1723. Find Minimum Time to Finish All Jobs

Hard Bit Manipulation Array Dynamic Programming Backtracking Bitmask Leetcode Link

Problem Description

In this problem, we are given an array called `jobs`, where each entry `jobs[i]` represents the time it takes to complete the `i`-th job. Additionally, there are `k` workers available to assign these jobs to. Each job must be assigned to one and only one worker. The "working time" for a worker is the total time they must spend to complete all jobs assigned to them. The objective is to distribute the jobs among the workers in such a way that the maximum working time for any single worker is minimized. This means we want to find the least amount of time any worker has to work if the jobs are assigned optimally. The function should return this minimum maximum working time.

Intuition

To solve this problem, we use a backtracking (or depth-first search) approach. Since we want to minimize the maximum working time of any worker, a brute force approach would involve trying all possible assignments of jobs to workers and finding the assignments with the minimum maximum working time, which would be very inefficient. This is where backtracking becomes helpful, as it allows us to explore possibilities but backtrack as soon as we know a certain assignment cannot be better than the best one found so far.

We start by sorting the jobs in decreasing order. This ensures that we are dealing with the largest jobs first, which gives us the chance to deal with difficult cases early and potentially prune the search space more effectively.

Using the `dfs` function (depth-first search), we recursively explore different job assignments, where we assign the `i`-th job to each worker one by one and recursively assign the next job. If adding the current job to a worker's existing workload would exceed the current best answer (which is stored in the global variable `ans`), we skip that assignment to prune the search space.

When we assign a job to a worker whose current workload is 0 (meaning they have not been assigned a job yet), and after this assignment, their workload becomes non-zero, we don't need to try assigning the job to other workers with 0 workloads. This is because the order in which workers without any assigned jobs start their first job doesn't matter.

The `ans` variable keeps track of the best (minimum) maximum working time found so far, and the `cnt` array tracks the current working time for each worker. The algorithm terminates when all jobs have been assigned and updates the `ans` with the minimum of the current maximum working times if it's better than the previously recorded maximum.

Solution Approach

The solution provided follows the backtracking approach to solve the problem efficiently. Here's a step-by-step walkthrough of the implementation of the solution based on the reference approach provided above:

1. **Sorting the jobs array**: Sorting the jobs in descending order helps in handling the largest jobs first during the backtracking process. This early handling of more significant jobs can lead to earlier pruning of the branches in the search tree, hence optimizing the performance.

```
jobs.sort(reverse=True)
```

2. **Depth-First Search (DFS)**: The `dfs` function is the core of the solution, which performs depth-first search (or backtracking).
   - When a job is to be assigned, the function iterates over all `k` workers to try and assign the job to each one of them.
   - It uses a counter array `cnt`, where `cnt[i]` is the total amount of time worker `i` has been assigned jobs.

```
def dfs(i):
```

3. **Pruning the search space**: Within the `dfs` function, there is a check to see if assigning the current job to a worker will result in a total working time that exceeds the best solution found so far (`ans`). If so, this branch of the recursion is abandoned; otherwise, the job is added to the worker's total time.

```
if cnt[j] + jobs[i] >= ans:
    continue
```

4. **Exploring and backtracking**: After assigning a job to a worker, the recursion continues with the next job (`i + 1`).

```
cnt[j] += jobs[i]
dfs(i + 1)
cnt[j] -= jobs[i]  # Backtrack
```

5. **Avoiding identical workers' permutations**: If a worker has not yet been assigned any job (`cnt[j] == 0`), there's no need to continue to the next worker after assigning a job to them because all workers are identical.

```
if cnt[j] == 0:
    break
```

6. **Updating the answer**: Once all jobs have been considered (`i == len(jobs)`), the function updates the maximum working time found so far (`ans`) if the current assignment is better.

```
ans = min(ans, max(cnt))
```

