Dynamic Programming (DP) is a programming paradigm that can systematically and efficiently explore all possible solutions to a problem. As such, it is capable of solving a wide variety of problems that often have the following characteristics:

1. The problem can be broken down into "overlapping subproblems" - smaller versions of the original problem that are re-used multiple times.
2. The problem has an "optimal substructure" - an optimal solution can be formed from optimal solutions to the overlapping subproblems of the original problem.

As a beginner, these theoretical definitions may be hard to wrap your head around. Don't worry though - at the end of this chapter, we'll talk about how to practically spot when DP is applicable. For now, let's look a little deeper at both characteristics.

The Fibonacci sequence is a classic example used to explain DP. For those who are unfamiliar with the Fibonacci sequence, it is a sequence of numbers that starts with 0, 1, and each subsequent number is obtained by adding the previous two numbers together.

If you wanted to find the n^{th} Fibonacci number $F(n)$, you can break it down into smaller **subproblems** - find $F(n-1)$ and $F(n-2)$ instead. Then, adding the solutions to these subproblems together gives the answer to the original question, $F(n-1) + F(n-2) = F(n)$, which means the problem has **optimal substructure**, since a solution $F(n)$ to the original problem can be formed from the solutions to the subproblems. These subproblems are also **overlapping** - for example, we would need $F(4)$ to calculate both $F(5)$ and $F(6)$.

These attributes may seem familiar to you. Greedy problems have optimal substructure, but not overlapping subproblems. Divide and conquer algorithms break a problem into subproblems, but these subproblems are not **overlapping** (which is why DP and divide and conquer are commonly mistaken for one another).

Dynamic programming is a powerful tool because it can break a complex problem into manageable subproblems, avoid unnecessary recalculation of overlapping subproblems, and use the results of those subproblems to solve the initial complex problem. DP not only aids us in solving complex problems, but it also greatly improves the time complexity compared to brute force solutions. For example, the brute force solution for calculating the Fibonacci sequence has exponential time complexity, while the dynamic programming solution will have linear time complexity. Throughout this explore card, you will gain a better understanding of what makes DP so powerful. In the next section, we'll discuss the two main methods of implementing a DP algorithm.

A Top-down and Bottom-up

There are two ways to implement a DP algorithm:

1. Bottom-up, also known as tabulation.
2. Top-down, also known as memoization.

Let's take a quick look at each method.

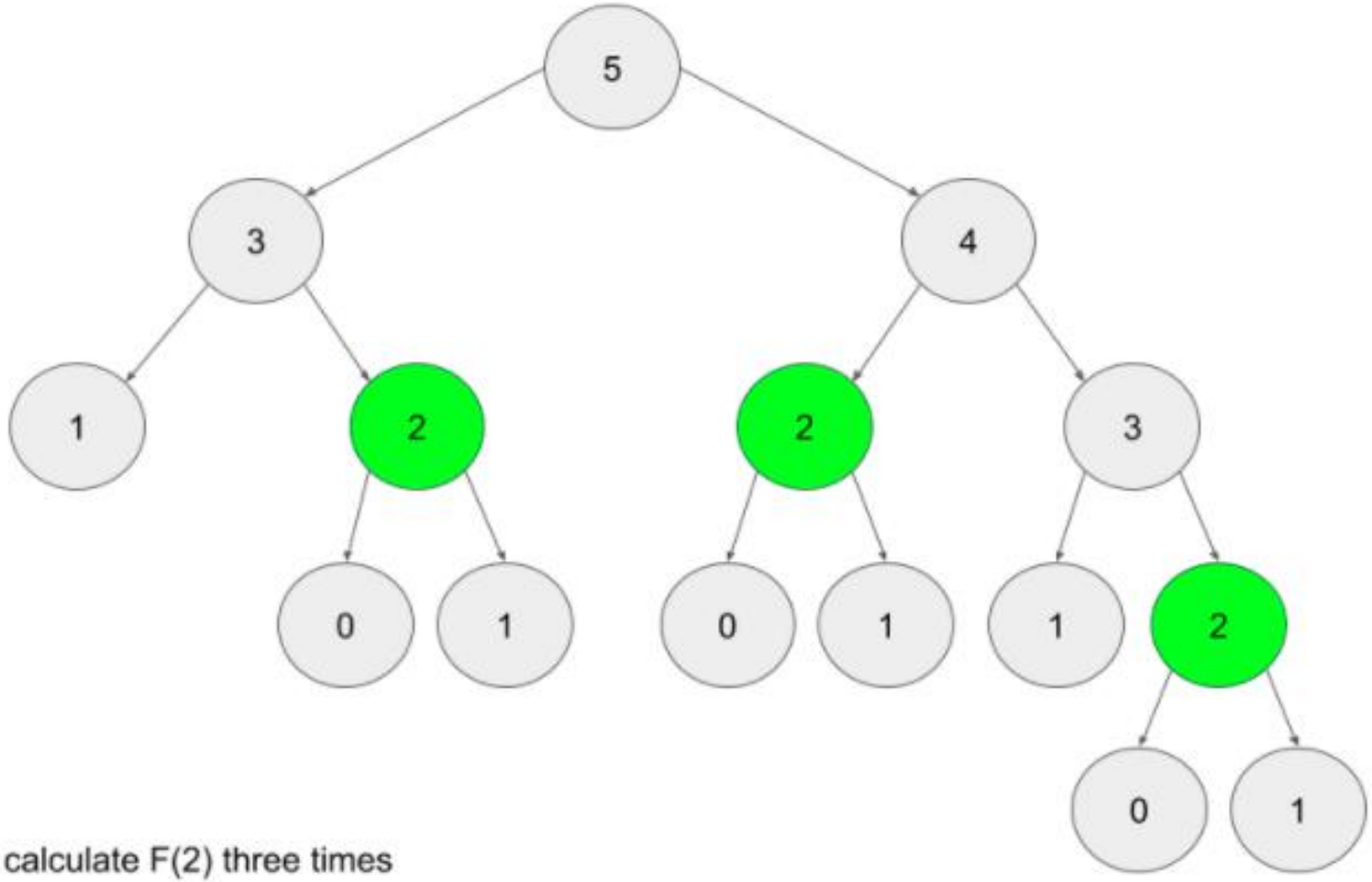
Bottom-up (Tabulation)

Bottom-up is implemented with iteration and starts at the base cases. Let's use the Fibonacci sequence as an example again. The base cases for the Fibonacci sequence are $F(0) = 0$ and $F(1) = 1$. With bottom-up, we would use these base cases to calculate $F(2)$, and then use that result to calculate $F(3)$, and so on all the way up to $F(n)$.

```
// Pseudocode example for bottom-up

F = array of length (n + 1)
F[0] = 0
F[1] = 1
for i from 2 to n:
    F[i] = F[i - 1] + F[i - 2]
```

Top-down (Memoization)

Top-down is implemented with recursion and made efficient with memoization. If we wanted to find the n^{th} Fibonacci number $F(n)$, we try to compute this by finding $F(n-1)$ and $F(n-2)$. This defines a recursive pattern that will continue on until we reach the base cases $F(0) = F(1) = 1$. The problem with just implementing it recursively is that there is a ton of unnecessary repeated computation. Take a look at the recursion tree if we were to find $F(5)$:



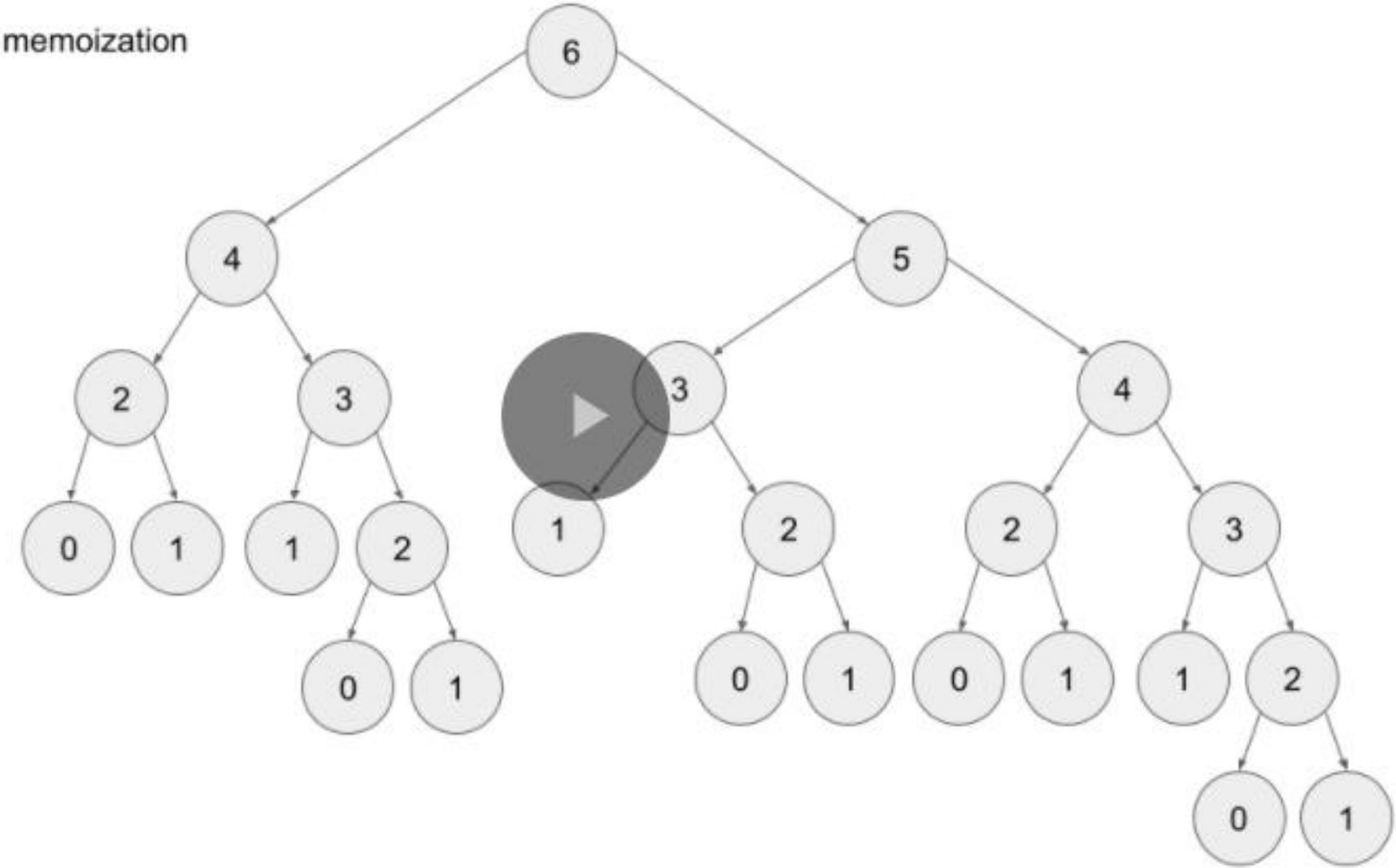
We have to calculate F(2) three times

Notice that we need to calculate $F(2)$ three times. This might not seem like a big deal, but if we were to calculate $F(6)$, this **entire image** would be only one child of the root. Imagine if we wanted to find $F(100)$ - the amount of computation is exponential and will quickly explode. The solution to this is to **memoize** results.

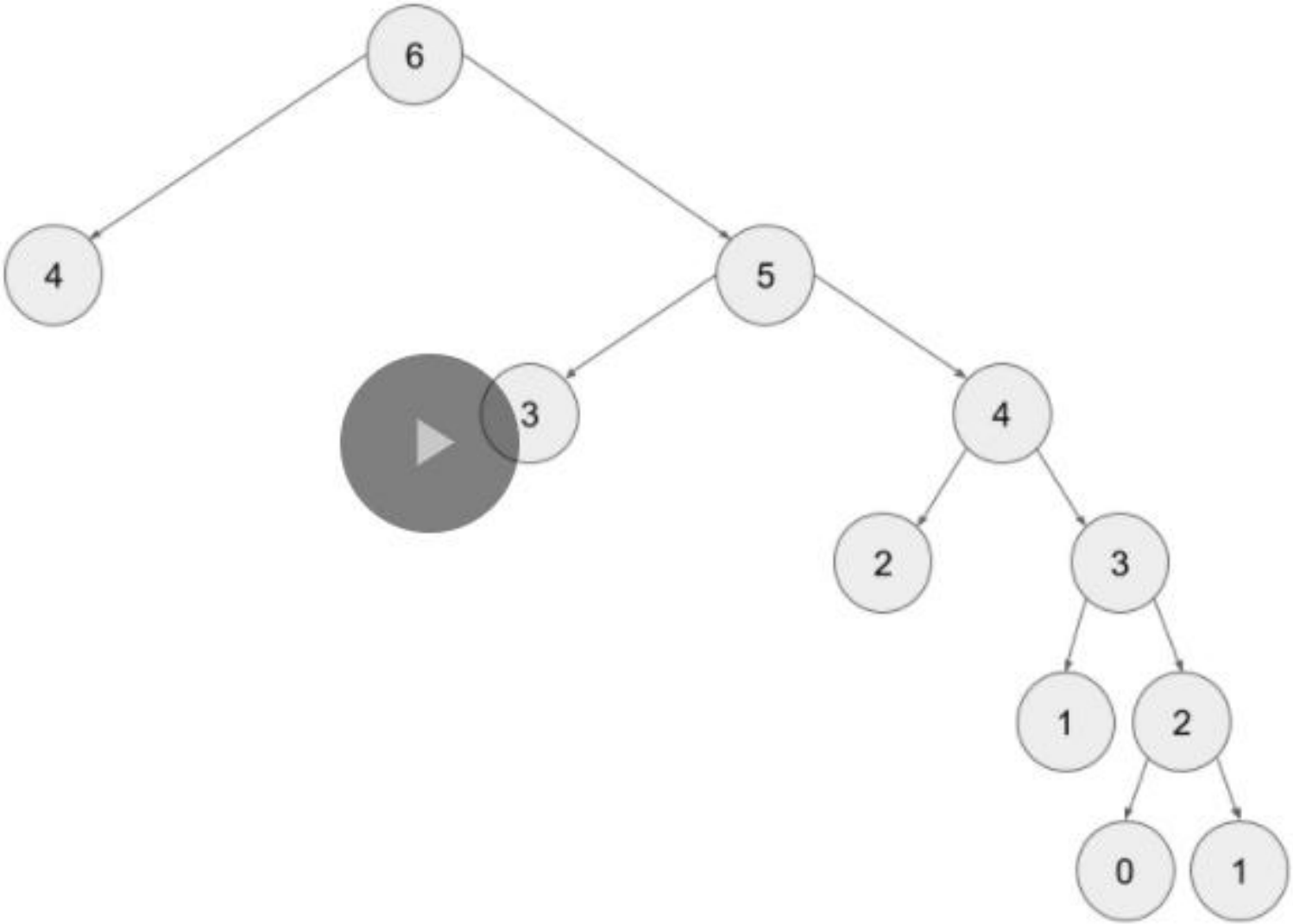
> **memoizing** a result means to store the result of a function call, usually in a hashmap or an array, so that when the same function call is made again, we can simply return the **memoized** result instead of recalculating the result.

After we calculate $F(2)$, let's store it somewhere (typically in a hashmap), so in the future, whenever we need to find $F(2)$, we can just refer to the value we already calculated instead of having to go through the entire tree again. Below is an example of what the recursion tree for finding $F(6)$ looks like with and without memoization:

F(6) without memoization



F(6) with memoization



```
// Pseudocode example for top-down

memo = hashmap
Function F(integer i):
    if i is 0 or 1:
        return i
    if i doesn't exist in memo:
        memo[i] = F(i - 1) + F(i - 2)
    return memo[i]
```

Which is better?

Any DP algorithm can be implemented with either method, and there are reasons for choosing either over the other. However, each method has one main advantage that stands out:

- A bottom-up implementation's runtime is usually faster, as iteration does not have the overhead that recursion does.
- A top-down implementation is usually much easier to write. This is because with recursion, the ordering of subproblems does not matter, whereas with tabulation, we need to go through a logical ordering of solving subproblems.

When to Use DP

Report Issue

When it comes to solving an algorithm problem, especially in a high-pressure scenario such as an interview, half the battle is figuring out how to even approach the problem. In the first section, we defined what makes a problem a good candidate for dynamic programming. Recall:

1. The problem can be broken down into "overlapping subproblems" - smaller versions of the original problem that are re-used multiple times
2. The problem has an "optimal substructure" - an optimal solution can be formed from optimal solutions to the overlapping subproblems of the original problem

Unfortunately, it is hard to identify when a problem fits into these definitions. Instead, let's discuss some common characteristics of DP problems that are easy to identify.

The first characteristic that is common in DP problems is that the problem will ask for the optimum value (maximum or minimum) of something, or the number of ways there are to do something. For example:

- What is the minimum cost of doing...
- What is the maximum profit from...
- How many ways are there to do...
- What is the longest possible...
- Is it possible to reach a certain point...

> **Note:** Not all DP problems follow this format, and not all problems that follow these formats should be solved using DP. However, these formats are very common for DP problems and are generally a hint that you should consider using dynamic programming.

When it comes to identifying if a problem should be solved with DP, this first characteristic is not sufficient. Sometimes, a problem in this format (asking for the max/min/longest etc.) is meant to be solved with a greedy algorithm. The next characteristic will help us determine whether a problem should be solved using a greedy algorithm or dynamic programming.

The second characteristic that is common in DP problems is that future "decisions" depend on earlier decisions. Deciding to do something at one step may affect the ability to do something in a later step. This characteristic is what makes a greedy algorithm invalid for a DP problem - we need to factor in results from previous decisions. Admittedly, this characteristic is not as well defined as the first one, and the best way to identify it is to go through some examples.

House Robber is an excellent example of a dynamic programming problem. The problem description is:

> You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.
>
> Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

In this problem, each decision will affect what options are available to the robber in the future. For example, with the test case $\text{nums} = [2, 7, 9, 3, 1]$, the optimal solution is to rob the houses with 2, 9, and 1 money. However, if we were to iterate from left to right in a greedy manner, our first decision would be whether to rob the first or second house. 7 is way more money than 2, so if we were greedy, we would choose to rob house 7. However, this prevents us from robbing the house with 9 money. As you can see, our decision between robbing the first or second house affects which options are available for future decisions.

Longest Increasing Subsequence is another example of a classic dynamic programming problem. In this problem, we need to determine the length of the longest (first characteristic) subsequence that is strictly increasing. For example, if we had the input $\text{nums} = [1, 2, 6, 3, 5]$, the answer would be 4, from the subsequence $[1, 2, 3, 5]$. Again, the important decision comes when we arrive at the 6 - do we take it or not take it? If we decide to take it, then we get to increase our current length by 1, but it affects the future - we can no longer take the 3 or 5. Of course, with such a small example, it's easy to see why we shouldn't take it - but how are we supposed to design an algorithm that can always make the correct decision with huge inputs? Imagine if nums contained $10,000$ numbers instead.

> When you're solving a problem on your own and trying to decide if the second characteristic is applicable, assume it isn't, then try to think of a counterexample that proves a greedy algorithm won't work. If you can think of an example where earlier decisions affect future decisions, then DP is applicable.

To summarize: if a problem is asking for the maximum/minimum/longest/shortest of something, the number of ways to do something, or if it is possible to reach a certain point, it is probably greedy or DP. With time and practice, it will become easier to identify which is the better approach for a given problem. Although, in general, if the problem has constraints that cause decisions to affect other decisions, such as using one element prevents the usage of other elements, then we should consider using dynamic programming to solve the problem. **These two characteristics can be used to identify if a problem should be solved with DP.**

> Note: these characteristics should only be used as guidelines - while they are extremely common in DP problems, at the end of the day DP is a very broad topic.

A Framework for DP Problems

Report Issue

Now that we understand the basics of DP and how to spot when DP is applicable to a problem, we've reached the most important part: actually solving the problem. In this section, we're going to talk about a framework for solving DP problems. This framework is applicable to nearly every DP problem and provides a clear step-by-step approach to developing DP algorithms.

> For this article's explanation, we're going to use the problem Climbing Stairs as an example, with a top-down (recursive) implementation. Take a moment to read the problem description and understand what the problem is asking.

Before we start, we need to first define a term: **state**. In a DP problem, a **state** is a set of variables that can sufficiently describe a scenario. These variables are called **state variables**, and we only care about relevant ones. For example, to describe every scenario in Climbing Stairs, there is only 1 relevant state variable, the current step we are on. We can denote this with an integer i. If $\mathrm{i} = 6$, that means that we are describing the state of being on the 6th step. Every unique value of i represents a unique **state**.

> You might be wondering what "relevant" means here. Picture this problem in real life: you are on a set of stairs, and you want to know how many ways there are to climb to say, the 10th step. We're definitely interested in what step you're currently standing on. However, we aren't interested in what color your socks are. You could certainly include sock color as a state variable. Standing on the 8th step wearing green socks is a different state than standing on the 8th step wearing red socks. However, changing the color of your socks will not change the number of ways to reach the 10th step from your current position. Thus the color of your socks is an **irrelevant** variable. In terms of figuring out how many ways there are to climb the set of stairs, the only **relevant** variable is what stair you are currently on.

