1335

Approach 4: Monotonic Stack - Better Time Complexity

Intuition

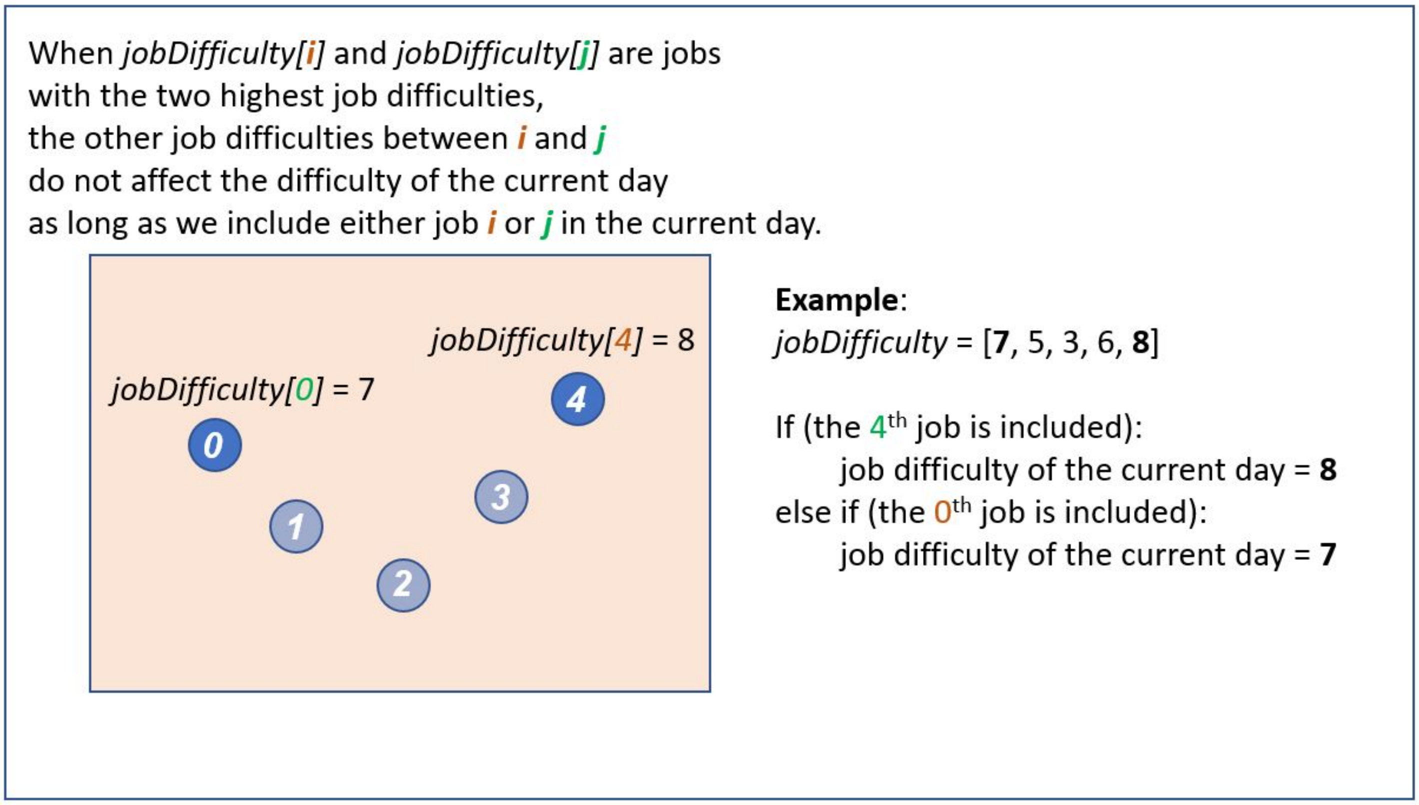
We have walked through the intuition and implementation of the top-down and bottom-up DP approaches. Being able to clearly articulate the logic and time/space complexity for these approaches will be more than sufficient for most interviews.

But is there an even better solution? Can we improve even more?

The answer is yes! This approach falls outside the scope of a typical interview. However, it will be fun to explore the solution, and to think about where else this methodology could be applied!

First, let's consider what inefficiency exists in the previous approach that we can improve on.

