2589. Minimum Time to Complete All Tasks

Hard | Stack | Greedy | Array | Binary Search | Sorting | Leetcode Link

Problem Description

In this task scheduling problem, we are given a list of tasks, each with a start time, end time, and a duration indicating how long the task needs to run. The computer can run these tasks simultaneously, and we have the flexibility to turn the computer on and off as needed. Our goal is to figure out the minimum amount of time the computer needs to be active (turned on) to accomplish all the tasks within their respective time ranges.

To visualize this problem, think of each task as a window of time during which it can be performed. The computer must be turned on long enough to "cover" these windows cumulatively. Since tasks can overlap in time, we want to make sure we are efficient about when we are running the computer to minimize the total on time.

Intuition

The intuition behind the solution approach is to manage the scheduling of tasks in a way that maximizes the overlap of their execution windows, hence reducing the total time the computer is on. To achieve this, we sort the tasks based on their end time. This allows us to prioritize the completion of tasks that need to be finished sooner.

Next, we try to select time points for each task starting from the task's end time and moving backwards to fill in its required duration. This increases the likelihood that the computer's operating time for one task will also count towards the operating times of subsequent tasks, essentially reusing time points. This greedy approach ensures that we do not unnecessarily turn on the computer earlier than needed for any given task.

To keep track of which time points have already been scheduled, we use an array `vis` as a timeline, marking off time points that have been allocated. For each task, we first determine how many of its required time points are already covered by other tasks (scheduled previously), and then we fill in the remaining duration as needed. The total number of "turned on" time points, `ans`, gives us the minimum time the computer needs to be active. After iterating through all tasks in this manner and updating the timeline (`vis` array), the final value of `ans` will be our desired minimum on time.

By being strategic about the order in which we choose time points for tasks and by optimizing for overlap, we arrive at a solution that minimizes the computer's active time while ensuring all tasks are completed within their respective time windows.

Solution Approach

The solution uses a greedy algorithm, a common pattern for optimization problems where we build up a solution piece by piece, selecting the most optimal choice at each step. In this case, sorting tasks by their end times and selecting time points for each task from end to start qualifies as such a choice. The underlying data structures used are a simple list for tracking tasks and an array (list) for recording which time points have already been selected.

Here's a step-by-step breakdown of the implementation:

1. **Sorting tasks**: We sort the `tasks` by their end times, which is done using Python's `sort` method with a lambda function defining the sorting key as `x[1]`, where `x` is each task and `x[1]` represents the end time of each task.

   ```
   tasks.sort(key=lambda x: x[1])
   ```

2. **Initializing the timeline**: An array `vis` is created with a sufficient size (2010 is chosen to cover the range of possible timestamps), initialized to 0s, indicating that no time points have been used yet.

   ```
   vis = [0] * 2010
   ```

3. **Greedy time point selection**: For each task, we determine how many time points are already covered (`sum(vis[start:end + 1])`) and reduce the `duration` of the task accordingly.

   ```
   duration -= sum(vis[start : end + 1])
   ```

4. **Filling in the remaining duration**: We iterate from the task's `end` time down to its `start` time, and for each unit of time, if it has not been used (`if not vis[i]`), we decrease the remaining `duration`, mark the time point as used (`vis[i] = 1`), and increment the `ans` count indicating the computer was turned on for another time unit.

   ```
   i = end
   while i >= start and duration > 0:
       if not vis[i]:
           duration -= 1
           vis[i] = 1
           ans += 1
       i -= 1
   ```

5. **Returning the result**: After processing all tasks, `ans` holds the total number of time points selected, which is the minimum time the computer needs to be on.

The strategy ensures that if a time point can serve multiple tasks, it will be used for all of them, reducing the total number of distinct time points, and thus, the total time the computer needs to be on. By iterating from the end of the task's window towards the start, we prioritize later usage over earlier, which aligns with our sorted task list and makes it more likely to reuse time points for subsequent tasks.

The overall time complexity of the algorithm is O(n * k), where n is the number of tasks, and k is the average range of the tasks (end - start). It is dominated by the time taken to find the sum of visited time points for each task and attempting to fill its remaining duration which might involve iterating over the range of the task.

