# 2365. Task Scheduler II

Medium Array Hash Table Simulation

## Problem Description

The problem provides us with an array `tasks`, where each element represents a type of task that needs to be completed in the order they are given. Furthermore, we are given an integer `space` which denotes the minimum number of days we must wait before we can complete a task of the same type again.

Our goal is to find the minimum number of days required to complete all tasks. It's important to note that we have the option to either:

- Complete the next task in the array, or
- Take a break if the next task can't be performed due to the space constraint.

## Intuition

The intuition behind the solution is to track when a task of a certain type can next be performed while iterating over the `tasks` array. Here's the thought process:

- Each day, we can perform a task or take a break. If we decide to complete a task, we need to make sure enough days have passed according to `space` if we previously completed the same type of task.
- A good way to track the earliest day we can perform each type of task again is by storing this information in a dictionary where the keys are task types and the values are the earliest day the task can be carried out again.
- We can initialize a counter, called `ans`, to keep track of the current day. For each task, we increment this counter by 1 to represent doing the task that day.
- Before performing a task, we check whether the current day (`ans`) is at least as large as the least day this task is allowed to be performed on again (which can be found from the dictionary).
- If we've already performed this task type and the earliest day we can perform it again is in the future (greater than `ans`), then we must fast-forward `ans` to that day.
- After completing a task, we update the dictionary entry for that task to the current day plus `space`, plus one more day to account for the completion day.
- We continue this process for each task in the tasks array. The final value of `ans` after processing all tasks will be the minimum number of days required.

## Solution Approach

The solution to this task scheduling problem involves a simple yet effective approach. Here's how the algorithm is implemented, leveraging the Python `defaultdict` data structure for efficient look-ups and updates:

- A `defaultdict` of type `int` named `day` is created. This is used to store the minimum day on which a task of a particular type can next be executed. Since it's a `defaultdict`, it will default any unset keys to `0` which is a convenient starting point for our `ans` since we need to start counting from day `1`.
- A variable `ans` is initialized to `0`. This variable represents the current day and it is incremented for each task, thus simulating the passage of each day as we perform tasks.
- We loop through each `task` in the `tasks` list with a `for` loop. The loop represents sequential task completion and ensures we respect the order of tasks as specified.
- For each task, we do the following:
  - Increment the current day (`ans`) by `1` since we're considering a task for completion on this new day.
  - The `max` function is used to identify if we should fast-forward the current day. It compares `ans` and the value of `day[task]`, meaning it checks whether we've waited enough days (according to `space`) to perform the same type of task again.
  - If the value in `day[task]` is larger (i.e., the next permitted day for this type of task is in the future), `ans` is updated to `day[task]`. Otherwise, `ans` remains the same as it just got incremented.
  - After potentially performing the task (because we can perform it today or we have fast-forwarded to a day when we can), we update the dictionary entry `day[task]` to `ans + space + 1`. This is the next day we're allowed to perform a task of this type, which is `space` days from the current day plus one for the day the task is completed.
- Once we have processed all tasks, the value of `ans` is the total number of days required to complete all tasks, as it's been incremented and fast-forwarded as necessary according to the task ordering and spacing rules. The function returns `ans`.

By utilizing the `defaultdict` to keep track of the waiting period for each task type, and by incrementing and possibly fast-forwarding `ans` on each iteration, the implementation minimizes the number of days required as per the `space` constraint. The space-time efficiency of this approach is optimal since it processes each task in a single pass and updates the dictionary entries in constant time.

### Example Walkthrough

Let's take a small example to illustrate the solution approach. Suppose the array `tasks` is `[1, 1, 2, 1]` and `space` is `2`. This means we need to wait at least 2 days before we can repeat the same task.

We initiate a `defaultdict` named `day` which will store the next available day a task can be performed. We also initiate a variable `ans` as `0` representing the current day.

Now let's walk through the tasks:

1. The current task is `1`. Since `day[1]` defaults to `0`, and `ans` is `0`, we increment `ans` to `1` and then update the `day[1]` to `ans + space + 1`, which equals `1 + 2 + 1 = 4`. Now, `day[1] = 4`.
2. The next task is again `1`. `day[1]` is `4`, but `ans` is currently `1`, so we can't perform this task today. We then fast-forward `ans` to `4` which is the next available day to perform task `1`. We complete the task, and update `day[1]` again to `4 + 2 + 1 = 7` because now the next time we can perform task `1` is after space days from day `4`.
3. Our next task is `2`. Checking `day[2]` which is `0` since this task has not been performed yet and `ans` is `4`. We can perform this task since the space constraint is met. We increment `ans` to `5` and set `day[2]` to `5 + 2 + 1 = 8`.
4. The final task is `1`. We check `day[1]` which is currently `7`. `ans` is `5`, meaning we need to fast-forward to day `7` to perform task `1`. We increment `ans` to `7` and perform the task updating `day[1]` now to `7 + 2 + 1 = 10`.

Having processed all tasks, `ans` is `7` indicating the minimum number of days required to complete all tasks given their order and the space constraint.

This walk-through illustrates how the solution methodically progresses through tasks, respecting the `space` constraint by using a dictionary to track availability and appropriately fast-forwarding the current day when needed.