Remember our conclusion in the previous approaches that the value of `min_diff[d][i]` only depends on the results of `min_diff[d - 1][j]` when `j > i`. From the state transfer function `min_diff_curr_day[i] = min(min_diff_curr_day[i], dailyMaxJobDiff + min_diff_next_day[j])`, it is evident that the job difficulty of the current day is solely determined by the maximum job difficulty seen so far, on the current day (between index `i` and `j`). Therefore, when `jobDifficulty[i]` and `jobDifficulty[j]` are the two jobs with the highest job difficulty between `i` and `j`, then the values of other job difficulties between `i` and `j` do not affect the difficulty of the current day as shown in the figure below.


When *jobDifficulty[i]* and *jobDifficulty[j]* are jobs
with the two highest job difficulties,
the other job difficulties between ***i*** and ***j***
do not affect the difficulty of the current day
as long as we include either job ***i*** or ***j*** in the current day.

Example:
jobDifficulty = [**7**, 5, 3, 6, **8**]

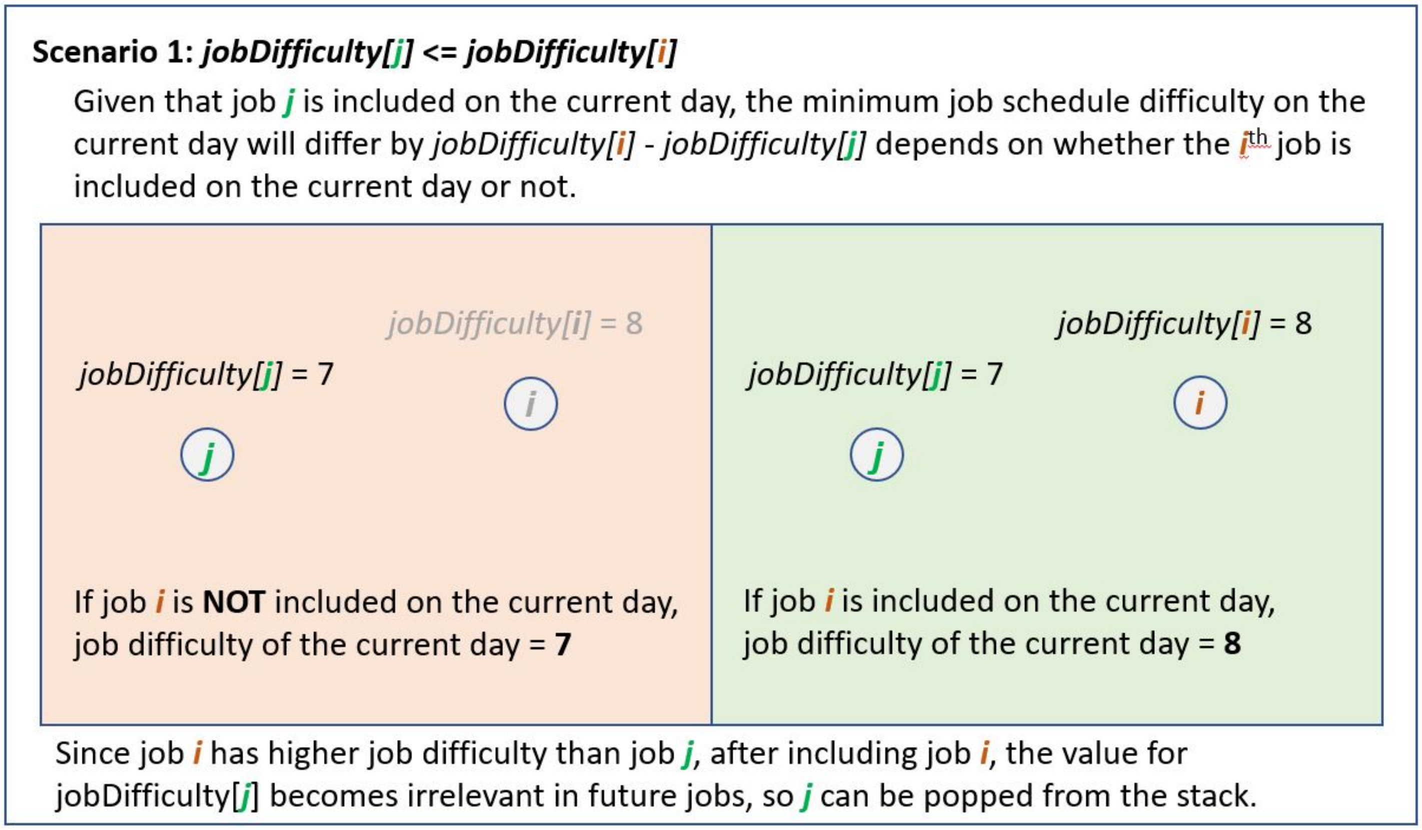
If (the 4th job is included):
job difficulty of the current day = **8**
else if (the 0th job is included):
job difficulty of the current day = **7**