The Framework

To solve a DP problem, we need to combine 3 things:

1. **A function or data structure that will compute/contain the answer to the problem for every given state**.

   For Climbing Stairs, let's say we have an function dp where $\mathrm{dp(i)}$ returns the number of ways to climb to the i^{th} step. Solving the original problem would be as easy as $\mathrm{return\ dp(n)}$.

   How did we decide on the design of the function? The problem is asking "How many distinct ways can you climb to the top?", so we decide that the function will represent how many distinct ways you can climb to a certain step - literally the original problem, but generalized for a given state.

> Typically, top-down is implemented with a recursive function and hash map, whereas bottom-up is implemented with nested for loops and an array. When designing this function or array, we also need to decide on state variables to pass as arguments. This problem is very simple, so all we need to describe a state is to know what step we are currently on i. We'll see later that other problems have more complex states.

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

   A recurrence relation is an equation that relates different states with each other. Let's say that we needed to find how many ways we can climb to the 30th stair. Well, the problem states that we are allowed to take either 1 or 2 steps at a time. Logically, that means to climb to the 30th stair, we arrived from either the 28th or 29th stair. Therefore, the number of ways we can climb to the 30th stair is equal to the number of ways we can climb to the 28th stair plus the number of ways we can climb to the 29th stair.

   The problem is, we don't know how many ways there are to climb to the 28th or 29th stair. However, we can use the logic from above to define a recurrence relation. In this case, $dp(i) = dp(i - 1) + dp(i - 2)$. As you can see, information about some states gives us information about other states.

> Upon careful inspection, we can see that this problem is actually the Fibonacci sequence in disguise! This is a very simple recurrence relation - typically, finding the recurrence relation is the most difficult part of solving a DP problem. We'll see later how some recurrence relations are much more complicated, and talk through how to derive them.

3. **Base cases, so that our recurrence relation doesn't go on infinitely.**

   The equation $\mathrm{dp(i)} = \mathrm{dp(i - 1)} + \mathrm{dp(i - 2)}$ on its own will continue forever to negative infinity. We need base cases so that the function will eventually return an actual number.

   Finding the base cases is often the easiest part of solving a DP problem, and just involves a little bit of logical thinking. When coming up with the base case(s) ask yourself: What state(s) can I find the answer to without using dynamic programming? In this example, we can reason that there is only 1 way to climb to the first stair (1 step once), and there are 2 ways to climb to the second stair (1 step twice and 2 steps once). Therefore, our base cases are $\mathrm{dp}(1) = 1$ and $\mathrm{dp}(2) = 2$.

   > We said above that we don't know how many ways there are to climb to the 28th and 29th stairs. However, using these base cases and the recurrence relation from step 2, we can figure out how many ways there are to climb to the 3rd stair. With that information, we can find out how many ways there are to climb to the 4th stair, and so on. Eventually, we will know how many ways there are to climb to the 28th and 29th stairs.

Example Implementations

Here is a basic top-down implementation using the 3 components from the framework:

Java | Python3

Copy

```java
class Solution {
    // A function that represents the answer to the problem for a given state
    private int dp(int i) {
        if (i <= 2) return i; // Base cases
        return dp(i - 1) + dp(i - 2); // Recurrence relation
    }

    public int climbStairs(int n) {
        return dp(n);
    }
}
```

Do you notice something missing from the code? We haven't memoized anything! The code above has a time complexity of $O(2^n)$ because every call to dp creates 2 more calls to dp. If we wanted to find how many ways there are to climb to the 250th step, the number of operations we would have to do is approximately equal to the number of atoms in the universe.

In fact, without the memoization, this isn't actually dynamic programming - it's just basic recursion. Only after we optimize our solution by adding memoization to avoid repeated computations can it be called DP. As explained in chapter 1, memoization means caching results from function calls and then referring to those results in the future instead of recalculating them. This is usually done with a hashmap or an array.

Java | Python3

Copy

```java
class Solution {
    private HashMap<Integer, Integer> memo = new HashMap<>();

    private int dp(int i) {
        if (i <= 2) return i;
        // Instead of just returning dp(i - 1) + dp(i - 2), calculate it once and then
        // store it inside a hashmap to refer to in the future
        if (!memo.containsKey(i)) {
            memo.put(i, dp(i - 1) + dp(i - 2));
        }

        return memo.get(i);
    }

    public int climbStairs(int n) {
        return dp(n);
    }
}
```

With memoization, our time complexity drops to $O(n)$ - astronomically better, literally.

> You may notice that a hashmap is overkill for caching here, and an array can be used instead. This is true, but using a hashmap isn't necessarily bad practice as some DP problems will require one, and they're hassle-free to use as you don't need to worry about sizing an array correctly. Furthermore, when using top-down DP, some problems do not require us to solve every single subproblem, in which case an array may use more memory than a hashmap.

We just talked a whole lot about top-down, but what about bottom-up? Everything is pretty much the same, except we will start from our base cases and iterate up to our final answer. As stated before, bottom-up implementations usually use an array, so we will use an array dp where $\mathrm{dp}[\mathrm{i}]$ represents the number of ways to climb to the i^{th} step.

Java | Python3

Copy

```java
class Solution {
    public int climbStairs(int n) {
        if (n == 1) return 1;

        // An array that represents the answer to the problem for a given state
        int[] dp = new int[n + 1];
        dp[1] = 1; // Base cases
        dp[2] = 2; // Base cases
        for (int i = 3; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2]; // Recurrence relation
        }

        return dp[n];
    }
}
```

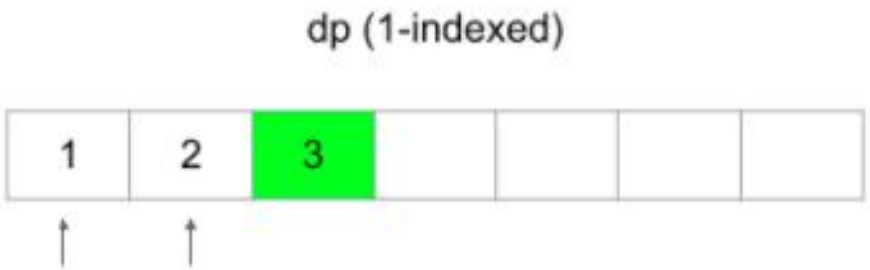
> Notice that the implementation still follows the framework exactly - the framework holds for both top-down and bottom-up implementations.

To Summarize

With DP problems, we can use logical thinking to find the answer to the original problem for certain inputs, in this case we reason that there is 1 way to climb to the first stair and 2 ways to climb to the second stair. We can then use a recurrence relation to find the answer to the original problem for any state, in this case for any stair number. Finding the recurrence relation involves thinking about how moving from one state to another changes the answer to the problem.

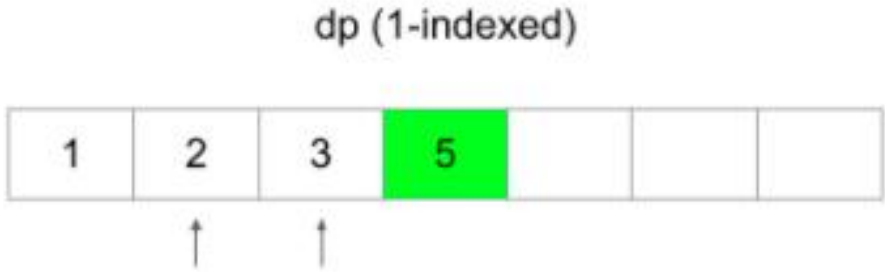
This is the essence of dynamic programming. Here's a quick animation for Climbing Stairs:

This is the essence of dynamic programming. Here's a quick animation for Climbing Stairs:



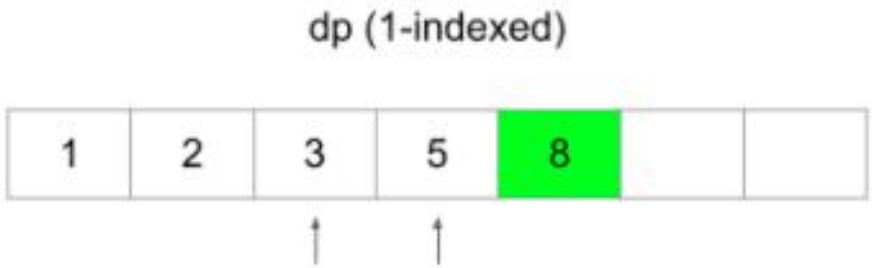
We are allowed to take either 1 or 2 steps. That means we can arrive at the 3rd step from either the first or second step. Therefore, the number of ways to climb to the 3rd stair is dp[1] + dp[2].

This is the essence of dynamic programming. Here's a quick animation for Climbing Stairs:



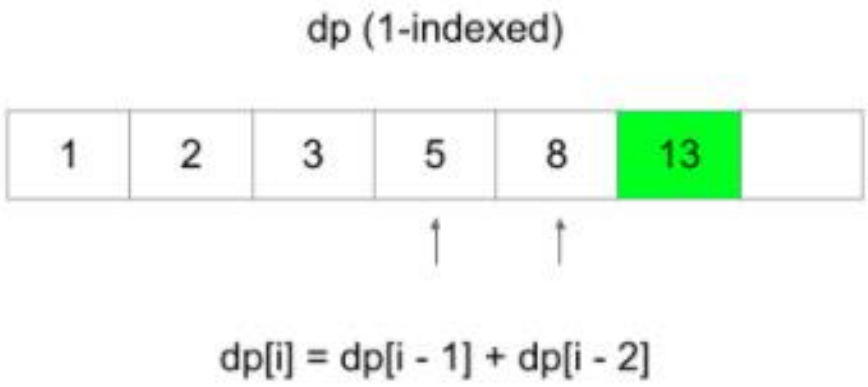
Initially, we didn't know how many ways there were to climb to the 3rd stair, but now we do. With this knowledge, we can calculate the number of ways to get to the 4th stair - dp[2] + dp[3].

This is the essence of dynamic programming. Here's a quick animation for Climbing Stairs:

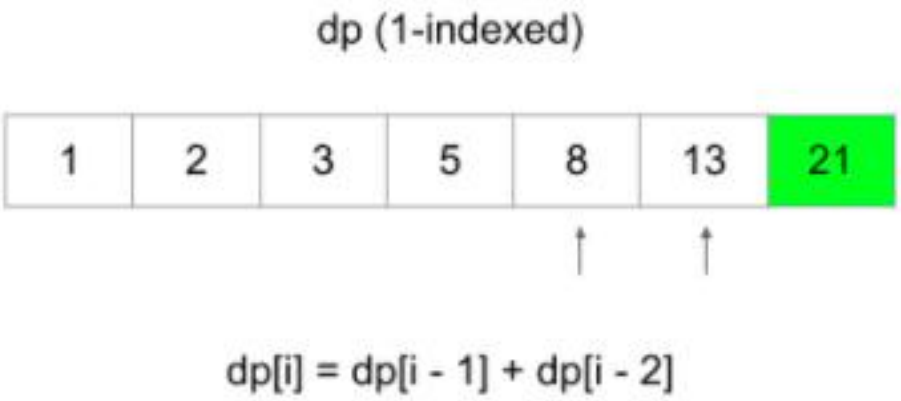


The process continues as long as we want it to - we can find the answer for any step.

This is the essence of dynamic programming. Here's a quick animation for Climbing Stairs:



This is the essence of dynamic programming. Here's a quick animation for Climbing Stairs:



Example 198. House Robber

Report Issue

> This is the first of 6 articles where we will use a framework to work through example DP problems. The framework provides a blueprint to solve DP problems, but when you are just starting to learn DP, deriving some of the logic yourself may be difficult. The objective of these articles is to talk through how to use the framework to work through each problem, and our goal is that, by the end of this, you will be able to independently tackle most DP problems using this framework.

In this article, we will be looking at the House Robber problem. In an earlier section of this explore card, we talked about how House Robber fits the characteristics of a DP problem. It's asking for the maximum of something, and our current decisions will affect which options are available for our future decisions. Let's see how we can use the framework to develop an algorithm for this problem.

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

First, we need to decide on state variables. As a reminder, state variables should be fully capable of describing a scenario. Imagine if you had this scenario in real life - you're a robber and you have a lineup of houses. If you are at one of the houses, the only variable you would need to describe your situation is an integer - the index of the house you are currently at. Therefore, the only state variable is an integer, say `i`, that indicates the index of a house.

> If the problem had an added constraint such as "*you are only allowed to rob up to k houses*", then k would be another necessary state variable. This is because being at, say house 4 with 3 robberies left is different than being at house 4 with 5 robberies left.
>
> You may be wondering - why don't we include a state variable that is a boolean indicating if we robbed the previous house or not? We certainly could include this state variable, but we can develop our recurrence relation in a way that makes it unnecessary. Building an intuition for this is difficult at first, but it becomes easier with practice.

The problem is asking for "the maximum amount of money you can rob". Therefore, we would use either a function $\mathrm{dp}(\mathrm{i})$ that returns the maximum amount of money you can rob up to and including house i, or an array dp where $\mathrm{dp}[\mathrm{i}]$ represents the maximum amount of money you can rob up to and including house i.

This means that after all the subproblems have been solved, $\mathrm{dp}[\mathrm{i}]$ and $\mathrm{dp}(\mathrm{i})$ both return the answer to the original problem for the subarray of nums that spans 0 to i inclusive. To solve the original problem, we will just need to return $\mathrm{dp}[\mathrm{nums.length} - 1]$ or $\mathrm{dp}(\mathrm{nums.length} - 1)$, depending if we do bottom-up or top-down.

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

> For this part, let's assume we are using a top-down (recursive function) approach. Note that the top-down approach is closer to our natural way of thinking and it is generally easier to think of the recurrence relation if we start with a top-down approach.

Next, we need to find a recurrence relation, which is typically the hardest part of the problem. For any recurrence relation, a good place to start is to think about a general state (in this case, let's say we're at the house at index i), and use information from the problem description to think about how other states relate to the current one.

If we are at some house, logically, we have 2 options: we can choose to rob this house, or we can choose to not rob this house.

1. If we decide not to rob the house, then we don't gain any money. Whatever money we had from the previous house is how much money we will have at this house - which is $\mathrm{dp}(\mathrm{i} - 1)$.
2. If we decide to rob the house, then we gain $\mathrm{nums}[\mathrm{i}]$ money. However, this is only possible if we did not rob the previous house. This means the money we had when arriving at this house is the money we had from the previous house without robbing it, which would be however much money we had 2 houses ago, $\mathrm{dp}(\mathrm{i} - 2)$. After robbing the current house, we will have $\mathrm{dp}(\mathrm{i} - 2) + \mathrm{nums}[\mathrm{i}]$ money.

From these two options, we always want to pick the one that gives us maximum profits. Putting it together, we have our recurrence relation: $\mathrm{dp}(\mathrm{i}) = \max(\mathrm{dp}(\mathrm{i} - 1), \mathrm{dp}(\mathrm{i} - 2) + \mathrm{nums}[\mathrm{i}])$.

3. **Base cases**

The last thing we need is base cases so that our recurrence relation knows when to stop. The base cases are often found from clues in the problem description or found using logical thinking. In this problem, if there is only one house, then the most money we can make is by robbing the house (the alternative is to not rob the house). If there are only two houses, then the most money we can make is by robbing the house with more money (since we have to choose between them). Therefore, our base cases are:

1. $\text{dp}(0) = \text{nums}[0]$
2. $\text{dp}(1) = \max(\text{nums}[0], \text{nums}[1])$

Top-down Implementation

Now that we have established all 3 parts of the framework, let's put it together for the final result. Remember: we need to memoize the function!

Java | Python3

Copy

```java
class Solution {
    private HashMap<Integer, Integer> memo = new HashMap<Integer, Integer>();
    private int[] nums;

    private int dp(int i) {
        // Base cases
        if (i == 0) return nums[0];
        if (i == 1) return Math.max(nums[0], nums[1]);
        if (!memo.containsKey(i)) {
            memo.put(i, Math.max(dp(i - 1), dp(i - 2) + nums[i])); // Recurrence relation
        }
        return memo.get(i);
    }

    public int rob(int[] nums) {
        this.nums = nums;
        return dp(nums.length - 1);
    }
}
```

Bottom-up Implementation

Here's the bottom-up approach: everything is the same, except that we use an array instead of a hash map and we iterate using a for-loop instead of using recursion.

Java | Python3

Copy

```java
class Solution {
    public int rob(int[] nums) {
        if (nums.length == 1) return nums[0];

        int[] dp = new int[nums.length];

        // Base cases
        dp[0] = nums[0];
        dp[1] = Math.max(nums[0], nums[1]);

        for (int i = 2; i < nums.length; i++) {
            dp[i] = Math.max(dp[i - 1], dp[i - 2] + nums[i]); // Recurrence relation
        }

        return dp[nums.length - 1];
    }
}
```

For both implementations, the time and space complexity is $O(n)$. We'll talk about time and space complexity of DP algorithms in depth at the end of this chapter. Here's an animation that shows the algorithm in action:

For both implementations, the time and space complexity is $O(n)$. We'll talk about time and space complexity of DP algorithms in depth at the end of this chapter. Here's an animation that shows the algorithm in action:

dp[i] represents the most money we can rob up to and including house i.

At dp[0], we can only rob the first house, so the most money we can rob is 2.

At dp[1], we can only rob one of the first two houses. Our best option is to rob the house with 7 money.

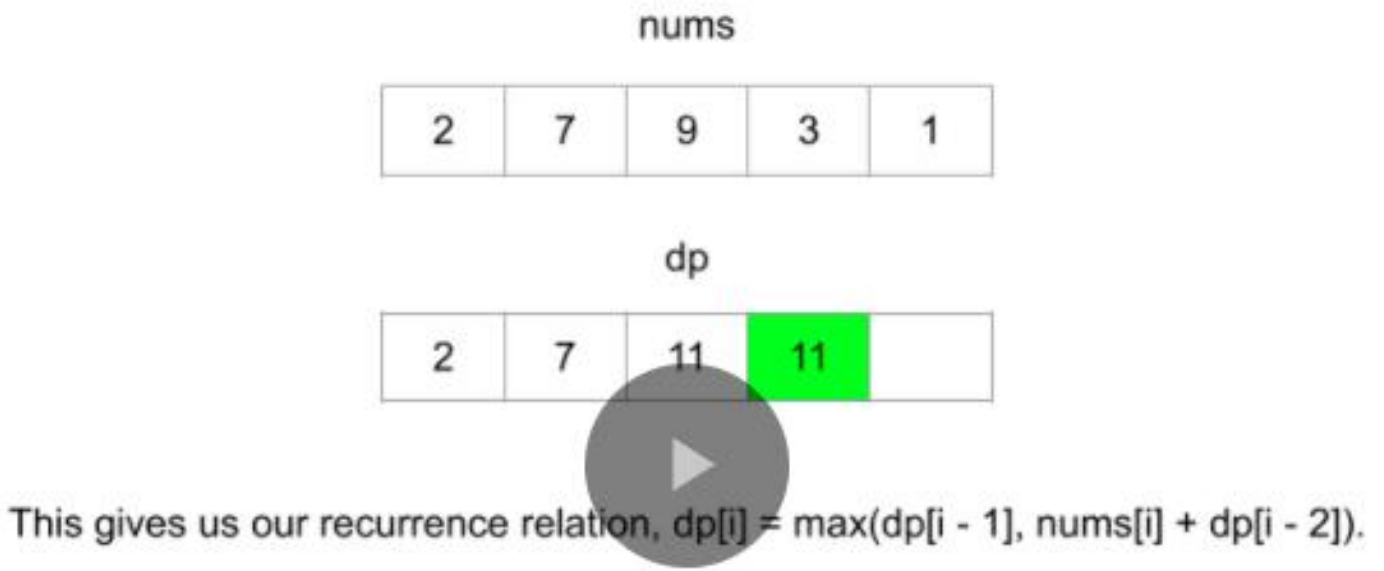
For both implementations, the time and space complexity is $O(n)$. We'll talk about time and space complexity of DP algorithms in depth at the end of this chapter. Here's an animation that shows the algorithm in action:

At the house at index 2, we have 2 options. Don't rob it, or rob it.

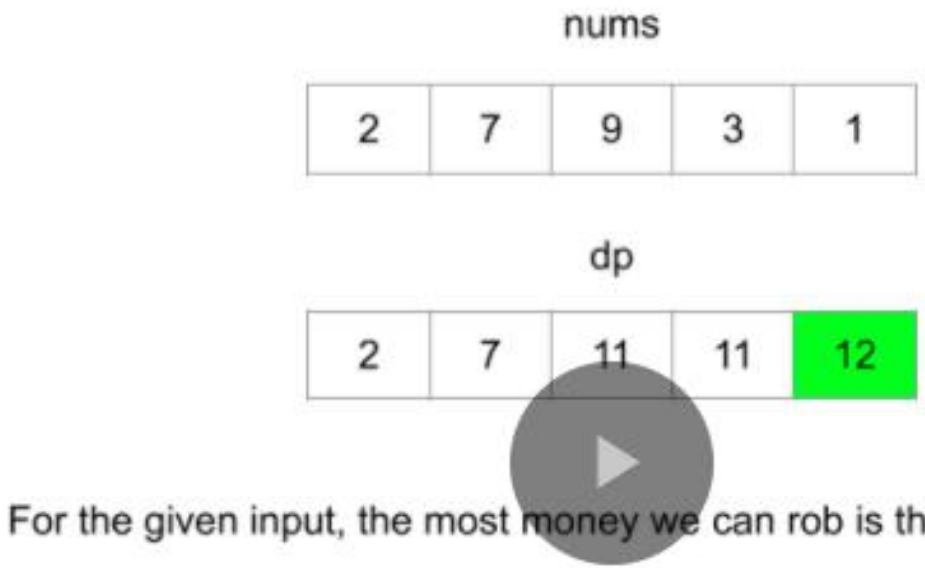
If we don't rob it, then we have the same money that we left the previous house with, 7.

