1723. Find Minimum Time to Finish All Jobs Bit Manipulation Bitmask Hard Array Dynamic Programming Backtracking

# Problem Description

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

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.

Leetcode Link

### To solve this problem, we use a backtracking (or depth-first search) approach. Since we want to minimize the maximum working time

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 1-th job to each

of any worker, a brute force approach would involve trying all possible assignments of jobs to workers and finding the assignment

with the minimum maximum working time, which would be very inefficient. This is where backtracking becomes helpful, as it allows

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.

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

## 2. Depth-First Search (DFS): The dfs function is the core of the solution, which performs depth-first search (or backtracking).

1 cnt[j] += jobs[i]

3 cnt[j] -= jobs[i] # Backtrack

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

1 cnt = [0] \* k

1 return ans

maximum working time.

Example Walkthrough

optimizing the performance.

the job is added to the worker's total time.

working time found so far (ans) if the current assignment is better.

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

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

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

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

1 jobs.sort(reverse=True)

Solution Approach

 It uses a counter array cnt, where cnt[j] is the total amount of time worker j has been assigned jobs. 1 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,

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.

- 1 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 (1 + 1).
- 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. 1 if cnt[j] == 0:

6. Updating the answer: Once all jobs have been considered (i == len(jobs)), the function updates the minimum maximum

8. Kick off the DFS: Start the DFS with the first job (i=0). 1 dfs(0)

9. Return the result: After the backtracking is complete, the ans variable holds the desired result, which is the minimum maximum

first search manner while trimming branches that exceed the current best solution, which efficiently leads to the minimum possible

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-

working time across all workers.

Now let's walk through the backtracking solution:

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

current ans, which is constantly being updated.

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

3. First DFS call

efficiently.

14

15

16

17

18

20

21

22

23

24

25

26

27

28

29

30

31

32

33

40

4

5

6

8

9

10

11

12

13

14

15

16

51

52

53

54

55

56

58

57 }

**Python Solution** 

class Solution:

from typing import List

def dfs(curr\_job\_index):

return

for worker in range(k):

continue

break

# Assign job to worker

dfs(curr\_job\_index + 1)

2 ans = float('inf')

1 cnt = [0] \* k # cnt = [0, 0]

 Backtrack to when job 3 was assigned to worker 0. Try assigning job 3 to worker 1 instead, cnt = [5, 3], and proceed with the next jobs similarly.

def minimumTimeRequired(self, jobs: List[int], k: int) -> int:

# Helper function for the depth-first search algorithm.

min\_time = min(min\_time, max(workers\_load))

# Try assigning current job to each worker one by one

workers\_load[worker] += jobs[curr\_job\_index]

workers\_load[worker] -= jobs[curr\_job\_index]

# Backtrack: Remove job from worker (undo assignment)

# Recursively assign the next job

if workers\_load[worker] == 0:

# Start the DFS with the first job (0-index)

# Pruning: skip if current assignment exceeds the known minimum time

if workers\_load[worker] + jobs[curr\_job\_index] >= min\_time:

private int[] workerLoads; // array to keep track of current load on each worker

// Entry method to find the minimum time required to complete all jobs with k workers

private int minimumTime; // minimum time for completing all jobs

Arrays.sort(jobs); // sort the jobs in ascending order

for (int i = 0, j = jobs.length - 1; <math>i < j; ++i, --j) {

// since all are identical at this point

int minimumJobTime; // Renamed from 'ans' to 'minimumJobTime' for clarity

// Method to find the minimum time required to complete all jobs given 'k' workers

vector<int> workerTimes(k); // Holds the current total time for each worker

// Sort jobs in descending order to try the largest jobs first for optimization

minimumJobTime = min(minimumJobTime, \*max\_element(workerTimes.begin(), workerTimes.end()));

minimumJobTime = INT\_MAX; // Initialize with the maximum possible value

if (workerLoads[j] == 0) {

int minimumTimeRequired(vector<int>& jobs, int k) {

dfs(0, k, jobs, workerTimes);

return;

sort(jobs.begin(), jobs.end(), greater<int>());

// Start the Depth-First Search (DFS) to assign jobs

// Update the minimum job time if necessary

// Iterate through each worker to assign the job

break;

// Reverse the jobs array to have jobs in descending order

private int workers; // number of workers available

public int minimumTimeRequired(int[] jobs, int k) {

private int[] jobs; // array to store jobs (sorted in descending order)

# Accessing 'min\_time' as a nonlocal variable to modify it

Backtrack and assign job 1 to worker 1, cnt = [8, 3], update ans to 8.

6. Skip identical workers' states: When a job is assigned to a worker with a vorkload, we don't have to consider permutations with other workers with 0 workload.

Assign job 5 to worker 0. cnt = [5, 0], then make a recursive DFS call dfs(1) and go to the next job.

Assign job 2 to worker 0, cnt = [10, 0], then dfs(3). Since the sum exceeds the best so far, we backtrack.

5. Backtracking and pruning: The DFS algorithm will backtrack whenever it encounters a sum that cannot be better than the

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

Assign job 3 to worker 0, cnt = [8, 0], then dfs(2). Since the sum is less than ans, we continue.