7. **Initialization**: Before diving into the backtracking, initialize the search with initial values.

```
cnt = [0] * k
ans = inf
```

8. **Kick off the DFS**: Start the DFS with the first job (`i=0`).

```
dfs(0)
```

9. **Return the result**: After the backtracking is complete, the `ans` variable holds the desired result, which is the minimum maximum working time across all workers.

```
return ans
```

The algorithm makes use of recursive backtracking and pruning techniques to navigate the space of possible allocations, aiming to minimize the maximum working time of any worker. The key here is to explore different combinations of job assignments in a depth-first search manner while trimming branches that exceed the current best solution, which efficiently leads to the minimum possible maximum working time.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Suppose we have the following input:

- `jobs = [5,3,2,1]`
- `k = 2` workers available

Now let's walk through the backtracking solution:

1. **Sorting the jobs array**: We sort the `jobs` array in descending order to deal with the larger jobs first.

```
jobs.sort(reverse=True) # jobs = [5, 3, 2, 1]
```

2. **Depth-First Search (DFS)**: Initialize the `dfs` function and the global variables `cnt` and `ans`.

```
cnt = [0] * k # cnt = [0, 0]
ans = float('inf')
```

3. **First DFS call**
   - We start our DFS from the first job; `dfs(0)` means we are considering how to assign the job with a time requirement of 5.
4. **Exploring job assignments**: We try assigning the job to each worker:
   - Assign job 1 to worker 0, `cnt = [5, 0]`, then make a recursive DFS call `dfs(1)` and go to the next job.
   - Assign job 1 to worker 0, `cnt = [8, 0]`, then `dfs(2)`. Since the sum is less than `ans`, we continue.
   - Assign job 2 to worker 0, `cnt = [10, 0]`, then `dfs(3)`. Since the sum exceeds the best so far, we backtrack.
   - Assign job 2 to worker 1, `cnt = [8, 2]`, then `dfs(3)`. We continue because it's still less than `ans`.
   - Assign job 3 to worker 0, `cnt = [9, 2]`, then we check if all jobs are assigned & update `ans` to 9.
   - Backtrack and assign job 1 to worker 1, `cnt = [8, 3]`, update `ans` to 8.
   - Backtrack to when job 1 was assigned to worker 0.
   - Try assigning job 1 to worker 1 instead, `cnt = [5, 3]`, and proceed with the next jobs similarly.
5. **Backtracking and pruning**: The DFS algorithm will backtrack whenever it encounters a sum that cannot be better than the current `ans`, which is considered pruning.
6. **Skip identical workers' states**: When a job is assigned to a worker with a 0 workload, we don't have to consider permutations with other workers with 0 workload.

After running through all possible combinations and backtracking when necessary, we find that the best distribution that minimizes the maximum working time for a worker is `[5, 3]` and `[2, 1]` with a maximum working time of 8. So the function will return 8.

This example demonstrates the depth-first search combined with pruning to avoid unnecessary work, leading us to the solution efficiently.

Python Solution

```
from typing import List

class Solution:
    def minimumTimeRequired(self, jobs: List[int], k: int) -> int:
        # Helper function for the depth-first search algorithm.
        def dfs(job_index, current_max):
            # Accessing 'ans' like as a nonlocal variable to modify it
            nonlocal min_time

            # Base case: all jobs have been assigned
            if job_index == len(jobs):
                # Update the minimum time if a better solution is found
                min_time = min(min_time, max(workers_load))
                return

            # Try assigning current job to each worker one by one
            for worker in range(k):
                # Pruning: Skip if current assignment exceeds the known minimum time
                if workers_load[worker] + jobs[job_index] >= min_time:
                    continue

                # Assign job to worker
                workers_load[worker] += jobs[job_index]
                # Recursively assign the next job
                dfs(job_index + 1, max(current_max, workers_load[worker]))
                # Backtrack: Remove job from worker (undo assignment)
                workers_load[worker] -= jobs[job_index]

                # If worker hadn't been assigned any job before, then this is the least loaded they can be
                # We need to try assigning jobs to other workers with the same load as it'll be equivalent
                if workers_load[worker] == 0:
                    break

        # Initialize each worker's load as 0
        workers_load = [0] * k
        # Sort the jobs in descending order to optimize the search (using greedy approach)
        jobs.sort(reverse=True)
        # Initialize the minimum time to infinity
        min_time = float('inf')
        # Start the DFS with the first job (0-index)
        dfs(0)
        # Return the minimum time found
        return min_time
```