If we do rob it, we gain 9 money, but we can only rob it if we did not rob the previous house. That means the money we had going into this house is the money we had 2 houses ago, 2. In total, 9 + 2 = 11, which is more than 7.

For both implementations, the time and space complexity is $O(n)$. We'll talk about time and space complexity of DP algorithms in depth at the end of this chapter. Here's an animation that shows the algorithm in action:



For both implementations, the time and space complexity is $O(n)$. We'll talk about time and space complexity of DP algorithms in depth at the end of this chapter. Here's an animation that shows the algorithm in action:



Up Next

Now that you've seen the framework in action, try solving these problems (located on the next 2 pages) on your own. If you get stuck, come back here for hints:

746. Min Cost Climbing Stairs

▼ Click here to show hint regarding state variables and `dp`

Let $\mathrm{dp}(\mathrm{i})$ be the minimum cost necessary to reach step i.

▼ Click here to show hint regarding the recurrence relation

We can arrive at step i from either step $\mathrm{i} - 1$ or step $\mathrm{i} - 2$. Choose whichever one is cheaper.

▼ Click here to show hint regarding base cases

Since we can start from either step 0 or step 1, the cost to reach these steps is 0.

1137. N-th Tribonacci Number

▼ Click here to show hint regarding state variables and `dp`

Let $\mathrm{dp(i)}$ represent the i^{th} tribonacci number.

▼ Click here to show hint regarding the recurrence relation

Use the equation given in the problem description.

▼ Click here to show hint regarding base cases

Use the base cases given in the problem description.

Use the base cases given in the problem description.

740. Delete and Earn

▼ Click here to show hint regarding preprocessing steps

Sort nums and **count** how many times each number occurs in nums.

▼ Click here to show hint regarding state variables and `dp`

Let $\text{dp}(\text{i})$ be the maximum number of points you can earn between i and the end of the sorted nums array.

▼ Click here to show hint regarding the recurrence relation

When we are at index i we have 2 options:

1. Take all numbers that match $\text{nums}[\text{i}]$ and skip all $\text{nums}[\text{i}] + 1$.
2. Do not take $\text{nums}[\text{i}]$ and move to the first occurrence of $\text{nums}[\text{i}] + 1$.

Choose whichever option yields the most points.

Bonus Hint: When is the first option guaranteed to be better than the second option?

▼ Click here to show hint regarding base cases

If we have reached the end of the nums array $(\text{i} = \text{nums.length})$ then return 0 because we cannot gain any more points.

A Example 1770. Maximum Score from Performing Multiplication Operations

Report Issue

> For this problem, we will again start by looking at a top-down approach.

In this article, we're going to be looking at the problem Maximum Score from Performing Multiplication Operations. We can tell this is a DP problem because it is asking for a maximum score, and every time we choose to use a number from nums, it affects all future possibilities. Let's solve this problem with the framework:

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

Since we're doing top-down, we need to decide on two things for our function dp. What state variables we need to pass to it, and what it will return. We are given two input arrays: nums and multipliers. The problem says we need to do m operations, and on the i^{th} operation, we gain score equal to $\text{multipliers}[\text{i}]$ times a number from either the left or right end of nums, which we remove after the operation. That means we need to know 3 things for each operation:

1. How many operations have we done so far; this tells us what number from multipliers we will be using?
2. The index of the leftmost number remaining in nums.
3. The index of the rightmost number remaining in nums.

We can use one state variable, i, to indicate how many operations we have done so far, which means $multipliers[i]$ is the current multiplier to be used. For the leftmost number remaining in $nums$, we can use another state variable, $left$, that indicates how many left operations we have done so far. If we have done, say 3 left operations, if we were to do another left operation we would use $nums[3]$ (because nums is 0-indexed). We can say the same thing for the rightmost remaining number - let's use a state variable $right$ that indicates how many right operations we have done so far.

It may seem like we need all 3 of these state variables, but we can formulate an equation for one of them using the other two. If we know how many elements we have picked from the leftside, $left$, and we know how many elements we have picked in total, i, then we know that we must have picked $i - left$ elements from the rightside. The original length of $nums$ is n, which means the index of the rightmost element is $right = n - 1 - (i - left)$. Therefore, we only need 2 state variables: i and $left$, and we can calculate $right$ inside the function.

Now that we have our state variables, what should our function return? The problem is asking for the maximum score from some number of operations, so let's have our function $dp(i, left)$ return the maximum possible score if we have already done i total operations and used $left$ numbers from the left side. To answer the original problem, we should return $dp(0, 0)$.

nums

-5	-3	-3	-2	7	1

multipliers

-10	-5	3	4	6

This is the original input.

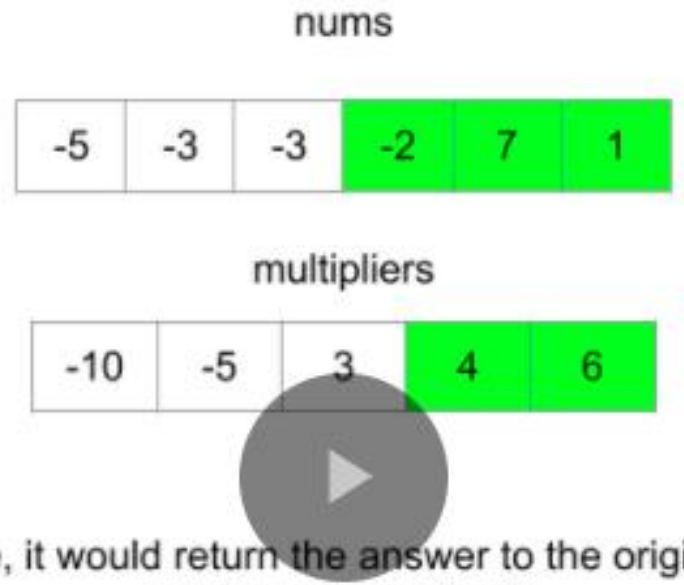
dp(i, l) returns the max possible score if we have already done *i* total operations and *l* left operations

nums

multipliers

If we called dp(2, 1) for example, it would return the answer to the original problem as if the highlighted section was the input.

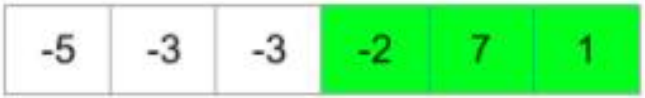
2 operations have been done, so 2 multipliers have been consumed. 1 multiplier was a left operation, so that means 1 had to have been a right, so 1 number is removed from each.



If we called dp(3, 3) for example, it would return the answer to the original problem as if the highlighted section was the input.

To answer the original problem, we return dp(0, 0) to consider the entire input.

nums



multipliers

If we called dp(3, 3) for example, it would return the answer to the original problem as if the highlighted section was the input.

To answer the original problem, we return dp(0, 0) to consider the entire input.

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

At each state, we have to perform an operation. As stated in the problem description, we need to decide whether to take from the left end ($\text{nums}[\text{left}]$) or the right end ($\text{nums}[\text{right}]$) of the current nums. Then we need to multiply the number we choose by $\text{multipliers}[\text{i}]$, add this value to our score, and finally remove the number we chose from nums. For implementation purposes, "removing" a number from nums means incrementing our state variables i and left so that they point to the next two left and right numbers.

Let $\text{mult} = \text{multipliers}[\text{i}]$ and $\text{right} = \text{nums.length} - 1 - (\text{i} - \text{left})$. The only decision we have to make is whether to take from the left or right of nums.

- If we choose left, we gain $\text{mult} \cdot \text{nums}[\text{left}]$ points from this operation. Then, the next operation will occur at $(\text{i} + 1, \text{left} + 1)$. i gets incremented at every operation because it represents how many operations we have done, and left gets incremented because it represents how many left operations we have done. Therefore, our total score is $\text{mult} \cdot \text{nums}[\text{left}] + \text{dp}(\text{i} + 1, \text{left} + 1)$.
- If we choose right, we gain $\text{mult} \cdot \text{nums}[\text{right}]$ points from this operation. Then, the next operation will occur at $(\text{i} + 1, \text{left})$. Therefore, our total score is $\text{mult} \cdot \text{nums}[\text{right}] + \text{dp}(\text{i} + 1, \text{left})$.

Since we want to maximize our score, we should choose the side that gives more points. This gives us our recurrence relation:

$$\text{dp}(\text{i}, \text{left}) = \max(\text{mult} \cdot \text{nums}[\text{left}] + \text{dp}(\text{i} + 1, \text{left} + 1),\ \text{mult} \cdot \text{nums}[\text{right}] + \text{dp}(\text{i} + 1, \text{left}))$$

Where $\text{mult} \cdot \text{nums}[\text{left}] + \text{dp}(\text{i} + 1, \text{left} + 1)$ represents the points we gain by taking from the left end of nums plus the maximum points we can get from the remaining nums array and $\text{mult} \cdot \text{nums}[\text{right}] + \text{dp}(\text{i} + 1, \text{left})$ represents the points we gain by taking from the right end of nums plus the maximum points we can get from the remaining nums array.

3. **Base cases**

The problem statement says that we need to perform m operations. When i equals m, that means we have no operations left. Therefore, we should return 0.

Top-down Implementation

Let's put the 3 parts of the framework together for a solution to the problem.

Protip: for Python, the functools module provides super handy tools that automatically memoize a function for us. We're going to use the `@lru_cache` decorator in the Python implementation.

> If you find yourself needing to memoize a function in an interview and you're using Python, check with your interviewer if using modules like functools is OK.

This particular problem happens to have very tight time limits. For Java, instead of using a hashmap for the memoization, we will use a 2D array. For Python, we're going to limit our cache size to 2000.

This particular problem happens to have very tight time limits. For Java, instead of using a hashmap for the memoization, we will use a 2D array. For Python, we're going to limit our cache size to 2000.

Java | Python3

Copy

```java
class Solution {
    private int[][] memo;
    private int[] nums, multipliers;
    private int n, m;

    private int dp(int i, int left) {
        if (i == m) {
            return 0; // Base case
        }

        int mult = multipliers[i];
        int right = n - 1 - (i - left);

        if (memo[i][left] == 0) {
            // Recurrence relation
            memo[i][left] = Math.max(mult * nums[left] + dp(i + 1, left + 1),
                                     mult * nums[right] + dp(i + 1, left));
        }

        return memo[i][left];
    }

```

```
    public int maximumScore(int[] nums, int[] multipliers) {
        n = nums.length;
        m = multipliers.length;
        this.nums = nums;
        this.multipliers = multipliers;
        this.memo = new int[m][m];
        return dp(0, 0);
    }
}
```

Bottom-up Implementation

In the bottom-up implementation, the array works the same way as the function from top-down. $\mathrm{dp}[\mathrm{i}][\mathrm{left}]$ represents the max score possible if i operations have been performed and left left operations have been performed.

Earlier in the explore card, we learned that while bottom-up is typically faster than top-down, it is often harder to implement. This is because the order in which we iterate needs to be precise. You'll see in the implementations below that we use the same math to calculate right, and the same recurrence relation but we need to iterate backwards starting from m (because the base case happens when i equals m). We also need to initialize dp with one extra row so that we don't go out of bounds in the first iteration of the outer loop.

Java | Python3 — Copy

```java
class Solution {
    public int maximumScore(int[] nums, int[] multipliers) {
        int n = nums.length;
        int m = multipliers.length;
        int[][] dp = new int[m + 1][m + 1];

        for (int i = m - 1; i >= 0; i--) {
            for (int left = i; left >= 0; left--) {
                int mult = multipliers[i];
                int right = n - 1 - (i - left);
                dp[i][left] = Math.max(mult * nums[left] + dp[i + 1][left + 1],
                                       mult * nums[right] + dp[i + 1][left]);
            }
        }

        return dp[0][0];
    }
}
```

The time and space complexity of both implementations is $O(m^2)$ where m is the length of $\mathrm{multipliers}$. We will talk about more in depth about time and space complexity at the end of this chapter.

1143. Longest Common Subsequence

▼ Click here to show hint regarding state variables and dp

Let $\mathrm{dp}(\mathrm{i}, \mathrm{j})$ represent the longest common subsequence between the string $\mathrm{text1}$ up to index i and $\mathrm{text2}$ up to index j.

▼ Click here to show hint regarding the recurrence relation

If $\mathrm{text1}[\mathrm{i}] == \mathrm{text2}[\mathrm{j}]$, then we should use this character, giving us $1 + \mathrm{dp}(\mathrm{i} - 1, \mathrm{j} - 1)$. Otherwise, we can either move one character back from $\mathrm{text1}$, or 1 character back from $\mathrm{text2}$. Try both.

▼ Click here to show hint regarding base cases

If i or j becomes less than 0, then we're out of bounds and should $\mathrm{return}\ 0$.

221. Maximal Square

▼ Click here to show hint regarding state variables and dp

Let $\mathrm{dp}[\mathrm{row}][\mathrm{col}]$ represent the largest possible square whose bottom right corner is on $\mathrm{matrix}[\mathrm{row}][\mathrm{col}]$.

▼ Click here to show hint regarding the recurrence relation

Any square with a 0 cannot have a square on it, and should be ignored. Otherwise, let's say you had a 3x3 square. Look at the bottom right corner of this 3x3 square. What do the squares above, to the left, and to the up-left of this square have in common? All 3 of those squares are the bottom-right square of a square that is (at least) 2x2.

▼ Click here to show hint regarding base cases

Just make sure to stay in bounds.

Maximum Score from Performing Multiplication Operations

You are given two integer arrays `nums` and `multipliers` of size `n` and `m` respectively, where `n >= m`. The arrays are **1-indexed**.

You begin with a score of `0`. You want to perform **exactly** `m` operations. On the `i^th` operation **(1-indexed)**, you will:

- Choose one integer `x` from **either the start or the end** of the array `nums`.
- Add `multipliers[i] * x` to your score.
- Remove `x` from the array `nums`.

Return *the **maximum** score after performing* `m` *operations.*

Example 1:

```
Input: nums = [1,2,3], multipliers = [3,2,1]
Output: 14
Explanation: An optimal solution is as follows:
- Choose from the end, [1,2,3], adding 3 * 3 = 9 to the score.
- Choose from the end, [1,2], adding 2 * 2 = 4 to the score.
- Choose from the end, [1], adding 1 * 1 = 1 to the score.
The total score is 9 + 4 + 1 = 14.
```

Example 2:

```
Input: nums = [-5,-3,-3,-2,7,1], multipliers = [-10,-5,3,4,6]
Output: 102
Explanation: An optimal solution is as follows:
- Choose from the start, [-5,-3,-3,-2,7,1], adding -5 * -10 = 50 to the score.
- Choose from the start, [-3,-3,-2,7,1], adding -3 * -5 = 15 to the score.
- Choose from the start, [-3,-2,7,1], adding -3 * 3 = -9 to the score.
- Choose from the end, [-2,7,1], adding 1 * 4 = 4 to the score.
- Choose from the end, [-2,7], adding 7 * 6 = 42 to the score.
The total score is 50 + 15 - 9 + 4 + 42 = 102.
```

Constraints:

- `n == nums.length`
- `m == multipliers.length`
- `1 <= m <= 10^3`
- `m <= n <= 10^5`
- `-1000 <= nums[i], multipliers[i] <= 1000`

Hide Hint #1

At first glance, the solution seems to be greedy, but if you try to greedily take the largest value from the beginning or the end, this will not be optimal.

Hide Hint #2

You should try all scenarios but this will be costy.

Hide Hint #3

Memoizing the pre-visited states while trying all the possible scenarios will reduce the complexity, and hence dp is a perfect choice here.

A Time and Space Complexity

Report Issue

Finding the time and space complexity of a dynamic programming algorithm may sound like a daunting task. However, this task is usually not as difficult as it sounds. Furthermore, justifying the time and space complexity in an explanation is relatively simple as well. One of the main points with DP is that we never repeat calculations, whether by tabulation or memoization, we only compute a state once. Because of this, the time complexity of a DP algorithm is directly tied to the number of possible states.

If computing each state requires F time, and there are n possible states, then the time complexity of a DP algorithm is $O(n \cdot F)$. With all the problems we have looked at so far, computing a state has just been using a recurrence relation equation, which is $O(1)$. Therefore, the time complexity has just been equal to the number of states. To find the number of states, look at each of your state variables, compute the number of values each one can represent, and then multiply all these numbers together.

Let's say we had 3 state variables: i, k, and $\mathrm{holding}$ for some made up problem. i is an integer used to keep track of an index for an input array nums, k is an integer given in the input which represents the maximum actions we can do, and $\mathrm{holding}$ is a boolean variable. What will the time complexity be for a DP algorithm that solves this problem? Let $\mathrm{n} = \mathrm{nums.length}$ and K be the maximum actions possible given in the input. i can be from 0 to $\mathrm{nums.length}$, k can be from 0 to K, and $\mathrm{holding}$ }can be true or false. Therefore, there are $\mathrm{n} \cdot \mathrm{K} \cdot 2$ states. If computing each state is $O(1)$, then the time complexity will be $O(n \cdot K \cdot 2) = O(n \cdot K)$.

Whenever we compute a state, we also store it so that we can refer to it in the future. In bottom-up, we tabulate the results, and in top-down, states are memoized. Since we store states, the space complexity is equal to the number of states. That means that in problems where calculating a state is $O(1)$, the time and space complexity are the same. In many DP problems, there are optimizations that can improve both complexities - we'll talk about this later.

Chapter 2 quiz

Report Issue

Multiple Choice Question The three main components of the framework are:

- [ ] **The choice between bottom-up and top-down**
- [x] **The base cases**
- [x] **The recurrence relation**
- [ ] **The number of dimensions**
- [ ] **The time complexity**
- [x] **A data structure and a way of visiting each DP state**

Redo Submit

Correct.

Commit these to memory! Understanding the components of the framework allows us to approach dynamic programming problems in a step by step way.

Multiple Choice Question Which of the following require state variables?

- [x] **Remaining number of moves allowed**
- [ ] **Original length of the input**
- [x] **Current index along the input**
- [x] **Number of keys currently being held**
- [ ] **The original number of moves allowed**

Redo Submit

Correct.

Constants should never be state variables.

Multiple Choice Question

Typically, implementing a top-down algorithm is easier than the equivalent bottom-up algorithm.

- (●) **True**
- ( ) **False**

Redo ✓ Submit

Correct.

Often it feels more intuitive to write the top-down implementation of a dynamic programming approach first. This is partly because in the top-down implementation, when the current state is not a base case, we can simply look at the recurrence relation to make the appropriate recursive calls. These recursive calls decide the order in which we visit each state. However, when writing the bottom-up implementation, we must carefully choose how we iterate over the state variables. We must ensure that we never visit a state before we have the results for all subproblems used to solve the current state. Fortunately, there is a way to systematically convert a top-down algorithm into its bottom-up version. So we can always write the more intuitive approach first (top-down) and then convert it to bottom-up if desired.

Multiple Choice Question

Generally, the time and space complexity of dynamic programming algorithms is directly related to:

- ○ **Whether the algorithm is top-down or bottom-up**
- ◉ **The number of possible states**
- ○ **Whether an array or hashtable is used for caching results**

↺ Redo ✓ Submit

Correct.

In many cases, the space complexity is the number of states, and the time complexity is the number of states multiplied by the cost of calculating a state.

A Iteration in the recurrence relation

Report Issue

In all the problems we have looked at so far, the recurrence relation is a static equation - it never changes. Recall Min Cost Climbing Stairs. The recurrence relation was:

$$\mathrm{dp(i)} = \mathrm{min(dp(i - 1) + cost[i - 1], dp(i - 2) + cost[i - 2])}$$

because we are only allowed to climb 1 or 2 steps at a time. What if the question was rephrased so that we could take up to k steps at a time? The recurrence relation would become dynamic - it would be:

$$\mathrm{dp(i)} = \mathrm{min(dp(j) + cost[j])} \text{ for all } \mathrm{(i - k)} \leq \mathrm{j} < \mathrm{i}$$

We would need iteration in our recurrence relation.

This is a common pattern in DP problems, and in this chapter, we're going to take a look at some problems using the framework where this pattern is applicable. While iteration usually increases the difficulty of a DP problem, particularly with bottom-up implementations, the idea isn't too complicated. Instead of choosing from a static number of options, we usually add a for-loop to iterate through a dynamic number of options and choose the best one.

A Example 1335. Minimum Difficulty of a Job Schedule

Report Issue

> We'll start with a top-down approach.