Example Walkthrough

Let's use a simplified example to illustrate the solution approach. Assume we have 3 tasks that need to be scheduled, with each task represented as `[start time, end time, duration]`.

Task List:

```
Task 1: [1, 4, 3]
Task 2: [2, 6, 2]
Task 3: [5, 8, 1]
```

Step 1: Sort the tasks by their end times. After sorting:

```
Task 1: [1, 4, 3]
Task 2: [2, 6, 2]
Task 3: [5, 8, 1]
```

The tasks are already sorted by their end times.

Step 2: Initialize the timeline (`vis`) as an array with sufficient size.

```
vis = [0] * 2010  # 2010 is just for example purposes and assumes no times will exceed this value.
```

Step 3: Move through the tasks one by one and try to cover each task's duration, starting from its end time.

- For Task 1 `[1, 4, 3]`, we start from time 4. Since no other task is scheduled yet, we mark times 4, 3, and 2 as used in `vis` and reduce `duration` to 0. The `ans` count is now 3.

```
vis = [..., 0, 1, 1, 1, 0, 0, ...]  # '...' represents unchanged values; index 2 to 4 are marked as 1.
ans = 3
```

- For Task 2 `[2, 6, 2]`, we see that time 2 is already used by Task 1. We only need to schedule Task 2 for one more unit of time. We use time 6, mark it as used, and increment `ans` to 4.

```
vis = [..., 0, 1, 1, 1, 0, 1, 0, ...]  # Now index 6 is also marked as 1.
ans = 4
```

- For Task 3 `[5, 8, 1]`, we start from time 8. It's free, so we mark it as used, and `ans` increments to 5.

```
vis = [..., 0, 1, 1, 1, 0, 1, 0, 1, ...]  # Index 8 is marked as 1.
ans = 5
```

Step 4: After all tasks have been scheduled, `ans` is 5, which means the minimum time the computer needs to be active is 5 units.

Summary: The tasks overlap between time 2 and 4, so the computer only needs to be active for a total of 5 time units to cover all tasks, even though the sum of individual task durations is 6. The greedy approach successfully minimized the computer's on time by scheduling overlapping tasks simultaneously.

Python Solution

```python
from typing import List

class Solution:
    def findMinimumTime(self, tasks: List[List[int]]) -> int:
        # Sort tasks by the end time
        tasks.sort(key=lambda x: x[1])

        # Initialize a list to keep track of visited time slots, assuming the maximum end time is less than 2010
        visited = [0] * 2010

        # Initialize answer to count the minimum time required to finish all tasks
        minimum_time_required = 0

        # Iterate over each task
        for start_time, end_time, duration in tasks:
            # Decrease duration by the number of already visited time slots within the task's window
            duration -= sum(visited[start_time:end_time + 1])

            # Initialize a pointer at the end time of the current task
            i = end_time

            # Check if we can place the task in the current window and still need more duration
            while i >= start_time and duration > 0:
                # If the time slot is not yet visited
                if not visited[i]:
                    # Decrease the remaining duration since we are using this time slot
                    duration -= 1

                    # Mark this time slot as visited
                    visited[i] = 1

                    # Increment the minimum time required since we've occupied another time slot
                    minimum_time_required += 1

                # Move to the previous time slot and repeat till the start time or the task duration is met
                i -= 1

        # Return the computed minimum time required to finish all tasks
        return minimum_time_required