With this in mind, we do not actually need to check every combination of `i` and `j` and update their values in the DP matrix. Instead, we can use a monotonic stack to reduce the total amount of calculations.

To better understand how we can use a monotonic stack to exploit this observation, let's start by focusing on two indices, `j` and `i` (where out of all the jobs between `j` and `i` inclusive, jobs at `j` and `i` will have the two greatest difficulties). Notice that the order of `i` and `j` is a different this time. Instead of `i` being the index of the first job done today and `j` being the index of the first job done tomorrow, in this approach, `i` is the index of the **last** job that we try to schedule for today, and `j` is the index of a job in the job schedule that is scheduled for today **before** index `i` (i.e., `j` < `i`).

Given that the jthj^{th}*jth* job is scheduled on the current day, let's find out how to update the value of minimum job schedule difficulty when we include the ithi^{th}*ith* job on the current day. There are two possible scenarios:

- Scenario 1: `jobDifficulty[j] <= jobDifficulty[i]`. After we include the ithi^{th}*ith* job on the current day, the minimum job schedule difficulty on the current day will increase by at most `jobDifficulty[i] - jobDifficulty[j]` compared with when the jthj^{th}*jth* job is the **last** job scheduled for the current day. Since job `i` has higher job difficulty than job `j`, the value for `jobDifficulty[j]` becomes irrelevant when scheduling any future jobs on the current day, so index `j` can be popped from the stack.