In this article, we'll be using the framework to solve 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 $\mathrm{dp}(\mathrm{i}, \mathrm{day})$ that returns the minimum difficulty of a job schedule which starts on the i^{th} job and day, day. To solve the original problem, we will just return $\mathrm{dp}(0, 1)$, since we start on the first day with no jobs done yet.

jobDifficulty

6	5	4	3	2	1

d = 2

This is the original input.

dp(i, day) returns the minimum possible difficulty of a job schedule created starting from job ***i*** on day ***day***.

Returning dp(0, 1) would solve the original problem, because we start with the first job on the first day.

jobDifficulty

6	5	4	3	2	1

d = 2

If we called dp(3, 2) for example, it would return the minimum difficulty of a job schedule that could be created if the highlighted section was the input, and if it was the 2nd day.

Since *day* is equal to d, it is the last day and we have to finish the rest of the jobs. Thus, dp(3, 2) will return 3 because the most difficult day remaining has difficulty 3.

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 $\mathrm{d} - \mathrm{day}$ 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, $\mathrm{jobDifficulty.length}$, that means from any given state $(\mathrm{i}, \mathrm{day})$, we are allowed to do the jobs from index i up to but not including index $\mathrm{n} - (\mathrm{d} - \mathrm{day})$.

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 $\mathrm{hardest}$ to keep track of the difficulty of the hardest job done today. If we choose to do jobs up to the j^{th} job (inclusive), where $\mathrm{i} \leq \mathrm{j} < \mathrm{n} - (\mathrm{d} - \mathrm{day})$ (as derived above), then that means on the next day, we start with the $(j+1)^{th}$ job. Therefore, our total difficulty is $\mathrm{hardest} + \mathrm{dp}(\mathrm{j}+1, \mathrm{day}+1)$. This gives us our scariest recurrence relation so far:

$\mathrm{dp}(\mathrm{i}, \mathrm{day}) = \min(\mathrm{hardest} + \mathrm{dp}(\mathrm{j}+1, \mathrm{day}+1))$ for all $\mathrm{i} \leq \mathrm{j} < \mathrm{n} - (\mathrm{d} - \mathrm{day})$, where
$\mathrm{hardest} = \max(\mathrm{jobDifficulty}[\mathrm{k}])$ for all $\mathrm{i} \leq \mathrm{k} \leq \mathrm{j}$.

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). $\mathrm{hardest}$ is the hardest job we do on the current day, which means it is also the difficulty of the current day. We add $\mathrm{hardest}$ 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.

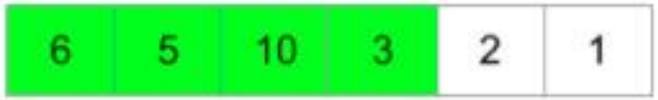
jobDifficulty

6	5	10	3	2	1

d = 3

Using this example, let's analyze the first state, where **i = 0**, **day = 1**

jobDifficulty



d = 3

How many jobs can we do on the first day?

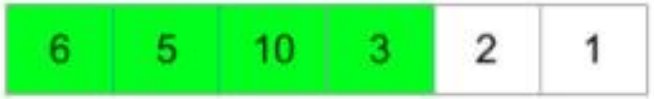
There are 3 days total and it is the first day, which means there are 2 **(d - *day*)** days after this one.

The problem says we need to do at least one job every day, which means we need to leave at least 2 jobs.

We can choose to do the highlighted jobs. We have to stop **before** index **n - (d - *day*) = 6 - (3 - 1) = 4**.

We should try all possibilities - do only the first job, do the first two jobs, do the first three jobs, and do all four jobs.

jobDifficulty



d = 3

So, on the first day, we can choose to do between 1 and 4 jobs. We need to do the jobs in order, so we should iterate from left to right.

We need to keep track of the hardest job we have seen so far on the current day, because that is what defines the difficulty of the day. If we did only 1 or 2 jobs, then **hardest = 6**. If we did 3 or 4 jobs, then **hardest = 10**.

Whatever our choice is, to find the difficulty of the job schedule, it is the current day's difficulty + the next state, which would be the next day, starting with the next job.

If we did 2 jobs on the first day, the schedule's difficulty would be **6 + dp(2, 2)**, because the next job is the one at index 2, and the next day is day 2.

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 ($\mathrm{day} == \mathrm{d}$), we need to finish up all the remaining jobs on this day, and the total difficulty will just be the largest number in $\mathrm{jobDifficulty}$ on or after index i.

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

We can precompute an array $\mathrm{hardestJobRemaining}$ where $\mathrm{hardestJobRemaining[i]}$ 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 ($\mathrm{n} < \mathrm{d}$), 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 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.

Java | Python3

Copy

```java
class Solution {
    private int n, d;
    private int[][] memo;
    private int[] jobDifficulty;
    private int[] hardestJobRemaining;

    private int dp(int i, int day) {
        // Base case, it's the last day so we need to finish all the jobs
        if (day == d) {
            return hardestJobRemaining[i];
        }

        if (memo[i][day] == -1) {
            int best = Integer.MAX_VALUE;
            int hardest = 0;
            // Iterate through the options and choose the best
            for (int j = i; j < n - (d - day); j++) {
                hardest = Math.max(hardest, jobDifficulty[j]);
                // Recurrence relation
                best = Math.min(best, hardest + dp(j + 1, day + 1));
            }
            memo[i][day] = best;
        }

        return memo[i][day];
    }
```

Java | Python3

Copy

```java
    public int minDifficulty(int[] jobDifficulty, int d) {
        n = jobDifficulty.length;
        // If we cannot schedule at least one job per day,
        // it is impossible to create a schedule
        if (n < d) {
            return -1;
        }

        hardestJobRemaining = new int[n];
        int hardestJob = 0;
        for (int i = n - 1; i >= 0; i--) {
            hardestJob = Math.max(hardestJob, jobDifficulty[i]);
            hardestJobRemaining[i] = hardestJob;
        }

        // Initialize memo array with value of -1.
        memo = new int[n][d + 1];
        for (int i = 0; i < n; i++) {
            Arrays.fill(memo[i], -1);
        }

        this.d = d;
        this.jobDifficulty = jobDifficulty;
        return dp(0, 1);
    }
}
```

Bottom-up Implementation

With bottom-up, we now use a 2D array where $\mathrm{dp}[\mathrm{i}][\mathrm{day}]$ 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.

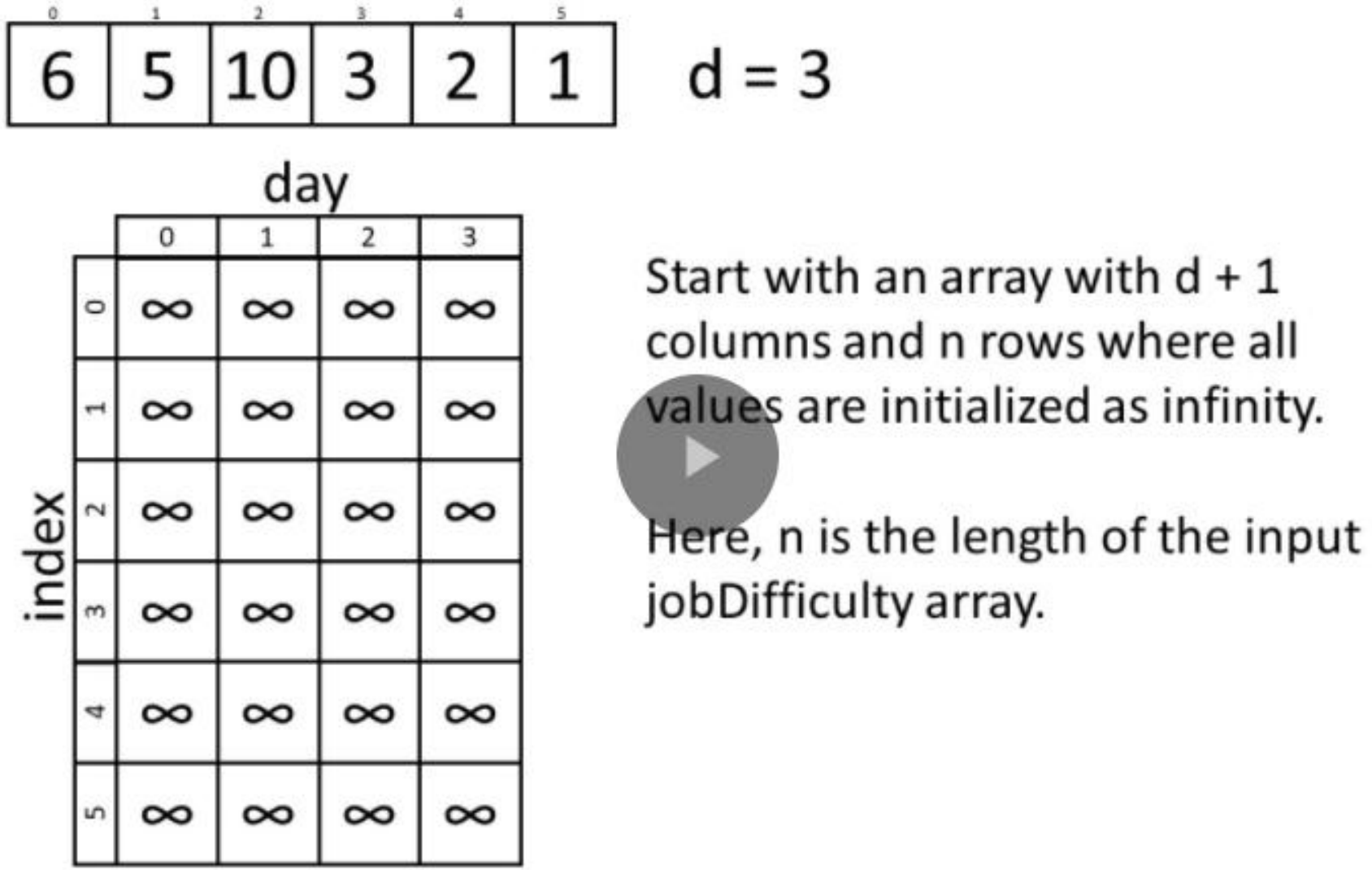
The for-loops should start iterating from the base cases, and there should be one for-loop for each state variable. Remember that one of our base cases is that on the final day, we need to complete all jobs. Therefore, our for-loop iterating over day should iterate from the final day to the first day. 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.


Java | Python3

```java
class Solution {
    public int minDifficulty(int[] jobDifficulty, int d) {
        int n = jobDifficulty.length;
        // If we cannot schedule at least one job per day,
        // it is impossible to create a schedule
        if (n < d) {
            return -1;
        }

        int dp[][] = new int[n][d + 1];
        for (int[] row: dp) {
            Arrays.fill(row, Integer.MAX_VALUE);
        }

        // Set base cases
        dp[n - 1][d] = jobDifficulty[n - 1];

        // On the last day, we must schedule all remaining jobs, so dp[i][d]
        // is the maximum difficulty job remaining
        for (int i = n - 2; i >= 0; i--) {
            dp[i][d] = Math.max(dp[i + 1][d], jobDifficulty[i]);
        }
```

```
        for (int day = d - 1; day > 0; day--) {
            for (int i = day - 1; i < n - (d - day); i++) {
                int hardest = 0;
                // Iterate through the options and choose the best
                for (int j = i; j < n - (d - day); j++) {
                    hardest = Math.max(hardest, jobDifficulty[j]);
                    // Recurrence relation
                    dp[i][day] = Math.min(dp[i][day], hardest + dp[j + 1][day + 1]);
                }
            }
        }

        return dp[0][1];
    }
}
```

Here's an animation showing the algorithm in action:



0	1	2	3	4	5
6	5	10	3	2	1

d = 3

index \ day	0	1	2	3
0	∞	∞	∞	∞
1	∞	∞	∞	∞
2	∞	∞	∞	∞
3	∞	∞	∞	∞
4	∞	∞	∞	∞
5	∞	∞	∞	∞

On the last day (day = d) we must schedule all of the remaining jobs. So our base case is dp[index][d] equals the highest difficulty job between index and the end of the jobDifficulty array.

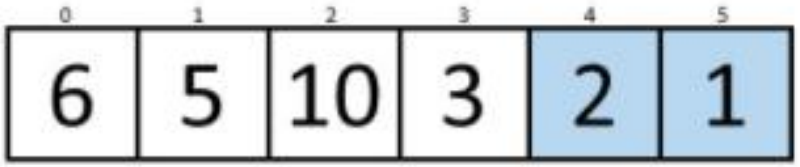
0	1	2	3	4	5
6	5	10	3	2	1

d = 3

day

index	0	1	2	3
0	∞	∞	∞	∞
1	∞	∞	∞	∞
2	∞	∞	∞	∞
3	∞	∞	∞	∞
4	∞	∞	∞	∞
5	∞	∞	∞	1

On the last day (day = d) we must schedule all of the remaining jobs. So our base case is dp[index][d] equals the highest difficulty job between index and the end of the jobDifficulty array.

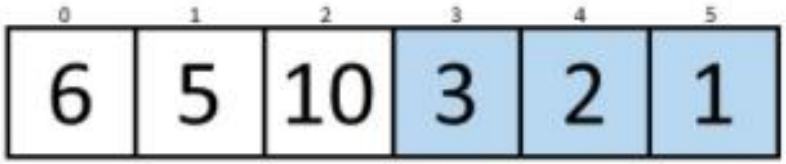


d = 3

day

index	0	1	2	3
0	∞	∞	∞	∞
1	∞	∞	∞	∞
2	∞	∞	∞	∞
3	∞	∞	∞	∞
4	∞	∞	∞	2
5	∞	∞	∞	1

On the last day (day = d) we must schedule all of the remaining jobs. So our base case is dp[index][d] equals the highest difficulty job between index and the end of the jobDifficulty array.

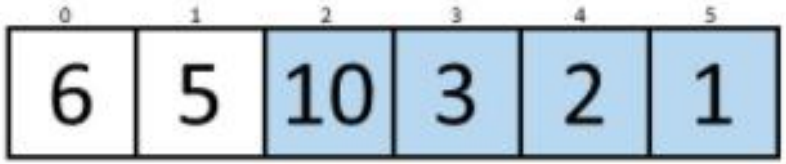


d = 3

day

index	0	1	2	3
0	∞	∞	∞	∞
1	∞	∞	∞	∞
2	∞	∞	∞	∞
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

On the last day (day = d) we must schedule all of the remaining jobs. So our base case is dp[index][d] equals the highest difficulty job between index and the end of the jobDifficulty array.



d = 3

day

index

	0	1	2	3
0	∞	∞	∞	∞
1	∞	∞	∞	∞
2	∞	∞	∞	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

On the last day (day = d) we must schedule all of the remaining jobs. So our base case is dp[index][d] equals the highest difficulty job between index and the end of the jobDifficulty array.

0	1	2	3	4	5
6	5	10	3	2	1

d = 3

index \ day	0	1	2	3
0	∞	∞	∞	∞
1	∞	∞	∞	10
2	∞	∞	∞	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

On the last day (day = d) we must schedule all of the remaining jobs. So our base case is dp[index][d] equals the highest difficulty job between index and the end of the jobDifficulty array.

0	1	2	3	4	5
6	5	10	3	2	1

d = 3

day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	∞	10
2	∞	∞	∞	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

On the last day (day = d) we must schedule all of the remaining jobs. So our base case is dp[index][d] equals the highest difficulty job between index and the end of the jobDifficulty array.

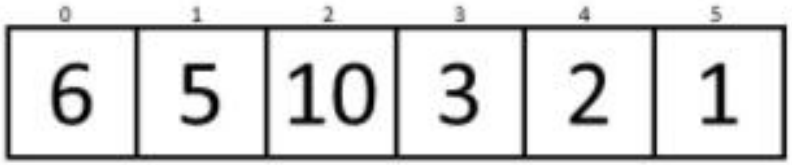
0	1	2	3	4	5
6	5	10	3	2	1

d = 3

day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	∞	10
2	∞	∞	∞	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

Next we will iterate day from our base case (day = d) towards our original query (index = 0, day = 1).



d = 3

day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	∞	10
2	∞	∞	∞	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

We can skip dp[0][2] because the first day is when day = 1 and we must have done at least one job (the job at index 0) on day 1, so it would be impossible for us to start day 2's job schedule with the job at index 0.

(For the same reason, it is not necessary to calculate dp[0][3] or dp[1][3])

0	1	2	3	4	5
6	5	10	3	2	1

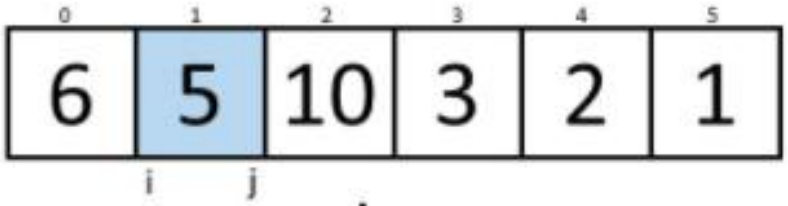
d = 3

day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	∞	10
2	∞	∞	∞	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

For the minimum difficulty job schedule that starts at the job at index 1 on day 2, we can use the recurrence relation to calculate dp[1][2].

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

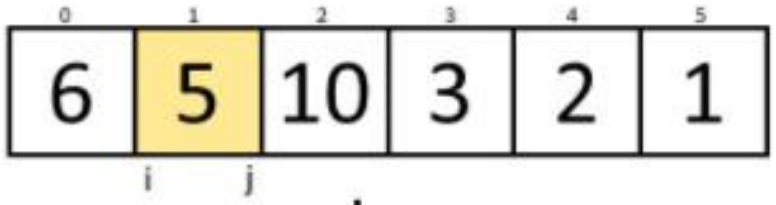
day

index

	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	∞	10
2	∞	∞	∞	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

In other words, i is the first job scheduled on the current day and j is the last job scheduled on the current day, let's mark these in blue.

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

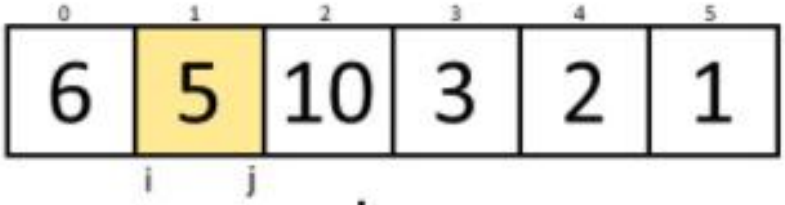
Hardest Job Scheduled Today

day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	∞	10
2	∞	∞	∞	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

hardest is the most difficult job scheduled on the current day, we will mark the hardest job in orange.

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

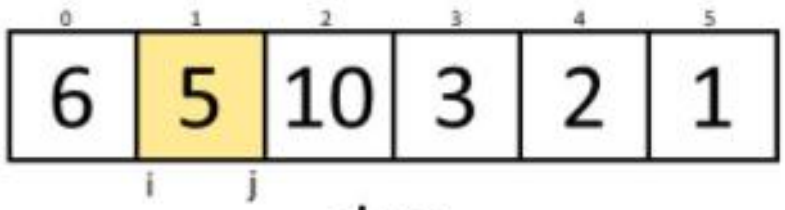
day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	∞	10
2	∞	∞	∞	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

The minimum difficulty job schedule starting at the job after j on the next day, which is dp(j + 1, day + 1), will be marked in purple.

$$dp(i, day) = \min(hardest + dp(j + 1, day + 1))$$

For all $i <= j < n - (d - day)$



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

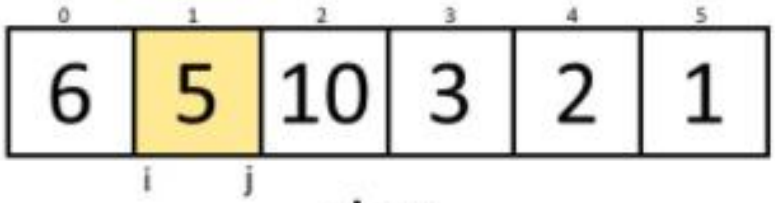
Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	∞	10
2	∞	∞	∞	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

Finally, we will simply follow the recurrence relation and update dp[index][day] as the minimum sum of hardest and dp[j + 1][day + 1] for all j in the range i to n – (d – day).

