In this article, we'll be using the framework to solve [Minimum Difficulty of a Job Schedule](https://leetcode.com/problems/minimum-difficulty-of-a-job-schedule/). We can tell this is a problem where Dynamic Programming can be used because we are asked for the minimum of something, and deciding how many jobs to do on a given day affects the jobs we can do on all future days. Let's start solving:

1. A **function** that answers the problem for a given state

Let's first decide on state variables. What decisions are there to make, and what information do we need to make these decisions? Reading the problem description carefully, there are d total days, and on each day we need to complete some number of jobs. By the end of the d days, we must have finished all jobs (in the given order). Therefore, we can see that on each day, we need to decide how many jobs to take.

* Let's use one state variable i, where i is the index of the first job that will be done on the current day.
* Let's use another state variable day, where day indicates what day it currently is.

The problem is asking for the minimum difficulty, so let's have a function dp(i, day) that returns the minimum difficulty of a job schedule which starts on the  *ith* job and day, day. To solve the original problem, we will just return dp(0, 1), since we start on the first day with no jobs done yet.

Table

Description automatically generated

Text

Description automatically generated

2. A **recurrence relation** to transition between states

At each state, we are on day day and need to do job i. Then, we can choose to do a few more jobs. How many more jobs are we allowed to do? The problem says that we need to do at least one job per day. This means we **must** leave at least d - day jobs so that all the future days have at least one job that can be scheduled on that day. If n is the total number of jobs, jobDifficulty.length, that means from any given state (i, day), we are allowed to do the jobs from index i up to but not including index n - (d - day).

We should try all the options for a given day - try doing only one job, then two jobs, etc. until we can't do any more jobs. The best option is the one that results in the easiest job schedule.

The difficulty of a given day is the most difficult job that we did that day. Since the jobs have to be done in order, if we are trying all the jobs we are allowed to do on that day (iterating through them), then we can use a variable hardest to keep track of the difficulty of the hardest job done today. If we choose to do jobs up to the *jth* job (inclusive), where i≤j<n - (d - day) (as derived above), then that means on the next day, we start with the  (*j*+1)*th* job. Therefore, our total difficulty is hardest + dp(j + 1, day + 1). This gives us our scariest recurrence relation so far:

dp(i, day) = min(hardest + dp(j + 1, day + 1)) for all i ≤ j<n - (d - day), where hardest = max(jobDifficulty[k]) for all i≤k≤j.

The codified recurrence relation is a scary one to look at for sure. However, it is easier to understand when we break it down bit by bit. On each day, we try all the options - do only one job, then two jobs, etc. until we can't do any more (since we need to leave some jobs for future days). hardest is the hardest job we do on the current day, which means it is also the difficulty of the current day. We add hardest to the next state which is the next day, starting with the next job. After trying all the jobs we are allowed to do, choose the best result.

Table

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

3. **Base cases**

Despite the recurrence relation being complicated, the base cases are much simpler. We need to finish all jobs in d days. Therefore, if it is the last day (day == d), we need to finish up all the remaining jobs on this day, and the total difficulty will just be the largest number in jobDifficulty on or after index i.

if day == d then return the maximum job difficulty between job i and the end of the array (inclusive).

We can precompute an array hardestJobRemaining where hardestJobRemaining[i] represents the difficulty of the hardest job on or after day i, so that we this base case is handled in constant time.

Additionally, if there are more days than jobs (n < d), then it is impossible to do at least one job each day, and per the problem description, we should return -1. We can check for this case at the very start.

### Top-down Implementation

Let's combine these 3 parts for a top-down implementation. Again, we will use [functools](https://docs.python.org/3/library/functools.html) in Python, and a 2D array in Java for memoization. In the Python implementation, we are passing None to lru\_cache which means the cache size is not limited. We are doing this because the number of states that will be re-used in this problem is large, and we don't want to evict a state early and have to re-calculate it.