```

Java Solution

```java
import java.util.Arrays;

class Solution {

    public int findMinimumTime(int[][] tasks) {
        // Sort the tasks based on their end times
        Arrays.sort(tasks, (a, b) -> a[1] - b[1]);

        // Initialize an array to track the usage of each time slot
        int[] visits = new int[2010];

        // Initialize an answer variable to keep the count of visited time slots
        int ans = 0;

        // Iterate through each task
        for (int[] task : tasks) {
            int startTime = task[0]; // Start time of the current task
            int endTime = task[1]; // End time of the current task
            int duration = task[2]; // Duration of the current task needs in terms of unvisited time slots

            // Decrease duration for each visited time slot within the task's time range
            for (int i = startTime; i <= endTime; i++) {
                duration -= visits[i];
                if (duration < 0) {
                    duration = 0; // Duration shouldn't be less than 0
                }
            }

            // Allocate unvisited time slots starting from the end time of the task
            for (int i = endTime; i >= startTime && duration > 0; i--) {
                if (visits[i] == 0) { // If time slot is unvisited
                    --duration; // Decrement the remaining duration
                    ans += visits[i] = 1; // Visit this time slot and increment the answer
                }
            }
        }

        // Return the total number of visited time slots used to execute all tasks
        return ans;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For sort
#include <bitset>

class Solution {
public:
    // Function to find the minimum time to complete all tasks
    int findMinimumTime(vector<vector<int>>& tasks) {
        // Sort the tasks by their end time in ascending order
        sort(tasks.begin(), tasks.end(), [&](const auto& a, const auto& b) { return a[1] < b[1]; });

        // Initialize a bitset to keep track of visited time slots, assuming maximum 2010 slots
        bitset<2010> visited;

        // Initialize the answer to store the minimum number of time slots required
        int minimumTimeRequired = 0;

        // Iterate over each task to schedule them
        for (const auto& task : tasks) {
            // Extracting the start time, end time and duration for each task
            int startTime = task[0];
            int endTime = task[1];
            int duration = task[2];

            // Calculate the amount of duration that can be allocated within the time window
            for (int i = startTime; i <= endTime; ++i) {
                // Reduce duration for already visited time slots within the window
                duration -= visited[i];
            }

            // Assign time slots for remaining duration in reverse order to meet end time
            for (int i = endTime; i >= startTime && duration > 0; --i) {
                // If the time slot is not yet visited
                if (!visited[i]) {
                    // Decrease remaining duration and mark this time slot as visited
                    --duration;
                    minimumTimeRequired += visited[i] = 1;
                }
            }
        }

        // Return the computed minimum time required to complete all tasks
        return minimumTimeRequired;
    }
};

```

Typescript Solution

```typescript
function findMinimumTime(tasks: number[][]): number {
  // Sort the tasks array by end time in ascending order
  tasks.sort((taskA, taskB) => taskA[1] - taskB[1]);

  // Create an array to keep track of visited times, initialized with zeros
  const visited = new Array(2010).fill(0);

  // Initialize the answer variable to count the minimum time needed to complete tasks
  let minimumTime = 0;

  // Iterate through each task to find the minimum time necessary
  for (let task of tasks) {
    const [startTime, endTime, taskDuration] = task; // Destructure task details

    let remainingDuration = taskDuration; // Initialize remaining task duration

    // Calculate the remaining duration after considering previously visited time slots
    for (let time = startTime; time <= endTime; ++time) {
      remainingDuration -= visited[time];
    }

    // Fill the time slots from end to start respecting the task duration
    for (let time = endTime; time >= startTime && remainingDuration > 0; --time) {
      if (visited[time] === 0) {
        --remainingDuration; // Decrease remaining duration as we fill the time slot
        minimumTime += visited[time] = 1; // Mark the time slot as visited and increment minimum time
      }
    }
  }

  // Return the computed minimum time necessary to complete the tasks
  return minimumTime;
}

```

Time and Space Complexity

Time Complexity

The given code sorts the `tasks` list, which takes `O(NlogN)` time, where `N` is the number of tasks.

After sorting, the code iterates through each task with a nested while loop. In the worst case, for each task, the while loop could iterate from `end` to `start`, which could be `O(M)`, where `M` is the range of possible start and end times.

As a rough upper bound, this nested iteration could result in the inner operation being executed `N * M` times, since for each of 'N' tasks, the while loop could potentially iterate 'M' times (where 'M' is the largest possible `end` time — in this case, 2010). However, because each time unit can only be visited once due to the `vis` condition, the total number of operations of the inner loop is bounded by 'M' across all tasks. This makes the nested while loop take `O(M)` in total.

Summing these two parts, the overall time complexity is `O(NlogN + M)`. Note that in practice, `M` is bounded by a constant (2010), so if we consider `M` to be a constant, then the time complexity simplifies to `O(NlogN)`.

Space Complexity

The space consumed by the solution includes the storage for the visited time units `vis`, which has a fixed size based on the problem constraints (`2010`), and the internal space used for the sorted array of tasks. Therefore, the space complexity is `O(M + N)`. If `M` is considered to be a constant due to its fixed upper bound, then the space complexity simplifies to `O(N)`.