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

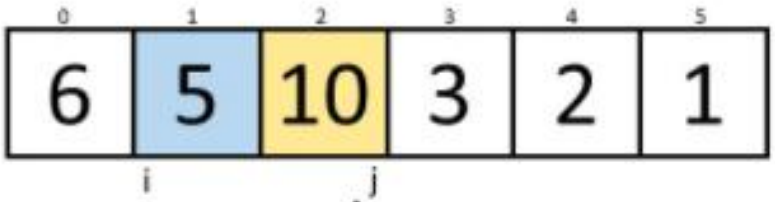
Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	15	10
2	∞	∞	∞	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

dp[1][2] = min(∞, 5 + 10)

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

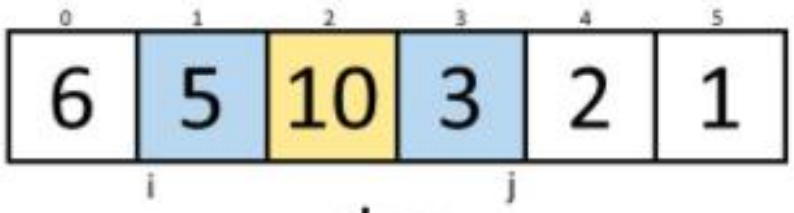
Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	13	10
2	∞	∞	∞	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

dp[1][2] = min(15, 10 + 3)

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

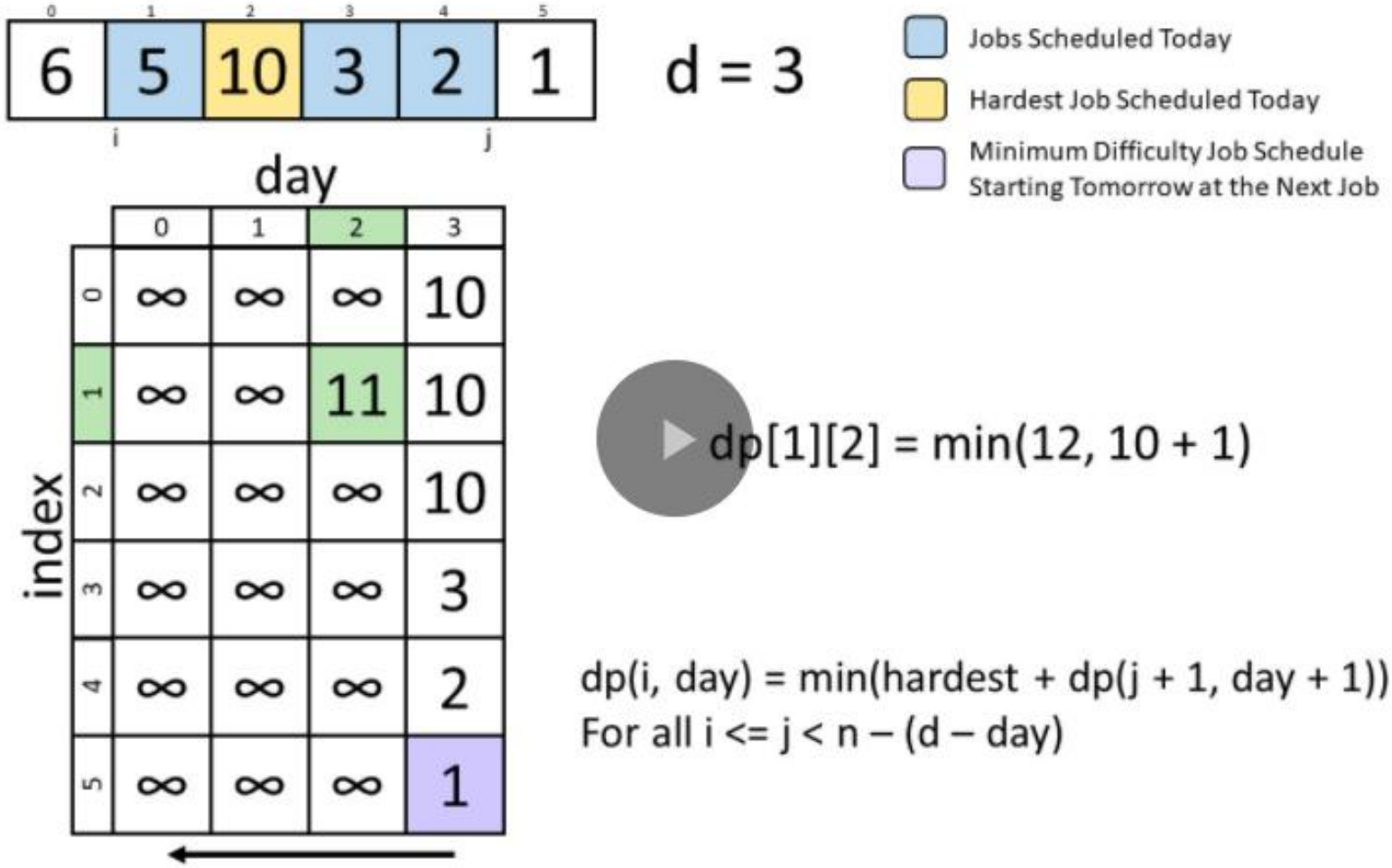
Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	12	10
2	∞	∞	∞	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

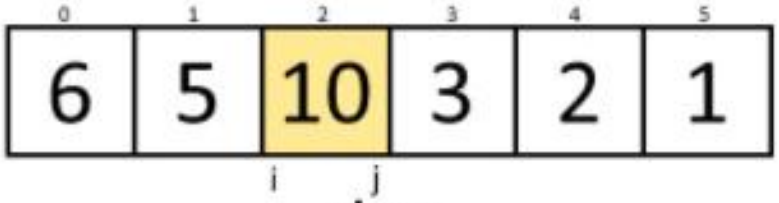
dp[1][2] = min(13, 10 + 2)

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



19 / 30

The time and space complexity of these algorithms can be quite tricky, and as in this example, there are



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

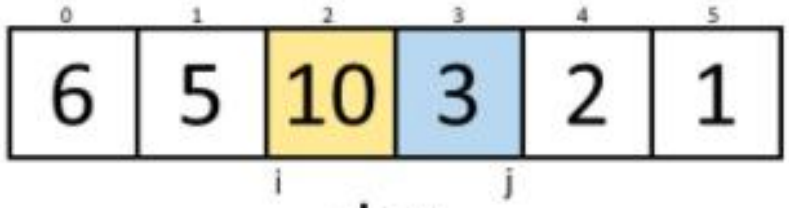
Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	11	10
2	∞	∞	13	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

dp[2][2] = min(∞, 10 + 3)

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

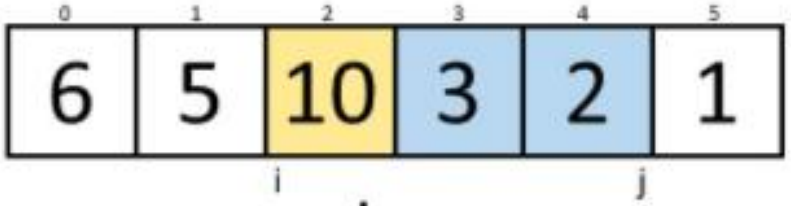
Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	11	10
2	∞	∞	12	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

dp[2][2] = min(13, 10 + 2)

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

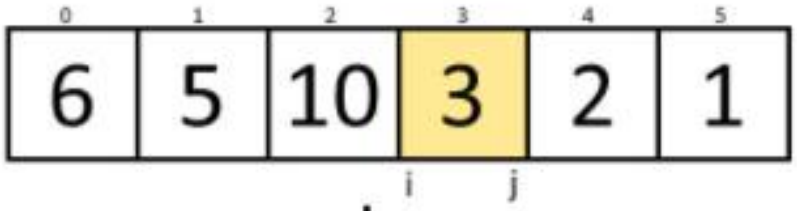
Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	11	10
2	∞	∞	11	10
3	∞	∞	∞	3
4	∞	∞	∞	2
5	∞	∞	∞	1

dp[2][2] = min(12, 10 + 1)

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

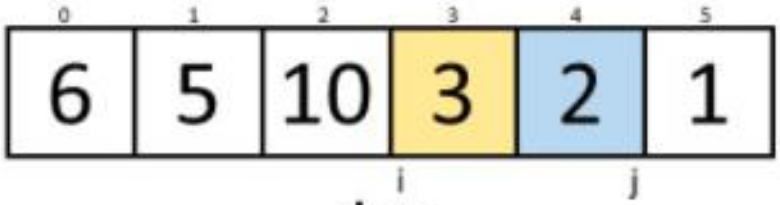
Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	11	10
2	∞	∞	11	10
3	∞	∞	5	3
4	∞	∞	∞	2
5	∞	∞	∞	1

dp[3][2] = min(∞, 3 + 2)

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

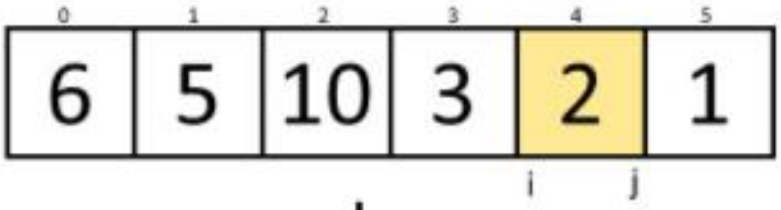
day

index

	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	11	10
2	∞	∞	11	10
3	∞	∞	4	3
4	∞	∞	∞	2
5	∞	∞	∞	1

dp[3][2] = min(5, 3 + 1)

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

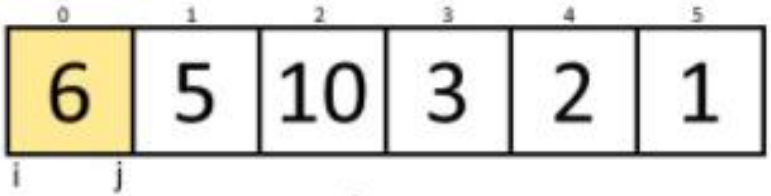
Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

day

index	0	1	2	3
0	∞	∞	∞	10
1	∞	∞	11	10
2	∞	∞	11	10
3	∞	∞	4	3
4	∞	∞	3	2
5	∞	∞	∞	1

dp[3][2] = min(∞, 2 + 1)

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

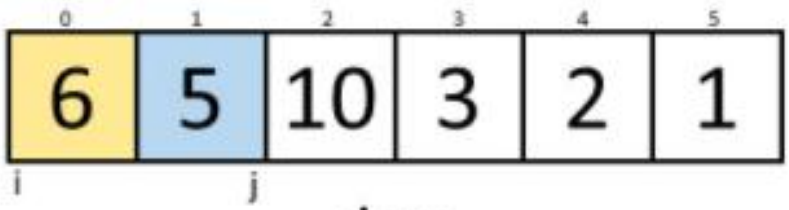
Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

day

index	0	1	2	3
0	∞	17	∞	10
1	∞	∞	11	10
2	∞	∞	11	10
3	∞	∞	4	3
4	∞	∞	3	2
5	∞	∞	∞	1

dp[0][1] = min(∞, 6 + 11)

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

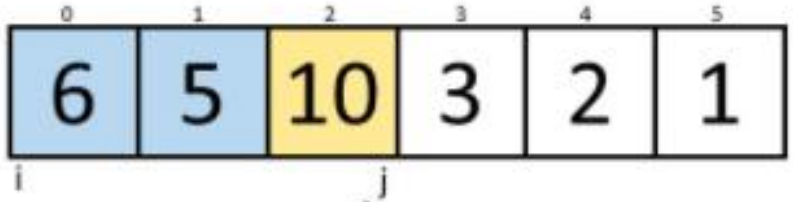
Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

day

index	0	1	2	3
0	∞	17	∞	10
1	∞	∞	11	10
2	∞	∞	11	10
3	∞	∞	4	3
4	∞	∞	3	2
5	∞	∞	∞	1

dp[0][1] = min(17, 6 + 11)

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

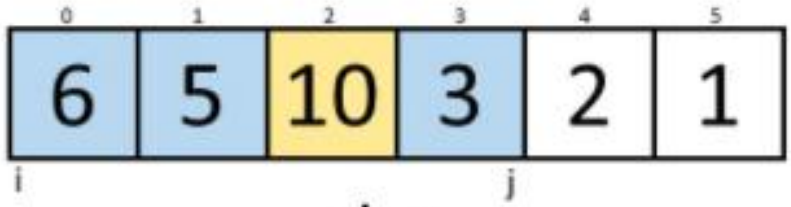
Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

day

index	0	1	2	3
0	∞	14	∞	10
1	∞	∞	11	10
2	∞	∞	11	10
3	∞	∞	4	3
4	∞	∞	3	2
5	∞	∞	∞	1

dp[0][1] = min(17, 10 + 4)

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)



d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

day

index	0	1	2	3
0	∞	13	∞	10
1	∞	∞	11	10
2	∞	∞	11	10
3	∞	∞	4	3
4	∞	∞	3	2
5	∞	∞	∞	1

dp[0][1] = min(14, 10 + 3)

dp(i, day) = min(hardest + dp(j + 1, day + 1))
For all i <= j < n – (d – day)

0	1	2	3	4	5
6	5	10	3	2	1

d = 3

Jobs Scheduled Today

Hardest Job Scheduled Today

Minimum Difficulty Job Schedule Starting Tomorrow at the Next Job

day

index	0	1	2	3
0	∞	13	∞	10
1	∞	∞	11	10
2	∞	∞	11	10
3	∞	∞	4	3
4	∞	∞	3	2
5	∞	∞	∞	1

At this point, we've obtained the result for our initial query:

dp[0][1] = 13

Meaning, that 13 is the minimum difficulty of a job schedule starting on day 1 with the job at index 0 and 3 days to schedule all jobs.

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 $n \cdot d$. This means the space complexity is $O(n \cdot d)$ 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 $n \cdot d$ 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 $d \cdot (n - d)$ states. This means the space complexity for top-down is $O(d \cdot (n - d))$. 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 $n \cdot d$ 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 $d \cdot (n - d)$ states. At each state, we go through a for-loop (with variable j) that iterates on average $\frac{n - d}{2}$ times. This means our time complexity for both algorithms is $O(d \cdot (n - d)^2)$.

To summarize:

Time complexity (both algorithms): $O(d \cdot (n-d)^2)$

Space complexity (top-down): $O((n-d) \cdot d)$

Space complexity (bottom-up): $O(n \cdot d)$

> 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.

The next problem is a classical one and popular in interviews. Try it out yourself, and come back here if you need some hints:

322. Coin Change

▼ Click here to show hint regarding state variables and dp

Let $\mathrm{dp}[\mathrm{i}]$ represent the fewest number of coins needed to make up i money.

▼ Click here to show hint regarding the recurrence relation.

For each coin available, we can get to i from i - coin. Make sure that $\mathrm{coin} \leq \mathrm{i}$.

▼ Click here to show hint regarding the base cases.

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

Minimum Difficulty of a Job Schedule

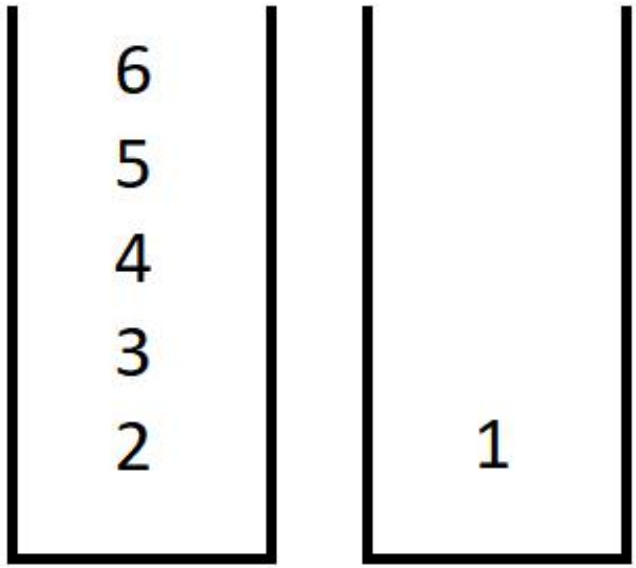
You want to schedule a list of jobs in `d` days. Jobs are dependent (i.e To work on the `i^th` job, you have to finish all the jobs `j` where `0 <= j < i`).

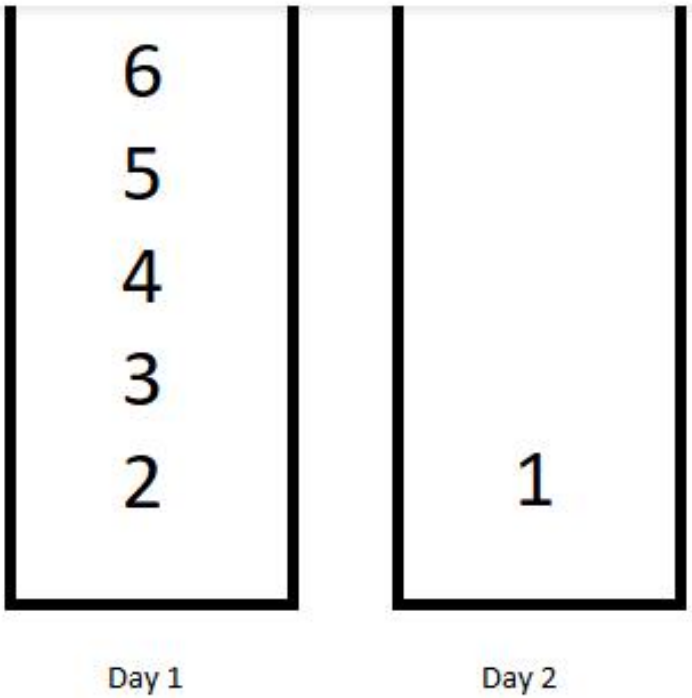
You have to finish **at least** one task every day. The difficulty of a job schedule is the sum of difficulties of each day of the `d` days. The difficulty of a day is the maximum difficulty of a job done on that day.

You are given an integer array `jobDifficulty` and an integer `d`. The difficulty of the `i^th` job is `jobDifficulty[i]`.

Return *the minimum difficulty of a job schedule*. If you cannot find a schedule for the jobs return `-1`.

Example 1:






```
Input: jobDifficulty = [6,5,4,3,2,1], d = 2
Output: 7
Explanation: First day you can finish the first 5 jobs, total difficulty = 6.
Second day you can finish the last job, total difficulty = 1.
The difficulty of the schedule = 6 + 1 = 7
```

Example 2:

```
Input: jobDifficulty = [9,9,9], d = 4
Output: -1
Explanation: If you finish a job per day you will still have a free day. you cannot find a schedule for
```

Hide Hint #1

Use DP. Try to cut the array into d non-empty sub-arrays. Try all possible cuts for the array.

Hide Hint #2

Use dp[i][j] where DP states are i the index of the last cut and j the number of remaining cuts. Complexity is O(n * n * d).

A Example 139. Word Break

Report Issue

In this article, we'll use the framework to solve Word Break. So far, in this card, this is the most unique and perhaps the most difficult problem to see that dynamic programming is a viable approach. This is because, unlike all of the previous problems, we will not be working with numbers at all. When a question asks, "is it possible to do..." it isn't necessarily a dead giveaway that it should be solved with DP. However, we can see that in this question, the order in which we choose words from wordDict is important, and a greedy strategy will not work.

> Recall back in the first chapter, we said that a good way to check if a problem should be solved with DP or greedy is to first assume that it can be solved greedily, then try to think of a counterexample.

Let's say that we had $s =$ "abcdef" and $\text{wordDict} = [$ "abcde", "ef", "abc", "a", "d"$]$. A greedy algorithm (picking the longest substring available) will not be able to determine that picking "abcde" here is the wrong decision. Likewise, a greedy algorithm (picking the shortest substring available) will not be able to determine that picking "a" first is the wrong decision.

With that being said, let's develop a DP algorithm using our framework:

> For this problem, we'll look at bottom-up first.

1. An **array** that answers the problem for a given state

Despite this problem being unlike the ones we have seen so far, we should still stick to the ideas of the framework. In the article where we learned about multi-dimensional dynamic programming, we talked about how an index variable, usually denoted i is typically used in DP problems where the input is an array or string. All the problems that we have looked at up to this point reflect this.

- With this in mind, let's use a state variable i, which keeps track of which index we are currently at in s.
- Do we need any other state variables? The other input is $wordDict$ - however, it says in the problem that we can reuse words from $wordDict$ as much as we want. Therefore, a state variable isn't necessary because $wordDict$ and what we can do with it never changes. If the problem was changed so that we can only use a word once, or say k times, then we would need extra state variables to know what words we are allowed to use at each state.

In all the past problems, we had a function dp return the answer to the original problem for some state. We should try to do the same thing here. The problem is asking, is it possible to create s by combining words in $wordDict$. So, let's have an array dp where $dp[i]$ represents if it is possible to build the string s up to index i from $wordDict$. To answer the original problem, we can return $dp[s.length - 1]$ after populating dp.

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

At each index i, what criteria determines if $dp[i]$ is true? First, a word from $wordDict$ needs to be able to **end** at index i. In terms of code, this means that there is some $word$ from $wordDict$ that matches the substring of s that starts at index $i - word.length + 1$ and ends at index i.

We can iterate through all states of i from 0 up to but not including $s.length$, and at each state, check all the words in $wordDict$ for this criteria. For each $word$ in $wordDict$, if s from index $i - word.length + 1$ to i is equal to $word$, that means $word$ **ends** at i. However, this is not the sole criteria.

