1986. Minimum Number of Work Sessions to Finish the Tasks

Medium | Bit Manipulation | Array | Dynamic Programming | Backtracking | Bitmask | Leetcode Link

Problem Description

You are given `n` tasks to complete, with their durations specified in an array called `tasks`, where the `i-th` task takes `tasks[i]` hours to complete. You also have a maximum duration you can work in a single work session, called `sessionTime`. A work session is defined as a period during which you can work for up to `sessionTime` consecutive hours before taking a break. The goal is to figure out the minimum number of work sessions required to complete all tasks under the following rules:

1. You must complete a task within the same work session if you start it.
2. After finishing a task, you can start a new one immediately.
3. The tasks can be completed in any order.

Your aim is to determine the minimum number of work sessions needed to finish all the assigned tasks without violating the conditions mentioned.

Intuition

Solving this problem involves understanding that it is a combinatorial optimization problem which suggests finding an optimal combination of tasks that fit within the session time limit. Since `sessionTime` is guaranteed to be greater than or equal to the maximum time for a single task, no task is unstartable.

The first step towards the solution is to generate all the possible combinations of tasks that can fit within a single session. This is achieved by iterating over each possible subset of tasks, where each subset is represented by a bitmask. A bitmask is a binary representation where each bit corresponds to whether a task is included in the subset or not (1 for included, 0 for not included).

For each subset, we check if the total time of the tasks in that subset does not exceed the `sessionTime`. If it doesn't, we mark this subset as a valid combination that can be completed within a single work session.

Next, we need to determine the minimum number of sessions required to complete all tasks. We initialize an array, `f`, to store the minimum sessions required for each subset of tasks. The value of `f[i]` represents the minimum number of sessions required to complete the subset `i`.

We use Dynamic Programming to build up the solution. For each subset `i`, we consider all possible valid combinations that have already been identified. We try to improve the minimum session count by checking if excluding a valid subset `j` from `i` (calculated as `i XOR j`) decreases the overall session count. In other words, we are trying to find the best previous state (`f[i XOR j]`) and then adding one more session for the current subset `j`.

Finally, `f[-1]` gives us the minimum number of sessions required to complete all the tasks, as it represents the state where all tasks have been included in some work session.

This approach efficiently finds the optimal number of work sessions by exploring and evaluating all possible task combinations and accumulating the results with dynamic programming.

Solution Approach

The implementation of this solution relies on several key concepts, including bit manipulation and dynamic programming.

Bit Manipulation

This algorithm uses bit manipulation to represent subsets of tasks. The key insight of using bitmasks is that a subset of `n` tasks can be represented as an `n`-bit integer. For example, if `n` is 3, then the binary `101` represents the subset where the first and third tasks are included, and the second task is excluded. This allows us to iterate over all possible subsets efficiently, using bitwise operations.

Dynamic Programming

Dynamic programming (DP) is used to build up the solution by reusing previously computed results. The DP array, `f`, is indexed by the subsets of tasks, where each index corresponds to a bitmask representing the subset. The value `f[i]` stores the minimum number of sessions required to complete the tasks in subset `i`.

Implementation Steps

1. First, we initialize an array, `ok`, to keep track of which subsets can be completed within a single session. We populate this array by iterating through all possible subsets (from `1` to `2^n - 1`), summing up the times of tasks included in each subset, and checking if the sum is within `sessionTime`.

2. Once we have identified all valid subsets, we initialize the DP array, `f`, with infinity (`inf`) to represent an initially unknown minimum. The starting state, `f[0]`, is set to `0`, because no sessions are needed when no tasks are included.

3. To compute the minimum sessions for each subset, we iterate over all subsets `i`. For each subset, we iterate through its submasks `j`. This part uses a nested looping structure where the inner loop uses a clever bit trick to iterate through all submasks of `i`. The expression `(j - 1) & i` ensures that we only consider submasks that are actual subsets of `i`.

4. For each submask `j`, if `j` is a valid subset (`ok[j]` is `True`), we check if we can get a better solution by combining the sessions required for the subset `i XOR j` (which means the subset `i` without the tasks in `j`) and the current submask `j`. If this is the case, we update `f[i]` with the minimum value between the current `f[i]` and `f[i XOR j] + 1`.

5. We continue this process until we've computed the optimal session counts for all subsets. The final answer will be stored in `f[-1]`, which represents the minimum number of work sessions needed for the complete set of tasks.

This implementation efficiently combines the powers of combinatorial enumeration via bit manipulation and the optimization capabilities of dynamic programming, resulting in an optimal solution to the task scheduling problem.

Example Walkthrough

Let's assume we have `n = 3` tasks with durations specified in an array called `tasks = [1, 2, 3]`, and we have a maximum session duration `sessionTime = 3`.