Assign job 2 to worker 1, cnt = [8, 2], then dfs(3). We continue because it's still less than ans.

Assign job 1 to worker 0, cnt = [9, 2], then we check if all jobs are assigned & update ans to 9.

nonlocal min\_time # Base case: all jobs have been assigned if curr\_job\_index == len(jobs): # Update the minimum time if a better solution is found

34 # Initialize each worker's load as 0 35 workers\_load = [0] \* k 36 # Sort the jobs in descending order to optimize the search (using greedy aproach) 37 jobs.sort(reverse=True) # Initialize the minimum time to infinity 38 39 min\_time = float('inf')

# If worker hadn't been assigned any job before, then this is the least loaded they can be

# No need to try assigning jobs to other workers with the same load as it'll be equivalent

```
# Return the minimum time found
           return min_time
43
44
```

Java Solution

1 class Solution {

this.workers = k;

int temp = jobs[i];

jobs[i] = jobs[j];

jobs[j] = temp;

dfs(0)

```
17
18
            this.jobs = jobs; // store the sorted jobs array
19
            workerLoads = new int[k]; // initialize the worker loads array
20
            minimumTime = Integer.MAX_VALUE; // initialize the minimum time to a large number
21
22
            // Start the depth-first search to assign jobs to workers
23
            dfs(0);
24
            return minimumTime;
25
26
27
       // Depth-first search method to assign jobs to workers and find the minimum time
28
       private void dfs(int jobIndex) {
29
            if (jobIndex == jobs.length) { // If all jobs have been assigned
30
                int maxLoad = 0; // find out the maximum load on any worker
31
                for (int load : workerLoads) {
32
                    maxLoad = Math.max(maxLoad, load);
33
34
                minimumTime = Math.min(minimumTime, maxLoad); // update minimum time if necessary
35
                return;
36
37
            // Iterate over workers and try to assign the current job to each worker
38
39
            for (int j = 0; j < workers; ++j) {</pre>
40
               // Skip assignment if job addition exceeds current minimum time
                if (workerLoads[j] + jobs[jobIndex] >= minimumTime) {
41
42
                    continue;
43
44
                workerLoads[j] += jobs[jobIndex]; // assign the job to the worker
45
                dfs(jobIndex + 1); // recurse to assign the next job
46
47
                // Backtrack: remove the job from the worker
                workerLoads[j] -= jobs[jobIndex];
48
49
50
               // If the current worker had 0 load, then there's no need to try further workers
```

### 13 return minimumJobTime; 14 15 // Helper method to perform DFS 16 17 void dfs(int currentIndex, int k, vector<int>& jobs, vector<int>& workerTimes) { if (currentIndex == jobs.size()) { // Base case: all jobs have been assigned

C++ Solution

1 class Solution {

2 public:

9

10

11

12

20

21

22

```
24
           for (int j = 0; j < k; ++j) {
25
               // Skip if adding this job exceeds the current minimum time
26
               if (workerTimes[j] + jobs[currentIndex] >= minimumJobTime) continue;
27
               // Assign the job to the worker
               workerTimes[j] += jobs[currentIndex];
28
               // Recur for the next job
               dfs(currentIndex + 1, k, jobs, workerTimes);
30
               // Backtrack: remove the job from the worker for the next iteration
31
32
               workerTimes[j] -= jobs[currentIndex];
               // Optimization: If the worker time is zero, no need to try assigning this job to other workers
33
               if (workerTimes[j] == 0) break;
34
35
36
37 };
38
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 try the largest jobs first for better optimization
       jobs.sort((a, b) \Rightarrow b - a);
       // Start the recursive search to assign jobs
       dfs(0, k, jobs, workerTimes);
10
       return minimumJobTime;
11
12 }
   // Recursive function to perform Depth-First Search (DFS) for job assignments
15 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 a new minimum is found
           minimumJobTime = Math.min(minimumJobTime, Math.max(...workerTimes));
19
           return;
20
       // Iterate through each worker to assign the job
21
22
       for (let j = 0; j < k; ++j) {
           // Skip if adding this job would make the worker's time exceed the current minimum job time
23
           if (workerTimes[j] + jobs[currentIndex] >= minimumJobTime) continue;
24
25
           // Assign the job to the worker
           workerTimes[j] += 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[j] -= jobs[currentIndex];
           // Optimization: if the current worker has no jobs, don't assign this job to other workers
31
```

### **Time Complexity** The time complexity of the minimumTimeRequired function is determined by the depth-first search (DFS) as it tries to assign jobs to

However, due to pruning:

Time and Space Complexity

if (workerTimes[j] === 0) break;

Space Complexity

32

33

35

34 };

26 27 28 29 30

- Since jobs are being sorted initially, this contributes 0(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,
- 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),

If cnt[j] + jobs[i] >= ans, the DFS does not continue down that path, which can significantly reduce the number of

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

workers in every possible combination to find the minimum possible time.

since each job can be assigned to any of the k workers.

factoring in the sorting of jobs, we have  $O(n \log n + k^n)$ .

- 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).