Remember, we are forming s by adding words together. That means, if a $word$ meets the first criteria and we want to use it in a solution, we would add it on top of another string. We need to make sure that the string before it is also formable. If $word$ meets the first criteria, it starts at index $i - word.length + 1$. The index before that is $i - word.length$, and the second criteria is that s up to this index is also formable from $wordDict$. This gives us our recurrence relation:

`dp(i) = true` if `s.substring(i - word.length + 1, i + 1) == word` and `dp[i - word.length] == true` for any `word` in `wordDict`, otherwise `false`

dp

F	F	F	F	F	F	F	F
l	e	e	t	c	o	d	e

s = “leetcode”
wordDict = [“leet”, “code”]

The dp table holds boolean values T (true) and F (false). Under each element is the corresponding letter from the input string.

dp[i] indicates if it is possible to use words from wordDict to build the input string up to index i.

dp

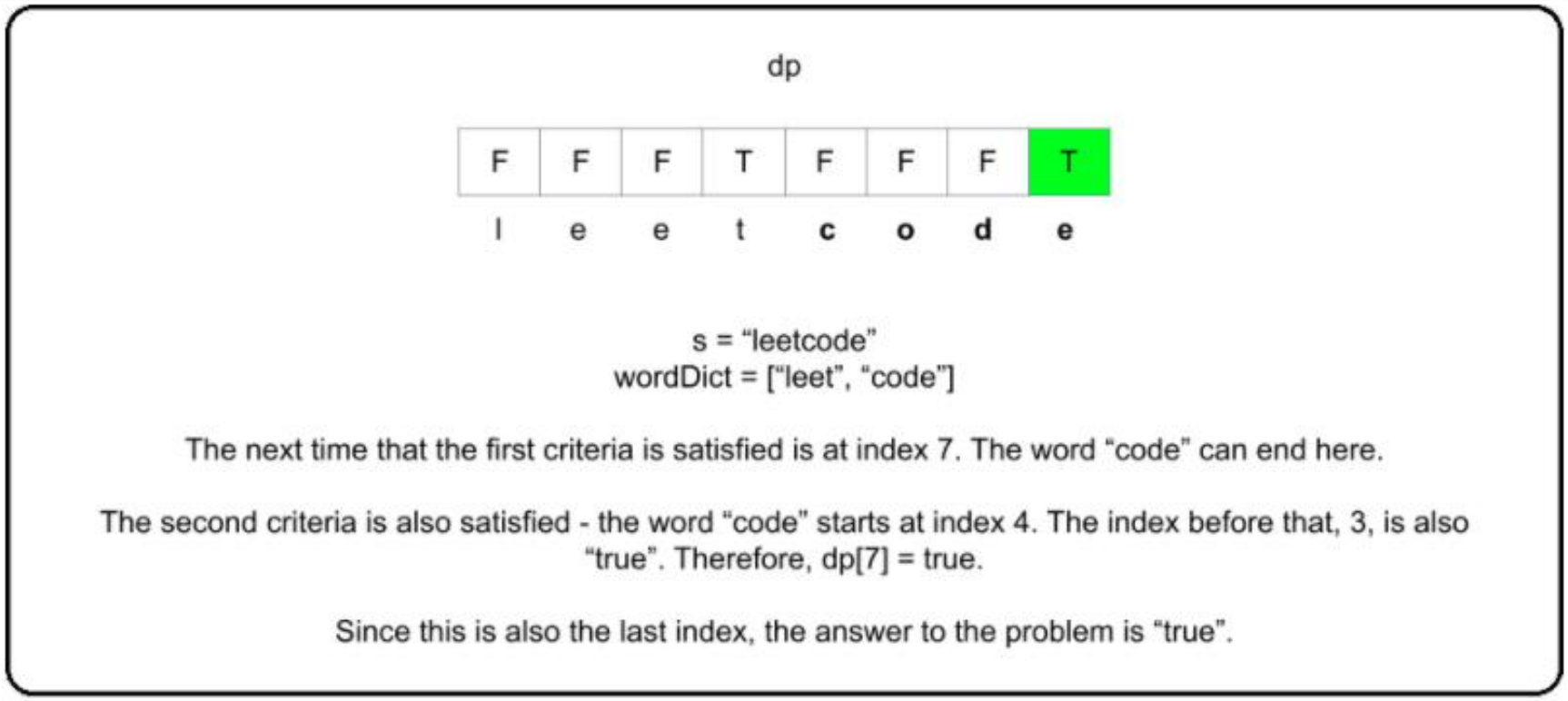
F	F	F	T	F	F	F	F
l	**e**	**e**	**t**	c	o	d	e

s = “leetcode”
wordDict = [“leet”, “code”]

The first criteria is that a word from wordDict can end at s[i]. The first occurrence of this is at index 3. The word “leet” can end here.

The second criteria is that the input string is formable up to the point before where the current string starts. Since this is the first word being used, the criteria is also met.

Both criterias are satisfied, dp[3] = true.



In summary, the criteria is:

1. A word from wordDict can **end** at the current index i.
2. If that word is to end at index i, then it starts at index i - word.length + 1. The index before that i - word.length should also be formable from wordDict.

3. **Base cases**

The base case for this problem is another simple one. The first word used from wordDict starts at index 0, which means we would need to check dp[-1] for the second criteria, which is out of bounds. To fix this, we say that the second criteria can also be satisfied by i == word.length - 1.

Bottom-up Implementation

Java | Python3

Copy

```java
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        boolean[] dp = new boolean[s.length()];
        for (int i = 0; i < s.length(); i++) {
            for (String word : wordDict) {
                // Make sure to stay in bounds while checking criteria
                if (i >= word.length() - 1 && (i == word.length() - 1 || dp[i - word.length()])) {
                    if (s.substring(i - word.length() + 1, i + 1).equals(word)) {
                        dp[i] = true;
                        break;
                    }
                }
            }
        }

        return dp[s.length() - 1];
    }
}
```

Top-down Implementation

In the top-down approach, we can check for the base case by returning $\texttt{true}$ if $i < 0$. In Java, we will memoize by using a -1 to indicate that the state is unvisited, 0 to indicate $\texttt{false}$, and 1 to indicate $\texttt{true}$.

Java | Python3

Copy

```java
class Solution {
    private String s;
    private List<String> wordDict;
    private int[] memo;

    private boolean dp(int i) {
        if (i < 0) return true;

        if (memo[i] == -1) {
            for (String word: wordDict) {
                if (i >= word.length() - 1 && dp(i - word.length())) {
                    if (s.substring(i - word.length() + 1, i + 1).equals(word)) {
                        memo[i] = 1;
                        break;
                    }
                }
            }
        }

        if (memo[i] == -1) {
            memo[i] = 0;
        }

        return memo[i] == 1;
    }

```

```
26
27      public boolean wordBreak(String s, List<String> wordDict) {
28          this.s = s;
29          this.wordDict = wordDict;
30          this.memo = new int[s.length()];
31          Arrays.fill(this.memo, -1);
32          return dp(s.length() - 1);
33      }
34  }
```

Let's say that $\mathrm{n} = \mathrm{s.length}$, $\mathrm{k} = \mathrm{wordDict.length}$, and L is the average length of the words in $\mathrm{wordDict}$. While the space complexity for this problem is the same as the number of states n, the time complexity is much worse. At each state i, we iterate through $\mathrm{wordDict}$ and splice s to a new string with average length L. This gives us a time complexity of $O(n \cdot k \cdot L)$.

Word Break

Solution

Given a string `s` and a dictionary of strings `wordDict`, return `true` if `s` can be segmented into a space-separated sequence of one or more dictionary words.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:

```
Input: s = "leetcode", wordDict = ["leet","code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".
```

Example 2:

```
Input: s = "applepenapple", wordDict = ["apple","pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
Note that you are allowed to reuse a dictionary word.
```

Example 3:

```
Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]
Output: false
```

```cpp
class Solution {
public:
    bool wordBreak(string s, vector<string>& D)
    {
        int n = s.size();
        vector<bool> dp(n+1, false);
        for(int j = 0; j<D.size(); j++)
        {
            int pos = s.find(D[j], 0);
            if(pos != string::npos && pos == 0){
                dp[D[j].size()-1] = true;
            }
        }

        for(int i=0; i<n; i++){
            if(dp[i]){
                for(int j = 0; j<D.size(); j++)
                {
                    int pos = s.find(D[j], i+1);
                    if(pos != string::npos && pos == i+1)
                    {
                        if(i+D[j].size() == n-1) return true;
                        dp[i+D[j].size()] = true;
                    }
                }
            }
        }
        return dp[n-1];
    }
};
```

Custom Testcase (Contribute !)

Run Code | Submit

Submission Result: Accepted

More Details >

Longest Increasing Subsequence

Solution

Given an integer array `nums`, return the length of the longest strictly increasing subsequence.

A **subsequence** is a sequence that can be derived from an array by deleting some or no elements without changing the order of the remaining elements. For example, `[3,6,2,7]` is a subsequence of the array `[0,3,1,6,2,2,7]`.

Example 1:

```
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.
```

Example 2:

```
Input: nums = [0,1,0,3,2,3]
Output: 4
```

Example 3:

```
Input: nums = [7,7,7,7,7,7,7]
Output: 1
```

C++

```cpp
class Solution {
public:
    int lengthOfLIS(vector<int>& a) {

        int n= a.size(); //size of the vector
        int dp[n]; //dp initilised. by the size of the vector a
        dp[0]=1; //array of length 1 is LIS trivially whose length is 1.
        int lis = dp[0]; //initialise variable lis as dp[0] i.e. 1
        for(int i=1;i<n;i++){ //for all the elements in the array from 1 to n-1 inclusive
            int x =0; //make a variable x and assign it to 0
            for(int j=0;j<i;j++)  //for all j from 0 to i-1 inclusive
                if(a[j] < a[i]) x= max(x,dp[j]); //whatever be the max of x and dp at index j
            dp[i]=x+1; //in the outer for loop, update dp[i] as x+1 this we have to do.
            lis=max(lis,dp[i]); //lis will keep track of curent max or dp[i] whatever max, return
        }
        return lis; //at the end return lis
    }
};
```

Custom Testcase (Contribute)

Shortcut: Command + '

Run Code Submit

Submission Result: Accepted More Details >

A State Transition by Inaction

Report Issue

This is a small pattern that occasionally shows up in DP problems. Here, "doing nothing" refers to two different states having the same value. We're calling it "doing nothing" because often the way we arrive at a new state with the same value as the previous state is by "doing nothing" (we'll look at some examples soon). Of course, a decision making process needs to coexist with this pattern, because if we just had all states having the same value, the problem wouldn't really make sense ($\mathrm{dp(i)} = \mathrm{dp(i - 1)}$?) It is just that if we are trying to maximize or minimize a score for example, sometimes the best option is to "do nothing", which leads to two states having the same value. The actual recurrence relation would look something like $\mathrm{dp(i, j)} = \mathrm{max(dp(i - 1, j), ...)}$.

Usually when we "do nothing", it is by moving to the next element in some input array (which we usually use i as a state variable for). As mentioned above, **this will be part of a decision making process due to some restriction in the problem**. For example, think back to House Robber: we could choose to rob or not rob each house we were at. Sometimes, not robbing the house is the best decision (because we aren't allowed to rob adjacent houses), then $\mathrm{dp(i)} = \mathrm{dp(i - 1)}$.

In the next article, we'll use the framework to solve a problem with this pattern.

A Example 188. Best Time to Buy and Sell Stock IV

Report Issue

> For this one, we're back to starting with top-down.

In this article, we'll be using the framework to solve Best Time to Buy and Sell Stock IV. This problem is rated as "hard" and may seem daunting at first, but with the framework, the logic behind solving this problem is very intuitive. We'll also make use of the pattern of "doing nothing". Like usual, let's use the framework to develop an algorithm:

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

What information do we need at each state/decision?

We need to know what day it is (so we can look up the current price of the stock), and we need to know how many transactions we have left. These two are directly related to the input.

The note in the problem description says that we cannot engage in multiple transactions at the same time. This means that at any moment, we are either holding one unit of stock or not holding any stock. We should have a state variable that indicates if we are currently holding stock. This variable is fine as a boolean, but for caching purposes, let's use an integer alternating between 0 and 1 (0 means not holding, 1 means holding).

To summarize, we have 3 state variables:

1. i, which represents we are on the i^{th} day. The current price of the stock is $\mathrm{prices[i]}$.
2. $\mathrm{transactionsRemaining}$, which represents how many transactions we have left. This number goes down by 1 whenever we sell a stock.
3. $\mathrm{holding}$, which is equal to 0 if we are not holding a stock, and 1 if we are holding a stock. If $\mathrm{holding}$ is 0, we have the option to buy a stock. Otherwise, we have the option to sell a stock.

The problem is asking for a maximum achievable profit. Therefore, let's have a function dp where $\mathrm{dp(i,\ transactionsRemaining,\ holding)}$ returns the maximum achievable profit starting from the i^{th} day with $\mathrm{transactionsRemaining}$ transactions remaining, and $\mathrm{holding}$ indicating if we start with a stock or not. To answer the original problem, we would return $\mathrm{dp}(0, \mathrm{k}, 0)$, as we start on day 0 with k transactions remaining and not holding a stock.

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

At each state, we need to make a decision that depends on what holding is. Let's split it up and look at our options one at a time:

- If we are holding stock, we have two options. We can sell, or not sell. If we choose to sell, we gain $\text{prices}[\text{i}]$ money, and the next state will be $(\text{i} + 1, \text{transactionsRemaining} - 1, 0)$. This is because it is the next day $(\text{i} + 1)$, we lose a transaction as we completed one by selling ($\text{transactionsRemaining} - 1$), and we are no longer holding a stock (0). In total, our profit is $\text{prices}[\text{i}] + \text{dp}(\text{i} + 1, \text{transactionsRemaining} - 1, 0)$. If we choose not to sell and **do nothing**, then we just move onto the next day with the same number of transactions, while still holding the stock. Our profit is $\text{dp}(\text{i} + 1, \text{transactionsRemaining}, \text{holding})$.
- If we are not holding stock, we have two options. We can buy, or not buy. If we choose to buy, we lose $\text{prices}[\text{i}]$ money, and the next state will be $(\text{i} + 1, \text{transactionsRemaining}, 1)$. This is because it is the next day, we have the same number of transactions because transactions are only completed on selling, and we now hold a stock. In total, our profit is $-\text{prices}[\text{i}] + \text{dp}(\text{i} + 1, \text{transactionsRemaining}, 1)$. If we choose not to buy and **do nothing**, then we just move onto the next day with the same number of transactions, while still not having stock. Our profit is $\text{dp}(\text{i} + 1, \text{transactionsRemaining}, \text{holding})$.

> Note that you could also set up the solution so that transactions are completed upon buying a stock instead.

Of course, we always want to make the best decision. We can see that in both scenarios, **doing nothing** is the same - $\mathrm{dp}(\mathrm{i}+1, \mathrm{transactionsRemaining}, \mathrm{holding})$. Therefore, we have a recurrence relation of:

`dp(i, transactionsRemaining, holding) = max(doNothing, sellStock)` if `holding == 1` otherwise `max(doNothing, buyStock)`

Where,
`doNothing = dp(i + 1, transactionsRemaining, holding)`,
`sellStock = prices[i] + dp(i + 1, transactionsRemaining - 1, 0)`, and
`buyStock = -prices[i] + dp(i + 1, transactionsRemaining, 1)`.

Recurrence relation for Best Time to Buy and Sell Stock IV

State variables:
i: represents day *i*
k: represents remaining transactions allowed
holding: 0 if not holding a stock, 1 if holding a stock

Next state	Buy a stock (holding = 0)	Sell a stock (holding = 1)	Do nothing (both cases)
i	i + 1	i + 1	i + 1
k	k	k - 1	k
holding	1	0	holding
profit	-prices[i] + dp(i+1, k, 1)	prices[i] + dp(i+1, k-1, 0)	dp(i+1, k, holding)

Always choose the option that maximizes profit.

3. **Base cases**

Both base cases are very simple for this problem. If we are out of transactions ($\text{transactionsRemaining} = 0$), then we should immediately return 0 as we cannot make any more money. If the stock is no longer on the market ($\text{i} = \text{prices.length}$), then we should also return 0, as we cannot make any more money.

Top-down Implementation

Java | Python3

Copy

```java
        }

        if (memo[i][transactionsRemaining][holding] == 0) {
            int doNothing = dp(i + 1, transactionsRemaining, holding);
            int doSomething;

            if (holding == 1) {
                // Sell Stock
                doSomething = prices[i] + dp(i + 1, transactionsRemaining - 1, 0);
            } else {
                // Buy Stock
                doSomething = -prices[i] + dp(i + 1, transactionsRemaining, 1);
            }

            // Recurrence relation. Choose the most profitable option.
            memo[i][transactionsRemaining][holding] = Math.max(doNothing, doSomething);
        }

        return memo[i][transactionsRemaining][holding];
    }

    public int maxProfit(int k, int[] prices) {
        this.prices = prices;
        this.memo = new int[prices.length][k + 1][2];
        return dp(0, k, 0);
    }
}
```

Bottom-up Implementation

Again, the recurrence relation is the same with top-down, but we need to be careful about how we configure our for loops. The base cases are automatically handled because the dp array is initialized with all values set to 0. For iteration direction and order, remember with bottom-up we start at the base cases. Therefore we will start iterating from the end of the input and with only 1 transaction remaining.

Java | Python3

Copy

```java
class Solution {
    public int maxProfit(int k, int[] prices) {
        int n = prices.length;
        int dp[][][] = new int[n + 1][k + 1][2];

        for (int i = n - 1; i >= 0; i--) {
            for (int transactionsRemaining = 1; transactionsRemaining <= k; transactionsRemaining++) {
                for (int holding = 0; holding < 2; holding++) {
                    int doNothing = dp[i + 1][transactionsRemaining][holding];
                    int doSomething;
                    if (holding == 1) {
                        // Sell stock
                        doSomething = prices[i] + dp[i + 1][transactionsRemaining - 1][0];
                    } else {
                        // Buy stock
                        doSomething = -prices[i] + dp[i + 1][transactionsRemaining][1];
                    }

                    // Recurrence relation
                    dp[i][transactionsRemaining][holding] = Math.max(doNothing, doSomething);
                }
            }
        }

        return dp[0][k][0];
    }
}
```

The time and space complexity of this problem for both implementations is the number of states since the recurrence relation is just a constant time formula. If $\mathrm{n} = \mathrm{prices.length}$, then this means the time and space complexity is $O(n \cdot k \cdot 2) = O(n \cdot k)$.

Up Next

We'll conclude this chapter with a practice problem similar to the one we just went through here. Remember the pattern of "doing nothing" and try to solve it yourself. If you get stuck, here are some hints:

A State Reduction

Report Issue

In an earlier chapter when we used the framework to solve Maximum Score from Performing Multiplication Operations, we mentioned that we could use 2 state variables instead of 3 because we could derive the information the 3rd one would have given us from the other 2. By doing this, we greatly reduced the number of states (as we learned earlier, the number of states is the product of the number of values each state variable can take). In most cases, reducing the number of states will reduce the time and space complexity of the algorithm.

This is called **state reduction**, and it is applicable for many DP problems, including a few that we have already looked at. State reduction usually comes from a clever trick or observation. Sometimes, as is in the case of Maximum Score from Performing Multiplication Operations, state reduction can result in lower time and space complexity. Other times, only the space complexity will be improved while the time complexity remains the same.

State reduction can also be achieved in the recurrence relation. Recall when we looked at House Robber. Only one state variable was used, i, which indicates what house we are currently at. An alternative way to solve the problem would be adding an extra boolean state variable prev that indicates if we robbed the previous house or not, and that would look something like this:

Python3

Copy

```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        @cache
        def dp(i, prev):
            if i < 0:
                return 0
            ans = dp(i - 1, False)
            if not prev:
                ans = max(ans, dp(i - 1, True) + nums[i])

            return ans

        return dp(len(nums) - 1, False)
```

However, we mentioned in the House Robber article: "*We certainly could include this state variable, but we can develop our recurrence relation in a way that makes it unnecessary.*". By using a clever recurrence relation and base case, we avoided the need for the extra state variable which reduces the number of states by a factor of 2.

> Note: state reductions for space complexity usually only apply to bottom-up implementations, while improving time complexity by reducing the number of state variables applies to both implementations.

When it comes to reducing state variables, it's hard to give any general advice or blueprint. The best advice is to try and think if any of the state variables are related to each other, and if an equation can be created among them. If a problem does not require iteration, there is usually some form of state reduction possible.

Another common scenario where we can improve space complexity is when the recurrence relation is static (no iteration) along one dimension. Let's look back at where we started - Fibonacci. Recall that the i^{th} Fibonacci number can be calculated with the recurrence relation:

$$F(i) = F(i-1) + F(i-2)$$

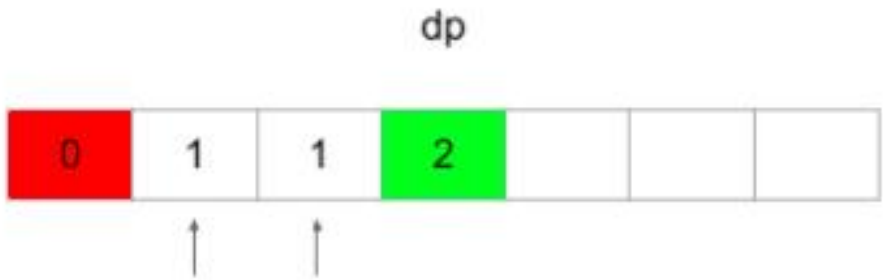
Because this recurrence relation is static, to calculate the i^{th} Fibonacci number, we only ever care about the previous two numbers. That means if we are using a bottom-up approach to find the n^{th} Fibonacci number and start from the base cases, we don't actually need to use an array and remember every single Fibonacci number.

Let's say we wanted $F(100)$. Starting from the base cases, we need to calculate every Fibonacci number from $F(2)$ to $F(99)$, but at the time of the actual calculation for $F(100)$, we only care about $F(98)$ and $F(99)$. The other 90+ Fibonacci numbers aren't needed, so storing all of them is a waste of space.