1. **Identify valid subsets**:
   - We start by generating all possible subsets of tasks and checking which can fit into a single session.
   - Subset `{1}` (binary `001`, total duration `1`) fits in a session.
   - Subset `{2}` (binary `010`, total duration `2`) fits in a session.
   - Subset `{3}` (binary `100`, total duration `3`) fits in a session.
   - Subset `{1, 2}` (binary `011`, total duration `3`) fits in a session.
   - Subsets `{1, 3}` (binary `101`) and `{2, 3}` (binary `110`), and `{1, 2, 3}` (binary `111`) do not fit as their durations exceed `sessionTime`.
2. **Dynamic Programming Initialization**:
   - Initialize DP array `f` with `infinity`, except for `f[0] = 0`. Here, `f[0]` corresponds to no tasks being complete.
   - The `f` array before starting the DP process: `f = [0, inf, inf, inf, inf, inf, inf, inf]`.
3. **Building up the DP table**:
   - For subset `{1}` (`001`), `f[1] = 1`, as it only needs one session.
   - For subset `{2}` (`010`), `f[2] = 1`, as it only needs one session.
   - For subset `{3}` (`100`), `f[4] = 1`, as it only needs one session.
   - For subset `{1, 2}` (`011`), `f[3] = 1`, as both can be done in one session.
4. **Using DP to find the minimum number of sessions**:
   - For subsets beyond `{1, 2}` (`011`), we need to consider splitting into submasks.
   - Calculate `f[5]` for subset `{1, 3}`:
     - The submasks are `{1}` (needs `1` session) and `{3}` (needs `1` session).
     - Since `{1, 3}` doesn't fit in one session, we add the session counts of the submasks: `f[5] = f[1] + f[4] = 1 + 1 = 2`.
   - Calculate `f[6]` for subset `{2, 3}`:
     - The submasks are `{2}` and `{3}`, each needing one session.
     - Since `{2, 3}` doesn't fit in one session, `f[6] = f[2] + f[4] = 1 + 1 = 2`.
   - Finally, for the full set `{1, 2, 3}` (`111`),
     - No single submask fits the entire set in one session.
     - The optimal way is to combine `{1, 2}` and `{3}`, so `f[7] = f[3] + f[4] = 1 + 1 = 2`.
5. **Conclusion**: The final DP array `f` stands as `[0, 1, 1, 1, 1, 2, 2, 2]`. The answer is `f[7]` which is `2`. Therefore, the minimum number of work sessions needed to finish all tasks is `2`.

Python Solution

```python
from math import inf
from typing import List

class Solution:
    def minSessions(self, tasks: List[int], sessionTime: int) -> int:
        # Calculate the number of tasks.
        num_tasks = len(tasks)

        # Initialize a list of booleans to keep track of which combinations of tasks
        # can fit into a single session.
        can_fit_session = [False] * (1 << num_tasks)

        # Check all combinations of tasks.
        for mask in range(1, 1 << num_tasks):
            # Calculate the total time of tasks in the current combination.
            total_time = sum(tasks[j] for j in range(num_tasks) if mask >> j & 1)

            # Set True in can_fit_session if the total time of tasks is within the sessionTime.
            can_fit_session[mask] = total_time <= sessionTime

        # Initialize an array to store the minimum sessions needed for every task combination.
        min_sessions_needed = [inf] * (1 << num_tasks)
        # Base case: zero tasks require zero sessions.
        min_sessions_needed[0] = 0

        # Calculate the minimum sessions required for all the combinations.
        for mask in range(1, 1 << num_tasks):
            # Store the current mask to iterate over its subsets.
            subset = mask
            # Iterate over all the subsets of the mask.
            while subset:
                # Check if the current subset can fit into one session.
                if can_fit_session[subset]:
                    # Update the minimum sessions needed if we can achieve a smaller number.
                    min_sessions_needed[mask] = min(min_sessions_needed[mask], min_sessions_needed[mask ^ subset] + 1)
                # Move to the next subset.
                subset = (subset - 1) & mask

        # Return the minimum sessions needed for all tasks.
        return min_sessions_needed[-1]

# Example usage:
# solution = Solution()
# print(solution.minSessions([1,2,3,4,5], 15))  # Output should be the minimum number of sessions required to complete all tasks
```