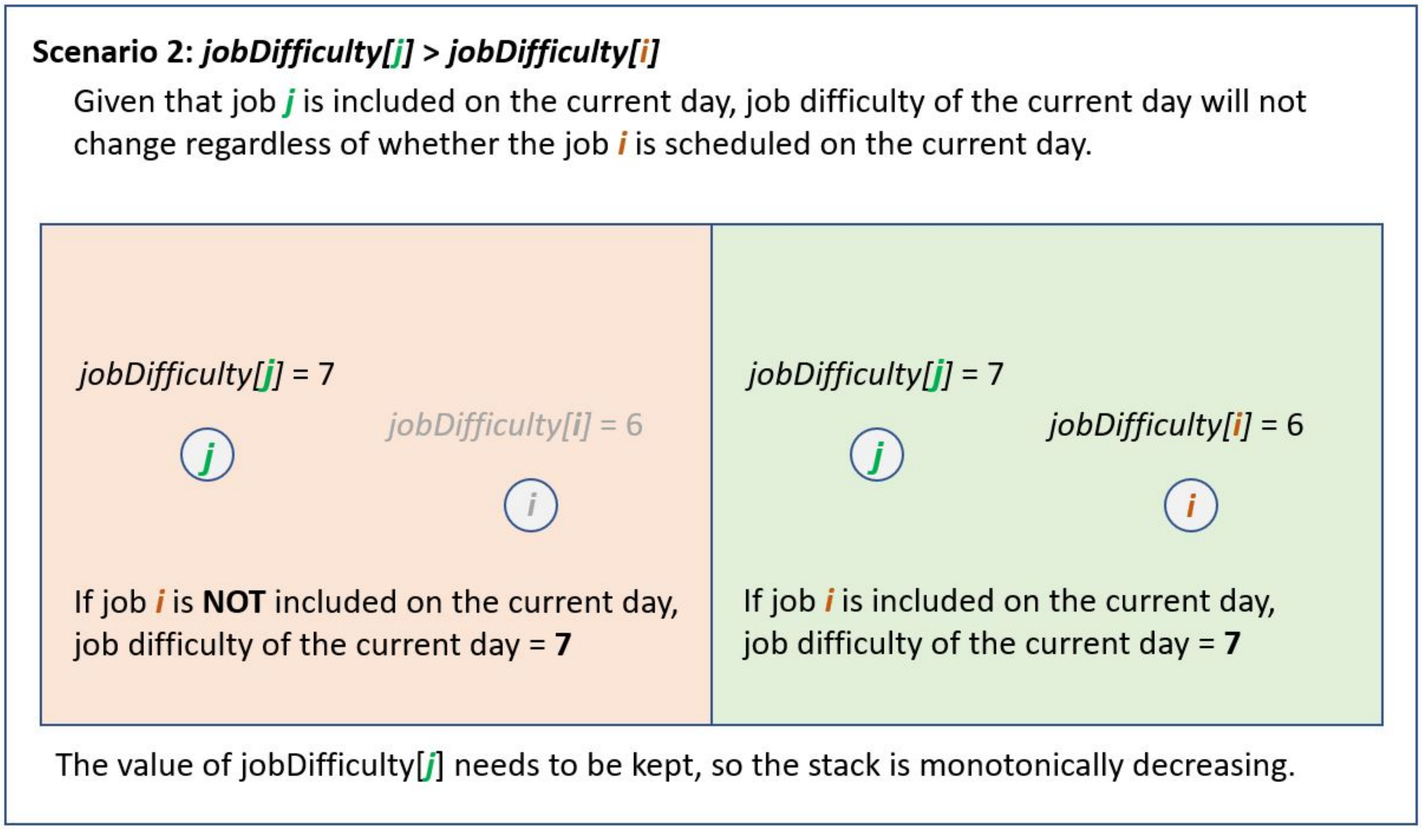

Scenario 1: *jobDifficulty[j] <= jobDifficulty[i]*

Given that job ***j*** is included on the current day, the minimum job schedule difficulty on the current day will differ by *jobDifficulty[i] - jobDifficulty[j]* depends on whether the *i*th job is included on the current day or not.

jobDifficulty[i] = 8
jobDifficulty[j] = 7
i
j
If job i is NOT included on the current day, job difficulty of the current day = 7
jobDifficulty[i] = 8
jobDifficulty[j] = 7
i
j
If job i is included on the current day, job difficulty of the current day = 8

Since job ***i*** has higher job difficulty than job ***j***, after including job ***i***, the value for jobDifficulty[*j*] becomes irrelevant in future jobs, so ***j*** can be popped from the stack.

- Scenario 2: `jobDifficulty[j] > jobDifficulty[i]`. The minimum job schedule difficulty on the current day only depends on the most difficult job scheduled on that day, which is `jobDifficulty[j]`. So after we include the ithi^{th}*ith* job on the current day, the minimum job schedule difficulty will be no higher than that of when the jthj^{th}*jth* job is the **last** job scheduled for the current day. We still need to keep job `j` in the stack for scheduling future jobs on the current day.


Scenario 2: *jobDifficulty[j] > jobDifficulty[i]*
Given that job *j* is included on the current day, job difficulty of the current day will not change regardless of whether the job *i* is scheduled on the current day.
jobDifficulty[j] = 7
jobDifficulty[i] = 6
j
i
If job *i* is **NOT** included on the current day, job difficulty of the current day = **7**
jobDifficulty[j] = 7
jobDifficulty[i] = 6
j
i
If job *i* is included on the current day, job difficulty of the current day = **7**
The value of jobDifficulty[*j*] needs to be kept, so the stack is monotonically decreasing.

To recap, in the above scenarios, every time a new job (`i`) is added to the current day's schedule, if any prior job (`j`) has a lower or equal job difficulty, then we can discard the job `j` as it becomes irrelevant for the job schedule difficulty of the current day (as in scenario 1). Otherwise, we will need to keep the value the job `j` when `jobDifficulty[j] > jobDifficulty[i]` (as in scenario 2). As a result, at any given moment, the difficulty of the jobs we keep in the stack will be monotonically decreasing. The monotonically decreasing relationship is a hint that we may be able to solve this problem more efficiently by using a monotonic stack. Notice that when the ithi^{th}*ith* job is added to the current day's schedule, we focus on whether previous jobs should be popped out from the stack and how to update the current state of job difficulty using the popped out job.