Fibonacci number, bottom up approach

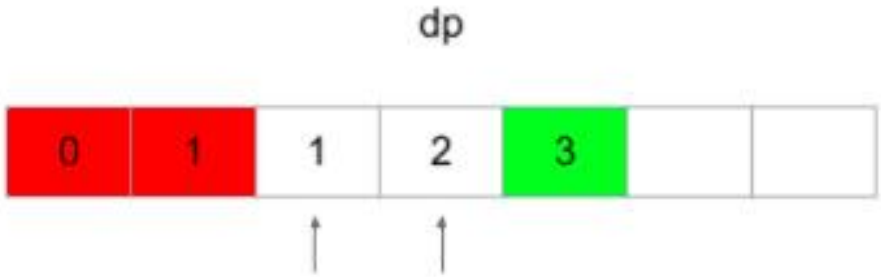
0	1					

We use dp[0] and dp[1] to calculate dp[2].



We use dp[1] and dp[2] to calculate dp[3].

dp[0] will never be used again.



We use dp[2] and dp[3] to calculate dp[4].

dp[0] and dp[1] will never be used again.

You can see that this pattern will continue. Only 2 numbers are needed at any time, so we can save on space by not storing every number we calculate.

Using only two variables instead, we can improve space complexity to $O(1)$ from $O(n)$ using an array. The time complexity remains the same.

Java | Python3

Copy

```java
class Solution {
    public int fib(int n) {
        if (n <= 1) return n;
        int one_back = 1;
        int two_back = 0;

        for (int i = 2; i <= n; i++) {
            int temp = one_back;
            one_back += two_back;
            two_back = temp;
        }

        return one_back;
    }
}
```

> Whenever you notice that values calculated by a DP algorithm are only reused a few times and then never used again, try to see if you can save on space by replacing an array with some variables. A good first step for this is to look at the recurrence relation to see what previous states are used. For example, in Fibonacci, we only refer to the previous two states, so all results before $n - 2$ can be discarded.

[4 variants][easy understanding][dp][recursion]

rajat_gupta_ ★ 1990 October 1, 2020 12:21 PM 1.5K VIEWS

44

1. Recursion [TLE] TC: O(2^n)

```
class Solution {
public:
    int minCostClimbingStairs(vector<int>& cost,int i) {
        if(i==0 || i==1) return 0;
        return min(minCostClimbingStairs(cost,i-2)+cost[i-2],minCostClimbingStairs(cost,i-1)+cost[i-1]);
    }
    int minCostClimbingStairs(vector<int>& cost) {
        return minCostClimbingStairs(cost,cost.size());
    }
};
```

2. dp [runtime beats 95.16 %] TC:O(n) SC:O(n)

```
class Solution {
public:
    int minCostClimbingStairs(vector<int>& cost) {
        int n=cost.size();
        int dp[n];
        dp[0]=cost[0];
        dp[1]=cost[1];
        for(int i=2;i<n;i++)
            dp[i]=min(dp[i-2],dp[i-1])+cost[i];

        return min(dp[n-1],dp[n-2]);
    }
};
```

3. dp [runtime beats 63.90 %] TC:O(n) SC:O(1)

```
class Solution {
public:
    int minCostClimbingStairs(vector<int>& cost) {
        int n=cost.size();
        int first=cost[0],second=cost[1];
        for(int i=2;i<n;i++){
            int r=min(first,second)+cost[i];
            first=second;
            second=r;
        }
        return min(first,second);
    }
};
```

4. dp [runtime beats 95.16 %] TC:O(n) SC:O(1)

```
class Solution {
public:
    int minCostClimbingStairs(vector<int>& cost) {
        int n=cost.size();
        for(int i=2;i<n;i++)
            cost[i]=min(cost[i-1],cost[i-2])+cost[i];
        return min(cost[n-1],cost[n-2]);
    }
};
```

Feel free to ask any question in the comment section.
I hope that you've found the solution useful.
In that case, **please do upvote and encourage me** to on my quest to document all leetcode problems😃
Happy Coding :)

Min Cost Climbing Stairs

Solution

You are given an integer array `cost` where `cost[i]` is the cost of i^{th} step on a staircase. Once you pay the cost, you can either climb one or two steps.

You can either start from the step with index `0`, or the step with index `1`.

Return *the minimum cost to reach the top of the floor*.

```cpp
class Solution {
public:
    int minCostClimbingStairs(vector<int>& cost) {
        int n=cost.size();
        for(int i=2;i<n;i++) cost[i]=min(cost[i-1],cost[i-2])+cost[i];
        return min(cost[n-1],cost[n-2]);
    }
};

```

A Counting DP

Report Issue

Most of the problems we have looked at in earlier chapters ask for either the maximum, minimum, or longest of something. However, it is also very common for a DP problem to ask for the number of distinct ways to do something. In fact, one of the first examples we looked at did this - recall that Climbing Stairs asked us to find the number of ways to climb to the top of the stairs.

> Another term used to describe this class of problems is "counting DP".

What are the differences with counting DP? With the maximum/minimum problems, the recurrence relation typically involves a $\max()$ or $\min()$ function. This is true for all types of problems we have looked at - iteration, multi-dimensional, etc. With counting DP, the recurrence relation typically just sums the results of multiple states together. For example, in Climbing Stairs, the recurrence relation was $\text{dp(i)} = \text{dp(i - 1)} + \text{dp(i - 2)}$. There is no $\max()$ or $\min()$, just addition.

Another difference is in the base cases. In most of the problems we have looked at, if the state goes out of bounds, the base case equals 0. For example, in the Best Time to Buy and Sell Stock questions, when we ran out of transactions or ran out of days to trade, we returned 0 because we can't make any more profit. In Longest Common Subsequence, when we run out of characters for either string, we return 0 because the longest common subsequence of any string and an empty string is 0. With counting DP, the base cases are often not set to 0. This is because the recurrence relation usually only involves addition terms with other states, so if the base case was set to 0 then you would only ever add 0 to itself. Finding these base cases involves some logical thinking - for example, when we looked at Climbing Stairs - we reasoned that there is 1 way to climb to the first step and 2 ways to climb to the second step.

Decode Ways

Solution

A message containing letters from `A-Z` can be **encoded** into numbers using the following mapping:

```
'A' -> "1"
'B' -> "2"
...
'Z' -> "26"
```

To **decode** an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, `"11106"` can be mapped into:

- `"AAJF"` with the grouping `(1 1 10 6)`
- `"KJF"` with the grouping `(11 10 6)`

Note that the grouping `(1 11 06)` is invalid because `"06"` cannot be mapped into `'F'` since `"6"` is different from `"06"`.

Given a string `s` containing only digits, return *the **number** of ways to **decode** it*.

The test cases are generated so that the answer fits in a **32-bit** integer.

Example 1:

```
Input: s = "12"
Output: 2
Explanation: "12" could be decoded as "AB" (1 2) or "L" (12).
```

Please like the post if you found it helpful

Below is the sequential thought process for this problem. First, I tried to solve it with recursion.

Recursion (TLE, Not Accepted)

```
class Solution {
public:
    int numDecodings(string s) {
        int ans=0;
        ans = recur(s,0);
        return ans;
    }
    int recur(string s, int pos){
        if(pos==s.size())return 1;
        if(s[pos]=='0')return 0;
        if(pos==s.size()-1)return 1;
        string b = s.substr(pos,2);

        int way1 = recur(s,pos+1);
        int way2=0;
        if(stoi(b)<=26&&stoi(b)>0){
            way2 = recur(s,pos+2);
        }
        return way1+way2;
    }
};
```

Then, I made a decision tree and saw that some values keep repeating and hence the need to store some values and reduce the extra processing.

Recursion with memoization (Accepted but took too much time)

```cpp
class Solution {
public:
    int numDecodings(string s) {
        vector<int> hg(s.size()+1,0);
        int ans=0;
        ans = recur(s,0,hg);
        return ans;
    }
    int recur(string s, int pos, vector<int>& hg){
        if(pos==s.size())return 1;
        if(s[pos]=='0')return 0;
        if(pos==s.size()-1)return 1;
        if(hg[pos]>0)return hg[pos];
        string b = s.substr(pos,2);

        int way1 = recur(s,pos+1,hg);
        int way2=0;
        if(stoi(b)<=26&&stoi(b)>0){
            way2 = recur(s,pos+2,hg);
        }
        hg[pos]= way1+way2;
        return hg[pos];
    }
};
```

Finally the DP solution which is the best of all.
DP Solution (Best method)

```
class Solution {
public:
    int numDecodings(string s) {
        vector<int> dp(s.size()+1);
        dp[0]=1;
        if(s[0]=='0')dp[1]=0;
        else dp[1]=1;
        for(int i=2;i<=s.size();i++){
            int way1,way2;
            if(s[i-1]=='0')way1=0;
            else way1=dp[i-1];
            if(stoi(s.substr(i-2,2))<=26&&stoi(s.substr(i-2,2))>0&&s[i-2]!='0')way2=dp[i-2];
            else way2=0;
            dp[i]=way1+way2;
        }
        return dp[s.size()];
    }
};
```

C++

```cpp
class Solution {
public:
    int numDecodings(string s) {
        vector<int> dp(s.size()+1);
        dp[0]=1;
        if(s[0]=='0')dp[1]=0;
        else dp[1]=1;
        for(int i=2;i<=s.size();i++){
            int way1,way2;
            if(s[i-1]=='0')way1=0;
            else way1=dp[i-1];
            if(stoi(s.substr(i-2,2))<=26&&stoi(s.substr(i-2,2))>0&&s[i-2]!='0')way2=dp[i-2];
            else way2=0;
            dp[i]=way1+way2;
        }
        return dp[s.size()];
    }
};
```

Kadane's Algorithm

Report Issue

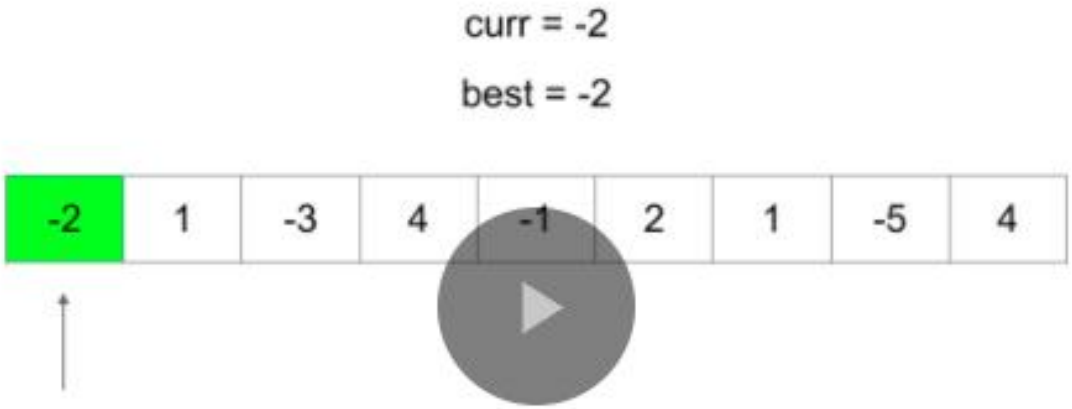
Kadane's Algorithm is an algorithm that can find the maximum sum subarray given an array of numbers in $O(n)$ time and $O(1)$ space. Its implementation is a very simple example of dynamic programming, and the efficiency of the algorithm allows it to be a powerful tool in some DP algorithms. If you haven't already solved Maximum Subarray, take a quick look at the problem before continuing with this article - Kadane's Algorithm specifically solves this problem.

Kadane's Algorithm involves iterating through the array using an integer variable `current`, and at each index `i`, determines if elements before index `i` are "worth" keeping, or if they should be "discarded". The algorithm is only useful when the array can contain negative numbers. If `current` becomes negative, it is reset, and we start considering a new subarray starting at the current index.

Pseudocode for the algorithm is below:

```
// Given an input array of numbers "nums",
1. best = negative infinity
2. current = 0
3. for num in nums:
    3.1. current = Max(current + num, num)
    3.2. best = Max(best, current)

4. return best
```

Line 3.1 of the pseudocode is where the magic happens. If `current` has become less than 0 from including too many or too large negative numbers, the algorithm "throws it away" and resets.



curr = 1

best = 1

The running count of -2 prior to this 1 is not worth keeping, so restart


curr = -2
best = 1
-2
1
-3
4
-1
2
1
-5
4

curr = 4

best = 4

The running count of -2 prior to this 4 is not worth keeping, so restart

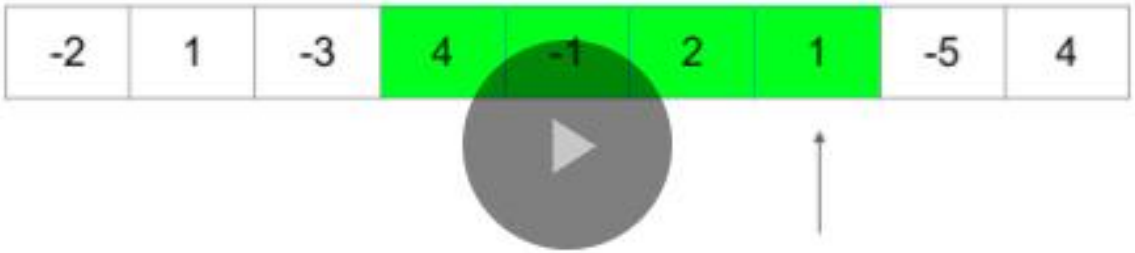

curr = 3
best = 4
-2
1
-3
4
-1
2
1
-5
4

curr = 5

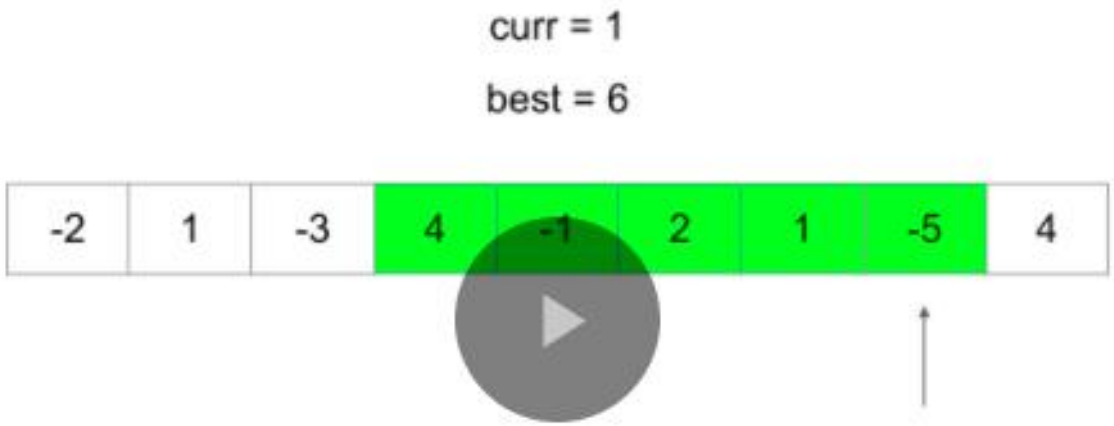
best = 5

curr = 6

best = 6



This entire time, our running count has stayed positive, so we're happy to keep it


curr = 1
best = 6
-2
1
-3
4
-1
2
1
-5
4

curr = 5

best = 6

best = 6

The highest subarray was found here.

While usage of Kadane's Algorithm is a niche, variations of Kadane's Algorithm can be used to develop extremely efficient DP algorithms. Try the next two practice problems with this in mind. No framework hints are provided here as implementations of Kadane's Algorithm do not typically follow the framework intuitively, although they are still *technically* dynamic programming (Kadane's Algorithm utilizes optimal sub-structures - it keeps the maximum subarray ending at the previous position in `current`).

Best Time to Buy and Sell Stock

Solution

You are given an array `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0`.

Example 1:

```
Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.
```

Example 2:

```
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transactions are done and the max profit = 0.
```

Constraints:

- `1 <= prices.length <= 10^5`
- `0 <= prices[i] <= 10^4`

C++

```cpp
class Solution{
     public:
    int maxProfit(vector<int> prices)
    {
         //for all the stocks, you must buy on previous day and sell on future day
        int maxCur = 0, maxSoFar = 0;
        for(int i = 1; i < prices.size(); i++)
        {
            maxCur = max(0, maxCur += prices[i] - prices[i-1]); //updating the current max
            maxSoFar = max(maxCur, maxSoFar);
            //updating the max so far which is max of currentmax and max so far
        }
        return maxSoFar; //at the end returning max so far
    }
};
```

Maximum Sum Circular Subarray

Solution

Given a **circular integer array** `nums` of length `n`, return *the maximum possible sum of a non-empty* ***subarray*** *of* `nums`.

A **circular array** means the end of the array connects to the beginning of the array. Formally, the next element of `nums[i]` is `nums[(i + 1) % n]` and the previous element of `nums[i]` is `nums[(i - 1 + n) % n]`.

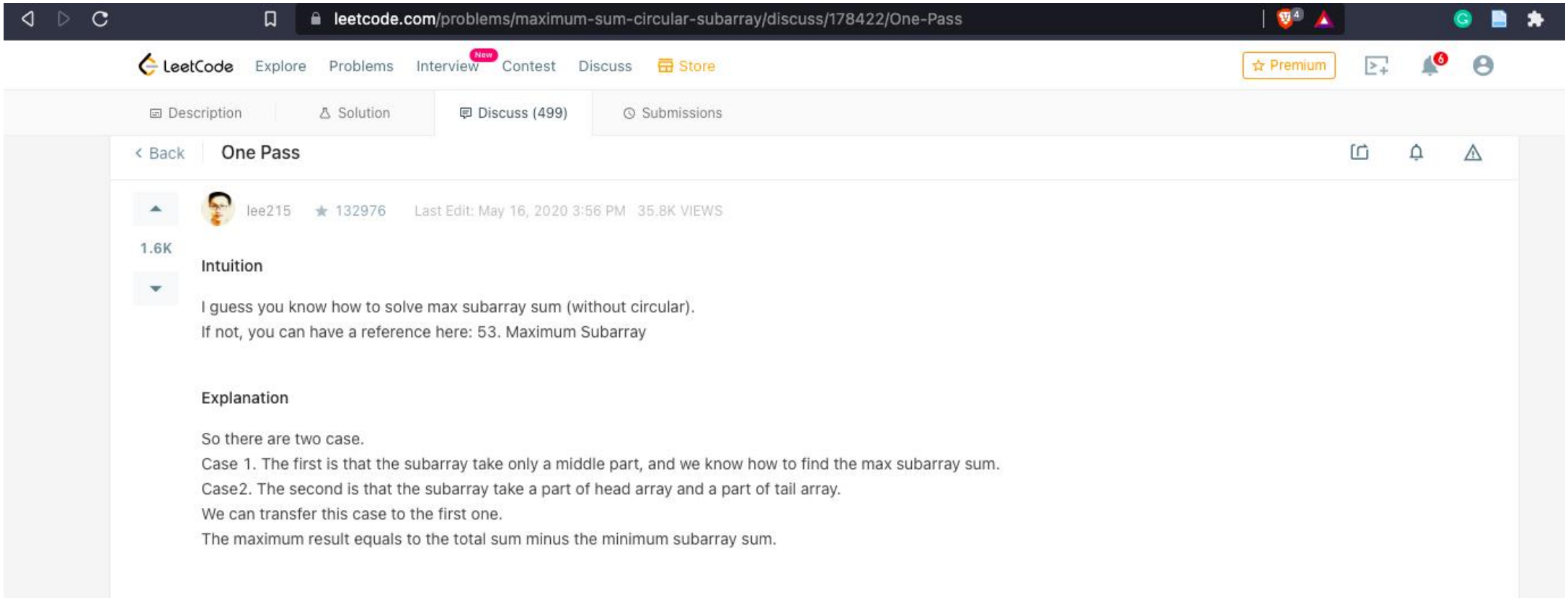
A **subarray** may only include each element of the fixed buffer `nums` at most once. Formally, for a subarray `nums[i], nums[i + 1], ..., nums[j]`, there does not exist `i <= k1`, `k2 <= j` with `k1 % n == k2 % n`.