Java Solution

```
class Solution {
    private int[] workersLoads; // Array to keep track of current load on each worker
    private int minimumTime; // Minimum time for completing all jobs
    private int[] jobs; // Array to store jobs sorted in descending order
    private int workers; // Number of workers available

    // Entry method to find the minimum time required to complete all jobs with k workers
    public int minimumTimeRequired(int[] jobs, int k) {
        this.jobs = jobs;
        Arrays.sort(jobs); // sort the jobs in ascending order

        // Reverse the job array to have jobs in descending order
        for (int i = 0, j = jobs.length - 1; i < j; ++i, --j) {
            int temp = jobs[i];
            jobs[i] = jobs[j];
            jobs[j] = temp;
        }
        this.jobs = jobs; // store the sorted jobs array
        workersLoads = new int[k]; // initialize the worker loads array
        minimumTime = Integer.MAX_VALUE; // initialize the minimum time to a large number

        // Start the depth-first search to assign jobs to workers
        dfs(0);
        return minimumTime;
    }

    // Depth-first search method to assign jobs to workers and find the minimum time
    private void dfs(int jobIndex) {
        if (jobIndex == jobs.length) { // If all jobs have been assigned
            int maxLoad = 0; // find out the maximum load on any worker
            for (int load : workersLoads) {
                maxLoad = Math.max(maxLoad, load);
            }
            minimumTime = Math.min(minimumTime, maxLoad); // update minimum time if necessary
            return;
        }

        // Iterate over workers and try to assign the current job to each worker
        for (int i = 0; i < workers; ++i) {
            // Skip assignment if job addition exceeds current minimum time
            if (workersLoads[i] + jobs[jobIndex] >= minimumTime) {
                continue;
            }
            workersLoads[i] += jobs[jobIndex]; // assign the job to the worker
            dfs(jobIndex + 1); // recurse to assign the next job

            // Backtrack: remove the job from the worker
            workersLoads[i] -= jobs[jobIndex];

            // If the current worker had 0 load, then there's no need to try further workers
            // since all are identical at this point
            if (workersLoads[i] == 0) {
                break;
            }
        }
    }
}
```

C++ Solution

```
class Solution {
public:
    int minimumTimeTime; // Renamed from 'ans' to 'minimumTimeTime' for clarity

    // Method to find the minimum time required to complete all jobs given 'k' workers
    int minimumTimeRequired(vector<int>& jobs, int k) {
        vector<int> workerTimes(k); // Holds the current total time for each worker
        minimumTimeTime = INT_MAX; // Initialize with the maximum possible value
        sort(jobs.begin(), jobs.end(), greater<int>()); // Sort jobs first for optimization
        sort(jobs.begin(), jobs.end(), greater<int>());
        // Start the Depth-First Search (DFS) to assign jobs
        dfs(0, k, jobs, workerTimes);
        return minimumTimeTime;
    }

    // Helper method to perform DFS
    void dfs(int currentIndex, int k, vector<int>& jobs, vector<int>& workerTimes) {
        if (currentIndex == jobs.size()) { // Base case: all jobs have been assigned
            // Update the minimum job time if necessary
            minimumTimeTime = min(minimumTimeTime, *max_element(workerTimes.begin(), workerTimes.end()));
            return;
        }

        // Iterate through each worker to assign the job
        for (int i = 0; i < k; ++i) {
            // Skip if adding this job would make the worker's time exceed the current minimum time
            if (workerTimes[i] + jobs[currentIndex] >= minimumTimeTime) continue;
            // Assign the job to the worker
            workerTimes[i] += jobs[currentIndex];
            // Recurse for the next job
            dfs(currentIndex + 1, k, jobs, workerTimes);
            // Backtrack: remove the job from the worker for the next iteration
            workerTimes[i] -= jobs[currentIndex];
            // Optimization: if the current worker has no jobs, don't assign this job to other workers
            if (workerTimes[i] == 0) break;
        }
    }
};
```