Now, with a monotonic stack in mind, let's approach the problem from a different angle (credit to @Lee215), as demonstrated in the slides below.

> Note, for the monotonic stack used here, instead of recording the values, we will keep track of the corresponding indices since we can always look up values in the `jobDifficulty` array and `min_diff_curr_day` array. So, the values that correspond to the indices contained in the stack are monotonically decreasing.

Contrary to the previous approach, where we iterated over the `days_remaining` from `1` to `d`. Here, we will iterate over the current day from `day` equals 0, which is the start of the job schedule, to `day` equals `d - 1`, which is the last day of the job schedule.

Each day, we will iterate over the indices `i` of all jobs that could be scheduled on the current day. We start at `i` equals `day` because at least one job per day must have been scheduled on previous days. For each index `i`, we will say this will be the index of the **last** job scheduled today. Thus, the current day's job minimum schedule difficulty will start as `jobDifficulty[i]` for `i = day` because, so far, the job at `i` is the only job scheduled today, and the job schedule difficulty (`min_diff_curr_day[i]`) is the difficulty of the job at index `i` plus the minimum difficulty of the job schedule that ended yesterday at index `i - 1`, which is `min_diff_prev_day[i - 1]`.

Now, let's try to schedule more jobs for today by increasing `i` and see how previous jobs at index `j` where `day <= j < i` will impact the minimum job difficulty of the current day. Here's the trick, as long as the difficulty of the job at index `i` is greater than the difficulty of all jobs between `j` and `i`, then extending a job schedule to include all jobs up to and including `i` will change the schedule difficulty by `jobDifficulty[i] - jobDifficulty[j]`. This is because `jobDifficulty[j]` is the most difficult job scheduled today so far, and by extending the schedule to include the job at `i`, the schedule difficulty for today will increase to `jobDifficulty[i]` from `jobDifficulty[j]`.

A naive way is to repeat this process by including each potential index `i`, and compare it with all previous job `j`, we will update the minimum job difficulty schedule that ends at job `i`. However, doing this for each `i` will result in O(n2⋅d)O(n^2 \cdot d)*O*(*n*2⋅*d*) total time required for this approach... which is no better than the previous approach.

So what makes this approach more efficient? The key insight is that after extending the job schedule that ends at index `i` once, we never need to consider extending it again. Here's why. Consider the case where we update the job schedule difficulty at `i` (`min_diff_curr_day[i]`) by extending the job schedule that ends at index `j` like so: `min_diff_curr_day[i] <= min_diff_curr_day[j] + (jobDifficulty[i]- jobDifficulty[j])`. Then at a later index (say `i + 1`), we update the job schedule difficulty at `i + 1` by extending the job schedule that ends at index `i` following the same equation. Let's take a close look at `min_diff_curr_day[i + 1]` when we extend the job schedule ending at `i` and `jobDifficulty[j] < jobDifficulty[i]`.

```
min_diff_curr_day[i + 1] <= min_diff_curr_day[i] + jobDifficulty[i + 1] -
jobDifficulty[i]
# Substitute in the above equation for min_diff_curr_day[i]
                         <= min_diff_curr_day[j] + jobDifficulty[i] -
jobDifficulty[j]
                                                 + jobDifficulty[i + 1] -
jobDifficulty[i]
                         = min_diff_curr_day[j] + jobDifficulty[i + 1] -
jobDifficulty[j]
```

Notice that extending the job schedule ending at `i` to include all jobs up to `i + 1` has a equal or tighter bound than extending the job schedule ending at `j` to include all jobs up to `i + 1` given `jobDifficulty[j] < jobDifficulty[i]`. For this reason, we only need to consider extending the job schedule that ends at `j` once, then the difficulty of the job schedule that ends at `j` can be forgotten.

The key insight is: once we have considered extending a schedule once, we never need to consider extending it again. Only considering the most recent schedules where today's difficulty is less than job `i` and safely ignoring schedules that have already been extended suggest that a stack will be a good data structure. For each `i`, we will pop the job schedules from the top of the stack where the difficulty of job `j` (the top of the stack) is less than `i`, and consider extending the popped job schedule to include `i`.

