1335. Minimum Difficulty of a Job Schedule

Hard | Array | Dynamic Programming | Leetcode Link

Problem Description

In this problem, we're deciding how to schedule a list of jobs over `d` days. Jobs must be completed in a specific order — to complete the `i-th` job, all jobs before it must be done. A job schedule has a "difficulty", which is calculated by summing up the daily difficulties over `d` days, and the difficulty of each day is given by the hardest (highest difficulty) job completed on that day.

The goal is to find the job schedule with the minimum total difficulty possible while ensuring that at least one job is finished each day. If you're given a set of jobs, each with its own difficulty level (stored in the array `jobDifficulty`), and a certain number of days `d`, your task is to find this minimum difficulty. If it's not possible to schedule at least one job per day, the function should return `-1`.

To illustrate, imagine you're painting houses, and each house has a different level of effort required. You have a deadline to finish painting `d` sets of houses. Each day, you can paint a continuous block of houses, and the difficulty of the day is the hardest house you painted. Your goal is to minimize your total effort over the deadline without taking any days off.

Intuition

The intuitive approach to this kind of problem involves understanding that it's a classic case for dynamic programming, where the optimal solution can be built from the optimal solutions to smaller subproblems.

We create a 2D array `f` where `f[i][j]` represents the minimum difficulty to complete the first `i` jobs in `j` days. Initially, we know that `f[0][0] = 0` because no difficulty exists if no jobs are done in 0 days. For all other values, we initialize `f[i][j]` to infinity (`inf`) since we have not yet computed the minimum difficulty for those scenarios.

As we attempt to solve for `f[i][j]`, we encounter a choice: which jobs to schedule on the `j-th` day? We could select any sequence of jobs ending with the `i-th` job to be done on the last day. This selection defines the difficulty of the `j-th` day by the hardest job in that sequence. The remaining jobs would have been completed in the previous `j-1` days.

So, for each day `j` and job `i`, we iterate backward through the job list, calculating the difficulty for doing the last `k` jobs on day `j`, where `k` ranges from `i` down to `1`. We determine the max job difficulty `mx` for the last `k` jobs and add it to the previously computed minimum difficulty needed to finish the first `k-1` jobs in `j-1` days. This gives us the total difficulty if we were to complete the last `k` jobs on the current day `j`. We keep track of the minimum value for `f[i][j]` as we iterate through all possible `k`.

The final answer to our scheduling problem is represented by `f[n][d]`, which tells us the minimum total difficulty to finish all `n` jobs within `d` days, based on how the jobs were divided across the days.

Solution Approach

The problem is solved using a dynamic programming algorithm. Let's break down the implementation provided in the reference solution and understand how it aligns with the established approach:

1. **Initialization**: We create a 2D list `f` that will serve as our DP table with `n + 1` rows and `d + 1` columns. Each element is initially set to infinity (`inf`). The table is used to store the minimum difficulty of completing `i` jobs in `j` days. We also know that `f[0][0]` should be 0 because no difficulty exists when no jobs are done in 0 days.

2. **DP Table Population**: We use two nested loops to populate this table. The outer loop iterates over the range `[1, n + 1)` to represent the jobs. The inner loop iterates over the range `[1, min(d + 1, i + 1))` to represent the days, ensuring we don't try to schedule jobs in more days than we have jobs or days allocated.

3. **Finding the Minimum Difficulty**: Within the inner loop, we introduce another loop iterating from `i` down to `1` (backwards). This iteration is critical; for each job `i` and day `j`, it checks all possibilities of completing some last `k` jobs on the current day `j`. The variable `mx` holds the maximum job difficulty for these `k` jobs.

4. **State Transition**: For each `k`, we calculate a candidate minimum difficulty by taking the previously computed minimum difficulty for `k-1` jobs done in `j-1` days (`f[k - 1][j - 1]`) and adding the difficulty of completing the current `k` jobs on the current day, represented by `mx`. This reflects the state transition equation:

   ```
   f[i][j] = min(f[i][j], f[k - 1][j - 1] + mx)
   ```

   Here, `f[i][j]` holds the minimum difficulty for `i` jobs done in `j` days, and we update it only if we find a lower difficulty than the current stored value.

5. **Returning the Result**: After populating the DP table, we look at `f[n][d]` to see the result. If we find that it's still `inf`, it means that it's not possible to schedule the jobs within `d` days, and we should return `-1`. Otherwise, `f[n][d]` contains the minimum difficulty job schedule.

The key algorithms used here are dynamic programming for solving the optimized subproblems and iterating in reverse order to identify the possible subsets for achieving daily goals. The use of a 2D DP table for storing intermediate results is a common practice in dynamic programming problems to avoid recalculating solutions to subproblems multiple times.

