max(task_counts.values())

 $idle_time = (max_freq - 1) * (n + 1)$

int[] taskCounts = new int[26];

int maxFrequency = 0;

// Maximum frequency among the tasks

Find the maximum frequency of any task

task_counts = Counter(tasks)

initially allocated for A. No additional idle time is needed because other tasks fill up the cooldown periods between As. In the end, the minimum number of time units the CPU requires to complete all tasks while respecting the cooldown period is 7.

Place task B (in the last idle slot)

Solution Implementation

Python from collections import Counter class Solution:

(maximum frequency of any task -1) * (n + 1) which gives the spaces between repetitions of the same task

Add the count of most frequent tasks to idle time to get minimum length of the task schedule

Thus, task A sets the backbone for the scheduling, taking up the first slot, and all other tasks are arranged around the idle times

Return the maximum of the length of tasks (if no idle time is necessary) and calculated minimum length return max(len(tasks), min_length) Java

class Solution {

```
for (char task : tasks) {
           // Convert the task from char type to an index for our count array
           int index = task - 'A';
           // Increment the count for this task
            taskCounts[index]++;
           // Update the maximum frequency
           maxFrequency = Math.max(maxFrequency, taskCounts[index]);
       // Count how many tasks have the maximum frequency
        int maxFrequencyTasks = 0;
        for (int count : taskCounts) {
            if (count == maxFrequency) {
               maxFrequencyTasks++;
       // Calculate the minimum length of the task schedule
       // Each block of tasks includes the cooldown period followed by the most frequent task itself
       // Then, add the number of tasks with maximum frequency to cover the last one without tailing idle time
        int minScheduleLength = Math.max(tasks.length, (maxFrequency -1) * (cooldown +1) + maxFrequencyTasks);
       return minScheduleLength;
C++
class Solution {
```

```
TypeScript
```

public:

```
// This function computes the least number of time units required to complete all tasks
// with a given cooling period between two same tasks.
function leastInterval(tasks: string[], coolingPeriod: number): number {
    const taskCount: number[] = new Array(26).fill(0); // Initialize an array to keep count of each task
    let maxCount: number = 0; // Variable to keep track of the maximum count of a single task
    // Count the tasks and find out the task with the maximum count
    tasks.forEach(task => {
        const index = task.charCodeAt(0) - 'A'.charCodeAt(0); // Convert char to an index between 0 and 25
        taskCount[index]++; // Increment the count for this task
        maxCount = Math.max(maxCount, taskCount[index]); // Update maxCount if current task's count is greater
    });
    // Count how many tasks have the same count as maxCount
    let tasksWithMaxCount: number = 0;
    taskCount.forEach(count => {
        if (count === maxCount) {
            tasksWithMaxCount++;
    })
    // Calculate the least interval
    // First part: Calculate the minimum slots required based on the most frequent task (maxCount - 1) times (coolingPeriod + 1)
    // Second part: Add the number of tasks that have the highest frequency (tasksWithMaxCount)
    const minSlotsRequired: number = (maxCount - 1) * (coolingPeriod + 1) + tasksWithMaxCount;
    // The result is the maximum between the actual size of the tasks and the minimum slots required
    return Math.max(tasks.length, minSlotsRequired);
```

// First part: Calculate the minimum slots required based on the most frequent task (maxCount — 1) times (coolingPeriod ¬

```
Time and Space Complexity
  The given Python function leastInterval calculates the least time it would take to finish all tasks considering a cooldown period
```

As for the space complexity:

these steps:

from collections import Counter

def leastInterval(self, tasks: List[str], n: int) -> int:

Count how many tasks have the maximum frequency

min_length = idle_time + max_freq_tasks_count

max_freq_tasks_count = sum(freq == max_freq for freq in task_counts.values())

Calculate the number of idle states needed, which is defined by the formula:

Count the occurrences of each task

max_freq = max(task_counts.values())

 $idle_time = (max_freq - 1) * (n + 1)$

return max(len(tasks), min length)

Find the maximum frequency of any task

task_counts = Counter(tasks)

class Solution:

Count each unique task and store the frequencies using Counter(tasks). The time complexity of this operation is O(T), where Trepresents the total number of tasks, since it requires iterating over each task once. Find the maximum frequency x among all tasks with max(cnt.values()). The time complexity here is O(U), where U represents the number of unique tasks.

(n intervals) between tasks that are the same. It uses Python Counter from the collections to achieve its purpose, following

(maximum frequency of any task -1) * (n + 1) which gives the spaces between repetitions of the same task

Return the maximum of the length of tasks (if no idle time is necessary) and calculated minimum length

Add the count of most frequent tasks to idle time to get minimum length of the task schedule

This is also O(U), the time needed to iterate over the unique task frequencies.

Calculate the number of tasks with the maximum frequency, using list comprehension sum(v == x for v in cnt.values()).

- Return the maximum between the actual number of tasks len(tasks) and a calculated value (x 1) * (n + 1) + s. The time complexity for this operation is constant, 0(1).
- The overall time complexity is dominated by the larger of O(T) or O(U). Given that the number of unique tasks U will always be less than or equal to the total number of tasks T, the time complexity is O(T).
- tasks. The space for storing maximum frequency x and the summation s is constant, O(1).
- Counter(tasks) constructs a hashmap to store the task frequencies, consuming O(U) space, where U is the number of unique

The temporary list comprehension used for the summation does not require additional space as it is computed inline and

summed immediately. Therefore, the space complexity of the function is O(U).

Overall, the time complexity is O(T) and the space complexity is O(U).