> "Consider extending the schedule" means updating `min_diff_curr_day[i]` to `min_diff_curr_day[j] + jobDifficulty[i] - jobDifficulty[j]` if it is less than the value of `min_diff_curr_day[i]`.

After popping all `j` from the stack where `jobDifficulty[j] <= jobDifficulty[i]` (scenario 1), if the stack is not empty, we will check scenario 2 once. Finally, we can push `i` onto the stack since we may consider extending the job schedule that ends at `i` later.

Now, let's take a look at how to maintain the monotonic stack and summarize the steps of the algorithm.

Algorithm

1. We will need two arrays, `min_diff_curr_day` and `min_diff_prev_day` to record the minimum total job difficulty of the current day and the previous day (one more day remaining).

For instance, `min_diff_curr_day[i]` records the minimum total job difficulty after completing the ithi^{th}*ith* job on the current day. Since we need to complete at lease one job on each day, when we are at day 0 (with `d` remaining days), we can only perform the 0th0^{th}0*t**h* job, so `min_diff_curr_day[i]` is `jobDifficulty[0]`. When we are not at the very first day, then we must have previously completed at least day−1thday-1^{th}*day*−1*t**h* jobs. On the current day, by completing the i=daythi = day^{th}*i*=*dayth* job, `min_diff_curr_day[i]` will be `min_diff_prev_day[i-1] + jobDifficulty[i]`.

Refer to the slides below for two examples of initialization at `day = 0` and `day = 1`, respectively.

1. We will then iterate through each day and update the `min_diff_curr_day` along the way for each possible last job scheduled on that day.

Specifically, we iterate through jobs with indices ranging from `i` equals `day` to `n`. To maintain a monotonically decreasing stack, if the job difficulty of the last element in the stack (`j`) is smaller than or equal to the current job (`i`), we will pop the last element (`j`) from the stack.

Since `jobDifficulty[j] <= jobDifficulty[i]`, by additionally completing jobs from index `j + 1` to index `i`, the total job difficulty `min_diff_curr_day[i]` will be no more than `min_diff_curr_day[j] + jobDifficulty[i] - jobDifficulty[j]`. So, we can update `min_diff_curr_day[i]` as `min(min_diff_curr_day[i], min_diff_curr_day[j] + jobDifficulty[i] - jobDifficulty[j])`

1. On each day, after we iterate through each indices `i`, popping all indices `j` off the stack where `jobDifficulty[j] <= jobDifficulty[i]`, and updating `min_diff_curr_day[i]`, there may be some remaining indices in the monotonically decreasing stack. Since we are using a monotonically decreasing stack, we know that `jobDifficulty[j] > jobDifficulty[i]` for all of the remaining indices in the stack. Until now, while updating `min_diff_curr_day[i]`, we only considered scenario 1 (when `jobDifficulty[j] <= jobDifficulty[i]`). Before we move to the next day, we must consider scenario 2 as well. Thus, if the stack is not empty, we compare the minimum difficulty job schedule ending at job `j` where `j` is on the top of the stack, and the total job difficulty in the current day `min_diff_curr_day[i]` will be no more than `min_diff_curr_day[j]`. So, we can update `min_diff_curr_day[i]` as `min(min_diff_curr_day[i], min_diff_curr_day[j])`.
2. At the end, we will return the total minimum job difficulty on the dthd^{th}*dth* day after completing the n−1thn-1^{th}*n*−1*t**h* job, which is `min_diff_prev_day[n - 1]`.

Refer to the slides below for how to update the DP states with the monotonically decreasing stack step by step. The height of each circle corresponds to the job difficulty.

Complexity Analysis

Let nn$n$ be the length of the `jobDifficulty` array, and dd$d$ be the total number of days.

- Time complexity: O(n⋅d)O(n \cdot d)$O(n \cdot d)$ as there are n⋅dn \cdot d$n \cdot d$ possible states. Using the stack solution, we need O(n)O(n)$O(n)$ time to calculate all nn$n$ states for each day.
- Space complexity: O(n)O(n)$O(n)$ as we only use one array of length `n` to store all DP states for the prior day and the current day, and the stack that will contain at most `n` elements.

If you are interested in practicing the skill to optimize the DP solution using a monotonic stack, you may want to try the following two LeetCode problems.