By carefully updating our DP table and checking all possibilities, we guarantee that we find the minimum difficulty job schedule while meeting all the constraints of the problem.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have the following job difficulties and days:

```
jobDifficulty = [7, 1, 4, 5]
d = 2
```

We want to schedule these 4 jobs over 2 days such that the total difficulty of the schedule is minimized.

Step-by-Step Solution:

1. **Initialization**: We create a 2D list `f` with 5 (n + 1) rows (representing 0 to 4 jobs) and 3 (d + 1) columns (representing 0 to 2 days). This results in a 5×3 matrix initialized to infinity, except for `f[0][0]` which is 0.

   ```
   f = [[0, inf, inf],
        [inf, inf, inf],
        [inf, inf, inf],
        [inf, inf, inf],
        [inf, inf, inf]]
   ```

2. **DP Table Population**: We iterate over jobs (`i` from 1 to 4) and for each job, over days (`j` from 1 up to the current job number `i` without exceeding day `d`).

3. **Finding the Minimum Difficulty**: We perform an inner backward iteration from `i` down to 1 for each job/day combination.

4. **State Transition**: We update the DP array `f` at position `[i][j]`.

   - For `i=1` and `j=1`, the only option to schedule the first job on the first day means:
     ```
     f[1][1] = 7 (which is jobDifficulty[0])
     ```
   - Moving to `i=2`, we need to check:
     - If we do job 1 and 2 on day 1, then:
       ```
       f[2][1] = max(jobDifficulty[0], jobDifficulty[1]) = max(7, 1) = 7
       ```
     - If we do job 2 on day 1 and job 1 on day 0 which is not possible, so we ignore it.
   - Now for `i=2` and `j=2`, we need to check:
     - If we do job 1 on day 1 and job 2 on day 2, then:
       ```
       f[2][2] is min(inf, f[1][1] + max(jobDifficulty[1])) = min(inf, 7 + 1) = 8
       ```

Carrying on with this process till we complete all the jobs, we keep updating `f` based on the maximum difficulty of scheduling the last `k` jobs on the current day `j` and adding it to the minimum difficulty of scheduling the first `i - k` jobs on the previous days `j - 1`.

5. **Returning the Result**: After populating the table, we check `f[4][2]` for our answer. If it's `inf`, then it's impossible to schedule, but in this case, suppose it is `15`, that would be our minimum difficulty to schedule all jobs over 2 days as per our example.

Finally, the DP table `f` might look something like this after being populated (with hypothetical values for illustration):

```
f = [[0, inf, inf],
     [7, inf, inf],
     [7, 8, inf],
     [10, 11, inf],
     [15, 12, 15]]
```

And the result is `f[4][2] = 15`, which is the minimum total difficulty to schedule all 4 jobs over 2 days.

Python Solution

```python
from typing import List

class Solution:
    def minDifficulty(self, job_difficulty: List[int], days: int) -> int:
        num_jobs = len(job_difficulty)
        # Initialize DP table with infinity representing impossible scenarios.
        # dp_table[i][j] will hold the minimum difficulty of scheduling the first i jobs in j days
        dp_table = [[float('inf')] * (days + 1) for _ in range(num_jobs + 1)]
        dp_table[0][0] = 0 # Base case: 0 jobs in 0 days has 0 difficulty

        # Loop through each job
        for i in range(1, num_jobs + 1):
            # Only consider scheduling up to the minimum of days or jobs done so far
            for j in range(1, min(days + 1, i + 1)):
                max_difficulty_on_last_day = 0
                # Try to end the j-th day with each possible last job
                for k in range(i, 0, -1):
                    max_difficulty_on_last_day = max(max_difficulty_on_last_day, job_difficulty[k - 1])
                    # Update the DP table by adding the difficulty of the last job to the optimal difficulty of previous jobs in j-
                    dp_table[i][j] = min(dp_table[i][j], dp_table[k - 1][j - 1] + max_difficulty_on_last_day)

        # If the difficulty of scheduling all jobs in d days is infinite, it means it is not possible, thus return -1.
        # Otherwise, return the calculated difficulty
        return -1 if dp_table[num_jobs][days] == float('inf') else dp_table[num_jobs][days]