Bottom-up Implementation

With bottom-up, we now use a 2D array where dp[i][day] represents the minimum difficulty of a job schedule that starts on day day and job i. It depends on the problem, but the bottom-up code generally has a faster runtime than its top-down equivalent. However, as you can see from the code, it looks like it is more challenging to implement. We need to first tabulate the base case and then work backwards from them using nested for loops.

The for-loops should start iterating from the base cases, and there should be one for-loop for each state variable. Remember that one of our base cases is that on the final day, we need to complete all jobs. Therefore, our for-loop iterating over day should iterate from the final day to the first day. Then, our next for-loop for i should conform to the restraints of the problem - we need to do at least one job per day.

Here's an animation showing the algorithm in action:

Table

Description automatically generated

Table

Description automatically generated

3A chart with numbers and letters

Description automatically generated with low confidence4A picture containing calendar

Description automatically generated5A picture containing calendar

Description automatically generated

6Text

Description automatically generated with low confidence7A picture containing calendar

Description automatically generated8A picture containing table

Description automatically generated

9Table

Description automatically generated

10Table

Description automatically generated

11Table

Description automatically generated

12Table

Description automatically generated

13Table

Description automatically generated

14Text, table

Description automatically generated

15Text, table

Description automatically generated

16Table

Description automatically generated

17Table

Description automatically generated

18Table

Description automatically generated with medium confidence

19Table

Description automatically generated with medium confidence

20Table

Description automatically generated with medium confidence

21Table

Description automatically generated with medium confidence

22Table

Description automatically generated with medium confidence

23Table

Description automatically generated with medium confidence

24Table

Description automatically generated with medium confidence

25Table

Description automatically generated

26Text, table

Description automatically generated with medium confidence

27Table

Description automatically generated

28Text, table

Description automatically generated

29Table

Description automatically generated with medium confidence

30Text, table

Description automatically generated

The time and space complexity of these algorithms can be quite tricky, and as in this example, there are sometimes slight differences between the top-down and bottom-up complexities.

Let's start with the bottom-up space complexity, because it follows what we learned in the previous chapter about finding time and space complexity. For this problem, the number of states is *n*⋅*d*. This means the space complexity is *O*(*n*⋅*d*) as our dp table takes up that much space.

The top-down algorithm's space complexity is actually a bit better. In top-down, when we memoize results with a hashtable, the hashtable's size only grows when we visit a state and calculate the answer for it. Because of the restriction of needing to complete at least one task per day, we don't actually need to visit all *n*⋅*d* states. For example, if there were 10 jobs and 5 days, then the state (9, 2) (starting the final job on the second day) is not reachable, because the 3rd, 4th, and 5th days wouldn't have a job to complete. This is true for both implementations and is enforced by our for-loops, and as a result, we only actually visit *d*⋅(*n*−*d*) states. This means the space complexity for top-down is *O*(*d*⋅(*n*−*d*)). This is one advantage that top-down can have over bottom-up. With the bottom-up implementation, we can't really avoid allocating space for *n*⋅*d* states because we are using a 2D array.

The time complexity for both algorithms is more complicated. As we just found out, we only actually visit *d*⋅(*n*−*d*) states. At each state, we go through a for-loop (with variable j) that iterates on average times. This means our time complexity for both algorithms is *O*(*d*⋅).

**To summarize:**

Time complexity (both algorithms): *O*(*d*⋅).

Space complexity (top-down): *O*((*n*−*d*)⋅*d*)

Space complexity (bottom-up): *O*(*n*⋅*d*)

While the theoretical space complexity is better with top-down, practically speaking, the 2D array is more space-efficient than a hashmap, and the difference in space complexities here doesn't justify saying that top-down will use less space than bottom-up.

**322. Coin Change**

Let dp[i] represent the fewest number of coins needed to make up i money.

For each coin available, we can get to i from i - coin. Make sure that coin ≤ i.

According to the constraints, the minimum value for a coin is 1. Therefore, it is impossible to have 0 money, therefore dp[0] = 0.