Java Solution

```java
import java.util.Arrays;

class Solution {
    public int minSessions(int[] tasks, int sessionTime) {
        // Number of tasks
        int numTasks = tasks.length;

        // An array to keep track of which subsets of tasks can fit into a single session
        boolean[] canFitInSession = new boolean[1 << numTasks];

        // Evaluate all subsets of tasks to see if they can fit in a single session
        for (int i = 1; i < (1 << numTasks); ++i) {
            int totalTime = 0;
            // Calculate total time for the current subset of tasks
            for (int j = 0; j < numTasks; ++j) {
                if ((i >> j & 1) == 1) {
                    totalTime += tasks[j];
                }
            }
            // Mark this subset as fitting in a session if the totalTime does not exceed sessionTime
            canFitInSession[i] = totalTime <= sessionTime;
        }

        // f[i] will hold the minimum number of sessions required for the set of tasks represented by 'i'
        int[] minSessionsRequired = new int[1 << numTasks];
        Arrays.fill(minSessionsRequired, Integer.MAX_VALUE); // Initialize with max value
        minSessionsRequired[0] = 0; // Base case: No tasks require 0 sessions

        // Iterate over all subsets of tasks
        for (int i = 1; i < (1 << numTasks); ++i) {
            // Consider all sub-subsets of the current subset 'i'
            for (int subset = i; subset > 0; subset = (subset - 1) & i) {
                // If the subset can fit in a session, try to update the minimum sessions required.
                if (canFitInSession[subset]) {
                    // the new state i ^ subset represents the remaining tasks after taking the session subset
                    minSessionsRequired[i] = Math.min(minSessionsRequired[i], minSessionsRequired[i ^ subset] + 1);
                }
            }
        }

        // The answer is the minimum number of sessions required to complete all tasks
        return minSessionsRequired[(1 << numTasks) - 1];
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <cstring>
#include <algorithm>

class Solution {
public:
    int minSessions(std::vector<int>& tasks, int sessionTime) {
        int n = tasks.size(); // Number of tasks
        std::vector<bool> ok(1 << n, false); // 'ok' array to flag valid subsets

        // Initialize the 'ok' array with subsets that can fit in a single session
        for (int mask = 1; mask < (1 << n); ++mask) {
            int totalTime = 0;
            for (int j = 0; j < n; ++j) {
                if ((mask >> j) & 1) {
                    totalTime += tasks[j];
                }
            }
            ok[mask] = (totalTime <= sessionTime);
        }

        // Initialize the 'dp' array to store the minimum number of sessions needed
        std::vector<int> dp(1 << n, INT_MAX);
        dp[0] = 0; // Base case: No tasks require 0 sessions

        // Calculate the minimum number of sessions required for each subset of tasks
        for (int i = 1; i < (1 << n); ++i) {
            for (int j = i; j; j = (j - 1) & i) { // Iterate through all submasks of i
                if (ok[j]) {
                    // If the current subset of tasks can be completed in one session,
                    // update the 'dp' value for the current combination of tasks.
                    dp[i] = std::min(dp[i], dp[i ^ j] + 1);
                }
            }
        }

        // The last element in 'dp' represents all tasks which is the answer
        return dp[(1 << n) - 1];
    }
};
```

Typescript Solution

```typescript
function minSessions(tasks: number[], sessionTime: number): number {
    const numTasks = tasks.length;
    // 'canCompleteSession' is an array indicating for each subset of tasks whether it can be completed in a single session.
    const canCompleteSession: boolean[] = new Array(1 << numTasks).fill(false);

    // Populate 'canCompleteSession' with true for subsets of tasks that fit within 'sessionTime'.
    for (let mask = 1; mask < 1 << numTasks; ++mask) {
        let totalTime = 0;
        for (let taskIndex = 0; taskIndex < numTasks; ++taskIndex) {
            if (((mask >> taskIndex) & 1) === 1) {
                totalTime += tasks[taskIndex];
            }
        }
        canCompleteSession[mask] = totalTime <= sessionTime;
    }

    // 'minSessionsNeeded' keeps track of the minimum number of sessions needed for each subset of tasks.
    const minSessionsNeeded: number[] = new Array(1 << numTasks).fill(Infinity);
    minSessionsNeeded[0] = 0;

    // Calculate the minimum number of sessions needed for all possible combinations of tasks.
    for (let mask = 1; mask < 1 << numTasks; ++mask) {
        // Explore submasks of 'mask' to split the tasks into multiple sessions.
        for (let subMask = mask; subMask > 0; subMask = (subMask - 1) & mask) {
            if (canCompleteSession[subMask]) {
                minSessionsNeeded[mask] = Math.min(minSessionsNeeded[mask], minSessionsNeeded[mask ^ subMask] + 1);
            }
        }
    }

    // Return the minimum number of sessions for all tasks.
    return minSessionsNeeded[(1 << numTasks) - 1];
}
```

Time and Space Complexity

The given Python code defines a method `minSessions` to find out the minimum number of work sessions required to finish all given tasks within a specified session time. The code utilizes bitmask Dynamic Programming (DP), where each state in DP represents a subset of tasks.

Time Complexity:

- Calculating the `ok` array requires iterating through all subsets of tasks, which are `2^n`, and summing up the tasks in each subset. This yields a time complexity of `O(n * 2^n)` for setting up the `ok` array, where `n` is the number of tasks.
- The nested loops for the DP solution iterate through all `2^n` subsets of tasks and for each subset, go through its submasks to update the `f[i]`. This results in another `O(n * 2^n)` operations (since the average number of submasks each mask has is proportional to `n`, considering the worst case). Combine both parts, and the total time complexity is `O(n * 2^n)`.

Space Complexity:

- The space is occupied by the `ok` array and the `f` array, both of which have `2^n` elements, resulting in `O(2^n)` space complexity.
- Additional space usage is minimal (constant space for the iteration variables and the sum), hence not impacting the overall space complexity.

So, to encapsulate:

- **Time Complexity:** `O(n * 2^n)`
- **Space Complexity:** `O(2^n)`