```

Java Solution

```java
class Solution {
    public int minDifficulty(int[] jobDifficulty, int d) {
        final int MAX_DIFFICULTY = Integer.MAX_VALUE / 2;  // Define a high number to represent an impossible situation
        int n = jobDifficulty.length;  // The total number of jobs
        int[][] dp = new int[n + 1][d + 1];  // Dynamic programming table

        // Initialize all dp table values with MAX_DIFFICULTY, except for the starting point
        for (int[] row : dp) {
            Arrays.fill(row, MAX_DIFFICULTY);
        }
        dp[0][0] = 0;  // Base case: 0 jobs on day 0 has difficulty 0

        // Fill in the dynamic programming table
        for (int i = 1; i <= n; ++i) {  // For each job
            for (int j = 1; j <= Math.min(d, i); ++j) {  // For each day, until the min of current job index and days
                int maxDifficulty = 0;  // To keep track of the maximum difficulty of jobs for the current day
                // Iterate through the previous jobs to find the minimum difficulty
                for (int k = i; k > 0; --k) {
                    maxDifficulty = Math.max(maxDifficulty, jobDifficulty[k - 1]);  // Find max difficulty among jobs
                    // Update the dp table for the minimum difficulty after completing current job on day j
                    dp[i][j] = Math.min(dp[i][j], dp[k - 1][j - 1] + maxDifficulty);
                }
            }
        }
        // If the final value is unmodified from the MAX_DIFFICULTY, it means it's not possible to schedule jobs within d days
        return dp[n][d] >= MAX_DIFFICULTY ? -1 : dp[n][d];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minDifficulty(vector<int>& jobDifficulty, int d) {
        int numJobs = jobDifficulty.size();

        // Create array to store the minimum difficulty on day j having completed job i
        int minDiff[numJobs + 1][d + 1];

        // Initialize the array with high initial values, except for start state
        memset(minDiff, 0x3f, sizeof(minDiff));
        minDiff[0][0] = 0;

        // Iterate through all jobs
        for (int i = 1; i <= numJobs; ++i) {
            // For each job, iterate through all possible days, but never more than i days
            for (int day = 1; day <= min(d, i); ++day) {
                int maxDifficulty = 0;

                // Looking for the last job on the previous day to minimize today's difficulty
                for (int k = i; k; --k) {
                    // Update the max difficulty for today's job set (k-1 to i)
                    maxDifficulty = max(maxDifficulty, jobDifficulty[k - 1]);

                    // Recurrence relation:
                    // minDiff for job i on day 'day' is the minimum of its current value or
                    // the sum of max difficulty encountered today and
                    // the best we could have done through job k-1 on day 'day-1'
                    minDiff[i][day] = min(minDiff[i][day], minDiff[k - 1][day - 1] + maxDifficulty);
                }
            }
        }

        // Check if it's possible to schedule jobs within d days. If not, return -1
        return minDiff[numJobs][d] == 0x3f3f3f3f ? -1 : minDiff[numJobs][d];
    }
};

```

Typescript Solution

```typescript
function minDifficulty(jobDifficulty: number[], days: number): number {
    const numOfJobs = jobDifficulty.length; // number of jobs
    const INF = 1 << 30; // represents an infinite difficulty, used as an initial value
    const dp: number[][] = new Array(numOfJobs + 1)
        .fill(0)
        .map(() => new Array(days + 1).fill(INF)); // initialize the dp array
    dp[0][0] = 0; // base case: 0 jobs done in 0 days has a difficulty of 0

    // Populate the dp table
    for (let i = 1; i <= numOfJobs; ++i) {
        for (let j = 1; j <= Math.min(days, i); ++j) { // Each job can only be scheduled if we have enough days
            let maxDifficulty = 0; // tracks the maximum difficulty of the current job set

            // Loop to find the minimum difficulty if the current job ends the current day
            for (let k = i; k > 0; --k) {
                maxDifficulty = Math.max(maxDifficulty, jobDifficulty[k - 1]);
                dp[i][j] = Math.min(dp[i][j], dp[k - 1][j - 1] + maxDifficulty);
            }
        }
    }

    // Return the minimum difficulty to complete all jobs in the given days
    // If it's still INF, it means it's impossible to schedule all jobs, hence return -1
    return dp[numOfJobs][days] < INF ? dp[numOfJobs][days] : -1;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n^2 * d)`. This is because there are three nested loops where the outermost loop runs for `n` iterations (jobs), the middle loop runs for at most `d` iterations (days), and the innermost loop also runs for at most `n` iterations, but on average will run fewer times as `k` decreases. However, in the worst case, the inner loop still contributes `n` operations per iteration of the middle loop. The product of these gives the time complexity of `O(n^2 * d)`.

The space complexity of the code is `O(n * d)`. This arises from the use of a 2D array `f` with dimensions `n + 1` by `d + 1`. Since the only other variables are constant size integers and not dependent on `n` or `d`, `f` dominates the space complexity.