## Solution Implementation

### Python

~~~
from collections import defaultdict

class Solution:
    def taskSchedulerII(self, tasks: List[int], space: int) -> int:
        # Dictionary to store the next eligible day on which a task can be performed
        next_eligible_day = defaultdict(int)
        # Initialize the current day counter
        current_day = 0

        # Loop through the given list of tasks
        for task in tasks:
            # Increment the day counter as we're performing a task each day
            current_day += 1
            # If the task has been performed before, make sure to respect the waiting period
            # Update the current day to the maximum of itself and the next eligible day for the task
            current_day = max(current_day, next_eligible_day[task])
            # Update the next eligible day for the current task by adding the space interval
            next_eligible_day[task] = current_day + space + 1

        # After processing all tasks, return the last day count as the total days required
        return current_day
```
**Note:** Before running this code, you will need to replace `List[int]` with the appropriate import from the `typing` module, l

```python
from typing import List
~~~

### Java

```
class Solution {
    public long taskSchedulerII(int[] tasks, int space) {
        // Map to store the next valid day a task can be scheduled
        Map<Integer, Long> nextValidDay = new HashMap<>();
        long currentDay = 0; // Represents the current day

        // Iterate through all tasks
        for (int task : tasks) {
            currentDay++; // Move to the next day
            // Check if we need to wait for a cooldown for the current task, and if necessary, jump to the next valid day
            currentDay = Math.max(currentDay, nextValidDay.getOrDefault(task, 0L));
            // Calculate and store the next valid day the current task can be executed based on the space (cooldown period)
            nextValidDay.put(task, currentDay + space + 1);
        }

        // The last value of currentDay will be the total time taken to complete all tasks
        return currentDay;
    }
}
```

### C++

```
#include <vector>
#include <unordered_map>

class Solution {
public:
    // Function to determine the minimum number of days required to complete all tasks.
    // Each identical task needs to be executed with at least a 'space' day gap between them.
    long long taskSchedulerII(vector<int>& tasks, int space) {
        unordered_map<int, long long> nextAvailableDay; // Maps a task to the next day it can be executed.
        long long currentDay = 0; // Tracks the current day.

        // Iterate through each task in tasks vector.
        for (int task : tasks) {
            ++currentDay; // Increment the current day by one.

            // Check if we need to wait for a task based on its space requirement.
            // If so, move the current day to the next available day for that task.
            if (nextAvailableDay.find(task) != nextAvailableDay.end()) {
                currentDay = max(currentDay, nextAvailableDay[task]);
            }

            // Update the next available day for the current task to be space days out.
            nextAvailableDay[task] = currentDay + space + 1;
        }

        // The result is the last day we finish a task.
        return currentDay;
    }
};
```

### TypeScript

```
// Function to calculate the number of days needed to complete given tasks
// with a cooldown period between tasks of the same type.
function taskSchedulerII(tasks: number[], space: number): number {
    // Map to keep track of the next available day for each task to be scheduled
    const nextAvailableDay = new Map<number, number>();
    // Variable to keep track of the current day
    let currentDay = 0;

    // Loop through each task in the given tasks array
    for (const task of tasks) {
        // Move to the next day
        currentDay++;

        // Check if there's a previously scheduled day for the current task and
        // update the current day if necessary, to respect the cooling period
        currentDay = Math.max(currentDay, nextAvailableDay.get(task) ?? 0);

        // Set the next available day for this task to current day plus space days
        nextAvailableDay.set(task, currentDay + space + 1);
    }

    // Return the total number of days needed to complete the tasks
    return currentDay;
}
```

~~~
from collections import defaultdict

class Solution:
    def taskSchedulerII(self, tasks: List[int], space: int) -> int:
        # Dictionary to store the next eligible day on which a task can be performed
        next_eligible_day = defaultdict(int)
        # Initialize the current day counter
        current_day = 0

        # Loop through the given list of tasks
        for task in tasks:
            # Increment the day counter as we're performing a task each day
            current_day += 1
            # If the task has been performed before, make sure to respect the waiting period
            # Update the current day to the maximum of itself and the next eligible day for the task
            current_day = max(current_day, next_eligible_day[task])
            # Update the next eligible day for the current task by adding the space interval
            next_eligible_day[task] = current_day + space + 1

        # After processing all tasks, return the last day count as the total days required
        return current_day
```
**Note:** Before running this code, you will need to replace `List[int]` with the appropriate import from the `typing` module, like t

```python
from typing import List
~~~

## Time and Space Complexity

The given Python code defines a function `taskSchedulerII` which takes a list of tasks and a cooldown period (`space`) and returns the total number of days needed to complete all tasks following the cooldown restriction for each task.

### Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of tasks. This is because there is a single for-loop iterating over each task exactly once. Inside the for-loop, dictionary operations (access and assignment) are utilized, which run in average-case `O(1)` time. Hence, the loop's operations do not depend on the size of the input beyond the iteration over the tasks, resulting in a linear time complexity.

The key line to note for the time complexity inside the loop is:

```
ans = max(ans, day[task])
```

This is a constant-time operation and does not change the overall linear time complexity of the algorithm.

### Space Complexity

The space complexity of the code is `O(m)`, where `m` is the number of unique tasks. A dictionary named `day` is used to store the next available day for each unique task after respecting the cooldown period. In the worst case, if all tasks are unique, the dictionary will have an entry for each one, resulting in space complexity proportional to the number of unique tasks.

The key part of the code for analyzing space complexity is the dictionary:

```
day = defaultdict(int)
```

The `day` dictionary only ever stores at the most one key-value pair per unique task, giving us the `O(m)` space complexity.

### Conclusion

The function is efficient in terms of time complexity, operating in linear time relative to the number of tasks. It is also relatively space efficient, although the space needed can grow with the number of unique tasks.