Typescript Solution

```
let minimumJobTime: number; // Holds the minimum job time found

// Function to find the minimum time required to complete all jobs given 'k' workers
function minimumTimeRequired(jobs: number[], k: number): number {
    let workerTimes: number[] = new Array(k).fill(0); // Holds the current total time for each worker
    minimumJobTime = Number.MAX_SAFE_INTEGER; // Initialize with the largest safe integer value due to the absence of INT_MAX
    // Sort jobs in descending order to handle the largest jobs first for better optimization
    jobs.sort((a, b) => b - a);
    // Start the Depth-First Search to assign jobs
    dfs(0, k, jobs, workerTimes);
    return minimumJobTime;
}

// Recursive function to perform Depth-First Search (DFS) for job assignments
const dfs = (currentIndex: number, k: number, jobs: number[], workerTimes: number[]): void => {
    if (currentIndex >= jobs.length) { // Base case: all jobs have been assigned
        // Update the minimum job time if necessary
        minimumJobTime = Math.min(minimumJobTime, Math.max(...workerTimes));
        return;
    }

    // Iterate through each worker to assign the job
    for (let i = 0; i < k; ++i) {
        // Skip if adding this job would make the worker's time exceed the current minimum job time
        if (workerTimes[i] + jobs[currentIndex] >= minimumJobTime) continue;
        // Assign the job to the worker
        workerTimes[i] += jobs[currentIndex];
        // Recurse for the next job
        dfs(currentIndex + 1, k, jobs, workerTimes);
        // Backtrack: remove the job from the worker for the next iteration
        workerTimes[i] -= jobs[currentIndex];
        // Optimization: if the current worker has no jobs, don't assign this job to other workers
        if (workerTimes[i] === 0) break;
    }
};
```

Time and Space Complexity

Time Complexity

The time complexity of the `minimumTimeRequired` function is determined by the depth-first search (DFS) as it tries to assign jobs to workers in every possible combination to find the minimum possible time.

- Since jobs are being sorted initially, this contributes `O(n log n)` to the time complexity, where `n` is the number of jobs.
- The DFS process has a worst-case time complexity of `O(k^n)` where `k` is the number of workers, and `n` is the number of jobs, since each job can be assigned to any of the `k` workers.
- However, due to pruning:
  - If `cnt[j] + jobs[i] >= ans`, the DFS does not continue down that path, which can significantly reduce the number of explored states.
  - If `cnt[j] == 0`, it breaks the inner loop, avoiding redundant assignments to empty slots which are equivalent.

The exact time complexity is hard to characterize due to these pruning strategies, but the worst-case without pruning is `O(k^n)`, factoring in the sorting of jobs, we have `O(n log n + k^n)`.

Space Complexity

For space complexity, the following considerations are taken into account:

- The recursive call stack of DFS contributes `O(n)` space complexity since the maximum depth of the recursion stack is `n` (the number of jobs).
- An array `cnt` of size `k` is used to keep track of the current sum of job times for each worker, contributing `O(k)` space complexity.

Therefore, the overall space complexity is `O(n + k)` due to the recursive stack and the `cnt` array. However, if we consider that the depth of the recursive stack dominates as `n` can potentially be larger than `k`, the space complexity simplifies to `O(n)`.