Example 1:

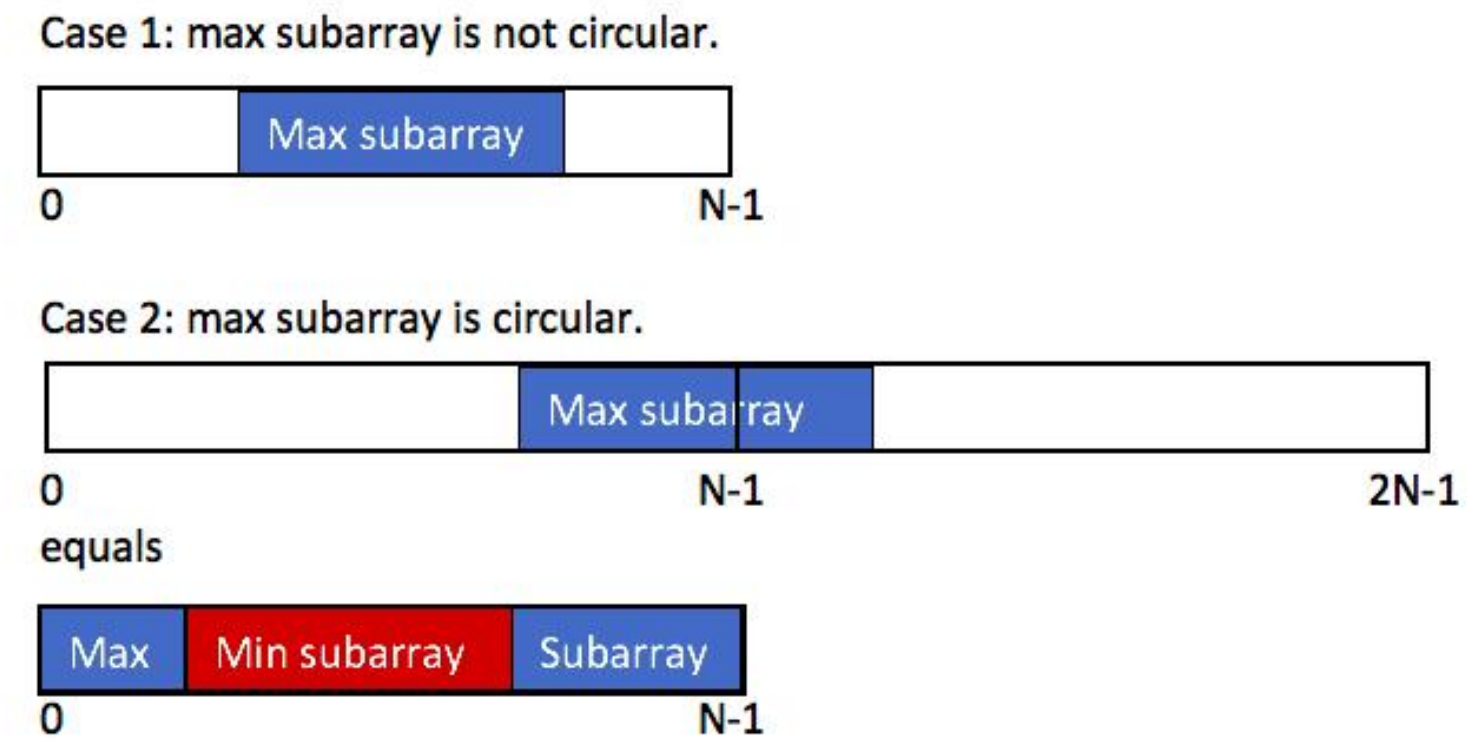
```
Input: nums = [1,-2,3,-2]
Output: 3
Explanation: Subarray [3] has maximum sum 3.
```

Example 2:

```
Input: nums = [5,-3,5]
Output: 10
Explanation: Subarray [5,5] has maximum sum 5 + 5 = 10.
```


leetcode.com/problems/maximum-sum-circular-subarray/discuss/178422/One-Pass
LeetCode
Explore
Problems
Interview
New
Contest
Discuss
Store
Premium
Description
Solution
Discuss (499)
Submissions
Back
One Pass
lee215
132976
Last Edit: May 16, 2020 3:56 PM
35.8K VIEWS
1.6K
Intuition
I guess you know how to solve max subarray sum (without circular).
If not, you can have a reference here: 53. Maximum Subarray
Explanation
So there are two case.
Case 1. The first is that the subarray take only a middle part, and we know how to find the max subarray sum.
Case2. The second is that the subarray take a part of head array and a part of tail array.
We can transfer this case to the first one.
The maximum result equals to the total sum minus the minimum subarray sum.

Here is a diagram by @motorix:



So the max subarray circular sum equals to

```
max(the max subarray sum, the total sum - the min subarray sum)
```

Prove of the second case

max(prefix+suffix)
= max(total sum - subarray)
= total sum + max(-subarray)
= total sum - min(subarray)

Corner case

Just one to pay attention:
If all numbers are negative, `maxSum = max(A)` and `minSum = sum(A)`.
In this case, `max(maxSum, total - minSum) = 0`, which means the sum of an empty subarray.
According to the deacription, We need to return the `max(A)`, instead of sum of am empty subarray.
So we return the `maxSum` to handle this corner case.

Complexity

One pass, time `O(N)`
No extra space, space `O(1)`

C++:

```
int maxSubarraySumCircular(vector<int>& A) {
    int total = 0, maxSum = A[0], curMax = 0, minSum = A[0], curMin = 0;
    for (int& a : A) {
        curMax = max(curMax + a, a);
        maxSum = max(maxSum, curMax);
        curMin = min(curMin + a, a);
        minSum = min(minSum, curMin);
        total += a;
    }
    return maxSum > 0 ? max(maxSum, total - minSum) : maxSum;
}
```

Multiple Choice Question We can reduce space complexity when:

- ○ **We are using top-down and not bottom-up**
- ◉ **The recurrence relation only involves a fixed number of states**
- ○ **Only when we have more than 1 state variable**

↺ Redo ✓ Submit

Correct.

Usually, reductions involve only temporarily caching results to subproblems. Once we know that the result of a subproblem will never be used again, we can discard it. This is a common optimization in bottom-up implementations. Because we iterate over all of the states in an orderly manner, we can determine which cached results will not be used again based on the recurrence relation and the current state.

Multiple Choice Question On its own, Kadane's Algorithm finds:

- (●) **Maximum subarray sum**
- ( ) **Average number in subarray**
- ( ) **Largest number in subarray**

↺ Redo ✓ Submit

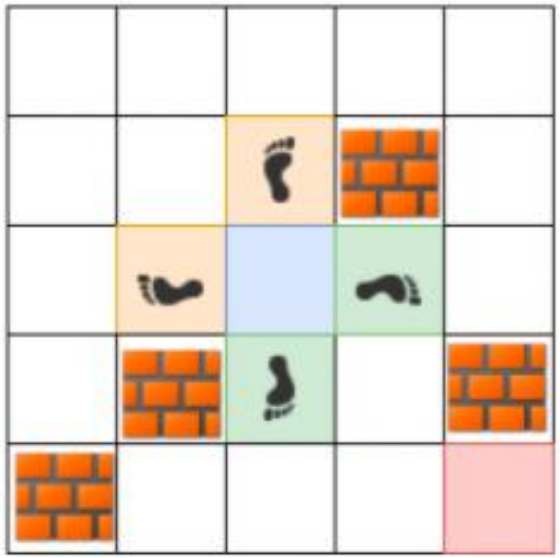
Correct.

Kadane's Algorithm can be extended to create very efficient algorithms.

A Pathing Problems

Report Issue

The last pattern we'll be looking at is pathing problems on a matrix. These problems have matrices as part of the input and give rules for "moving" through the matrix in the problem description. Typically, DP will be applicable when the allowed movement is constrained in a way that prevents moving "backwards", for example if we are only allowed to move down and right.



If we are allowed to move in all 4 directions, then it might be a graph/BFS problem instead. This pattern is sometimes combined with other patterns we have looked at, such as counting DP.

In terms of difficulty, these problems are usually less difficult than the average DP problem as the recurrence relation is usually directly related to the rules of traversal. Most of these problems are also very similar or are variations of each other, and because of this, knowing a general approach to these problems can go a long way.

Let's walk through one last example with the framework, and then finish this card with a few good practice problems.

A Example 62. Unique Paths

Report Issue

> The bottom-up approach for pathing problems is often more intuitive than bottom-up for other types of dynamic programming problems - so this is a good chance for us to practice starting with a bottom-up approach.

In this article, we'll use the framework to solve Unique Paths. This problem asks us to find the number of distinct ways to do something, which is a hint that we should consider using dynamic programming, and we discussed how to do so in the previous chapter. Let's get started:

1. An **array** that answers the problem for a given state

State variables are usually easy to find in pathing problems. Similar to how we need one index (i) for 1D array inputs, with pathing problems on a 2D matrix, we need two indices (row and col) to denote position. Some problems have added constraints that will require additional state variables, but there doesn't seem to be anything of the sort in this problem. Therefore, we will just use two state variables row which represents the current row, and col which represents the current column.

The problem is asking for the number of paths to the final square, so let's have $\mathrm{dp}[\mathrm{row}][\mathrm{col}]$ represent how many paths there are from the start (top-left corner) to the square at $(\mathrm{row}, \mathrm{col})$. We will return $\mathrm{dp}[\mathrm{m} - 1][\mathrm{n} - 1]$ where m and n are the number of rows and columns respectively.

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

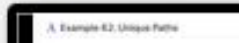
The problem says that we are allowed to move down or right. That means, if we are at some square, we arrived from either the square above or the square to the left. These two squares are $(\text{row - } 1, \text{col})$ and $(\text{row}, \text{col - } 1)$. Since we can arrive at the current square from either of these squares, the number of ways to get to the current square is the sum of the number of ways to get to these two squares. Either of these may be out of the grid bounds, so we should make sure to check for that. This gives us our simple recurrence relation:

$\text{dp}[\text{row}][\text{col}] = \text{dp}[\text{row - } 1][\text{col}] + \text{dp}[\text{row}][\text{col - } 1]$, where $\text{dp}[\text{row - } 1][\text{col}]$ and $\text{dp}[\text{row}][\text{col - } 1]$ is equal to 0 if out of bounds.

3. **Base cases**

In the previous chapter, when talking about counting DP problems, we said that the base cases need to be set to nonzero values so that the terms in the recurrence relation don't just stay stuck at zero. In this problem, we start in the top-left corner. How many ways are there for us to get to the first square? Only 1 - we start on it. Therefore, our base case is $\text{dp}[0][0] = 1$.

> Note: If you have trouble coming up with the recurrence relation, sometimes it helps to come up with the base case(s) first. Then walk through how you would find the result for states that are slightly more complicated than the base case(s), such as $\text{dp}[0][1]$, $\text{dp}[1][1]$, and $\text{dp}[2][1]$. Often, this process of manually solving the problem for simple states can help you understand what the recurrence relation should be.



Bottom-up Implementation

Putting it all together for the final solution:

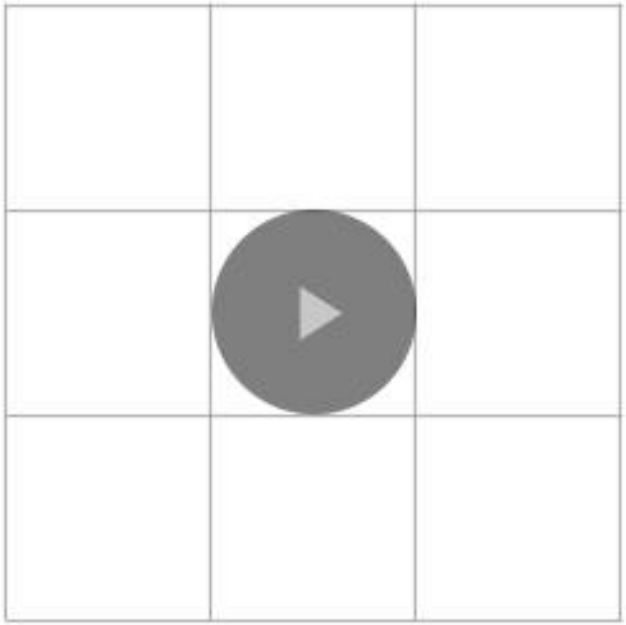
Java | Python3

Copy

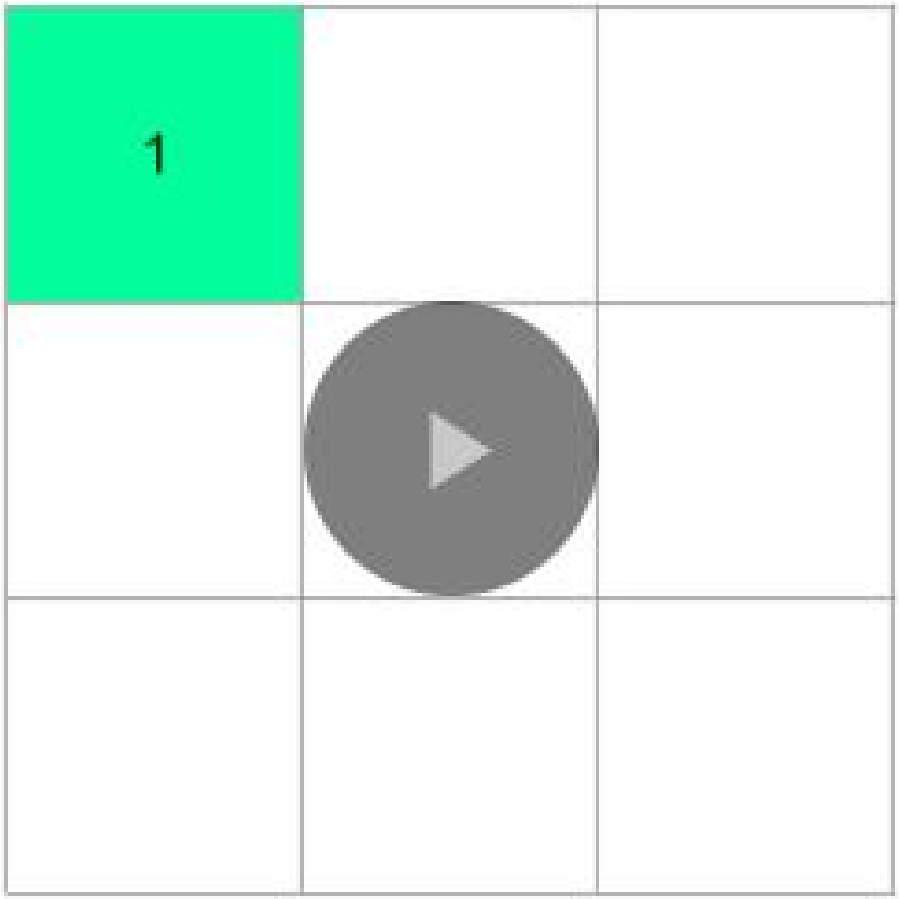
```java
class Solution {
    public int uniquePaths(int m, int n) {
        int[][] dp = new int[m][n];
        dp[0][0] = 1; // Base case

        for (int row = 0; row < m; row++) {
            for (int col = 0; col < n; col++) {
                if (row > 0) {
                    dp[row][col] += dp[row - 1][col];
                }
                if (col > 0) {
                    dp[row][col] += dp[row][col - 1];
                }
            }
        }

        return dp[m - 1][n - 1];
    }
}
```

Here's an animation showing the algorithm in action:

How many unique paths are there from top left to bottom right?

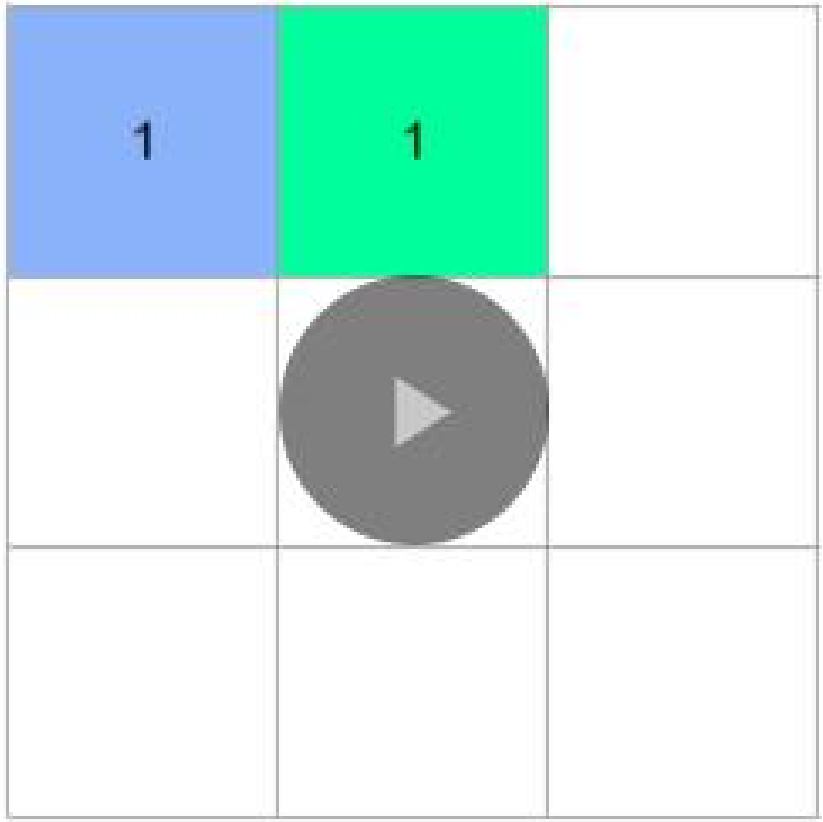


Base case: there is one “path” to get to the start



Iterate row by row using the recurrence relation (squares out of bounds count as 0):

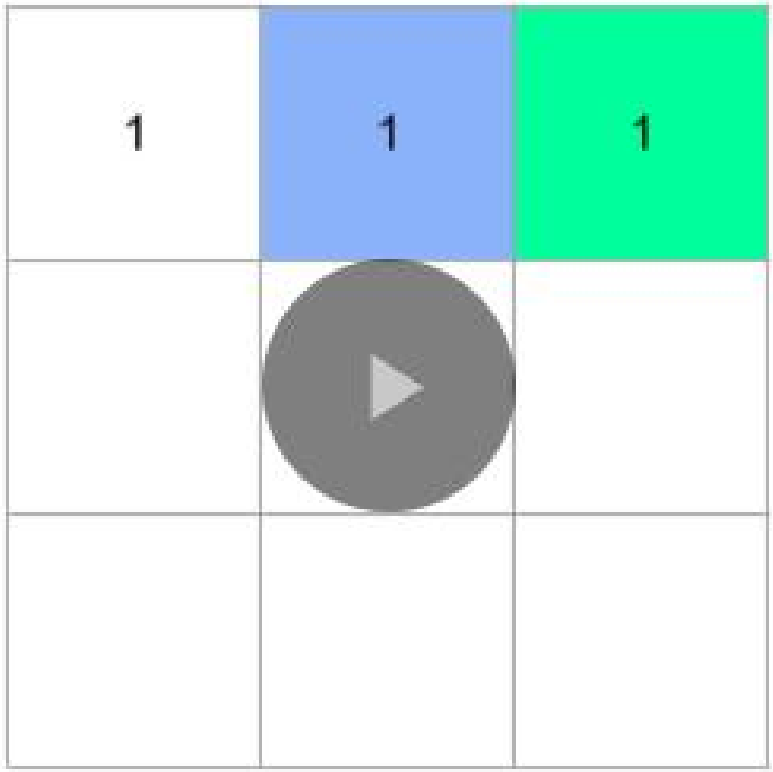
$$dp[row][col] = dp[row - 1][col] + dp[row][col - 1]$$



Green = current square
Blue = squares from the recurrence relation

Iterate row by row using the recurrence relation (squares out of bounds count as 0):

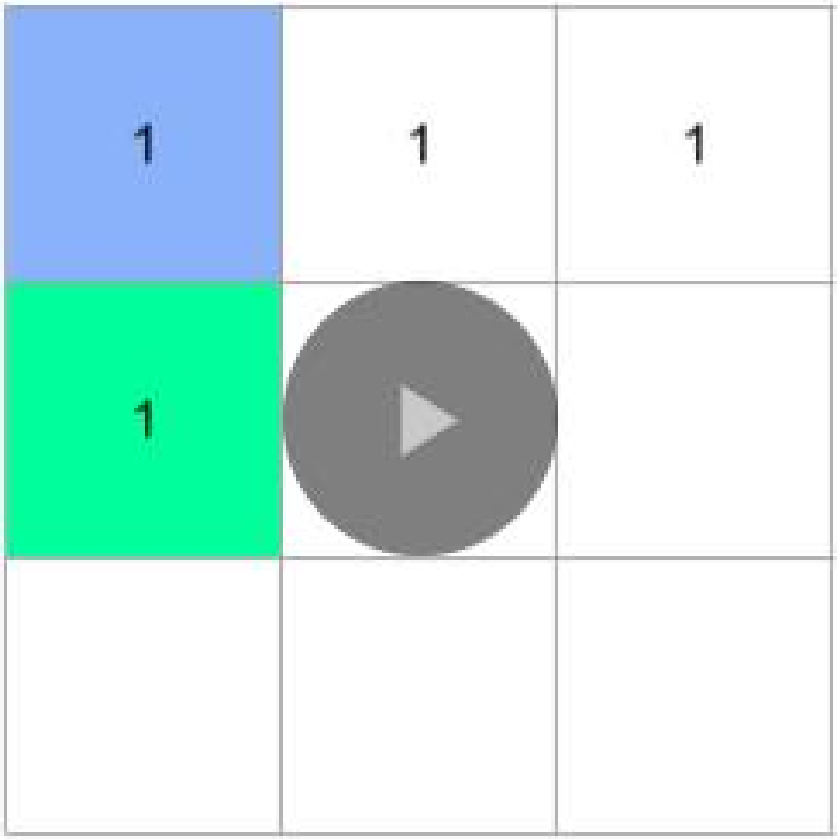
$$dp[row][col] = dp[row - 1][col] + dp[row][col - 1]$$



Green = current square
Blue = squares from the recurrence relation

Iterate row by row using the recurrence relation (squares out of bounds count as 0):

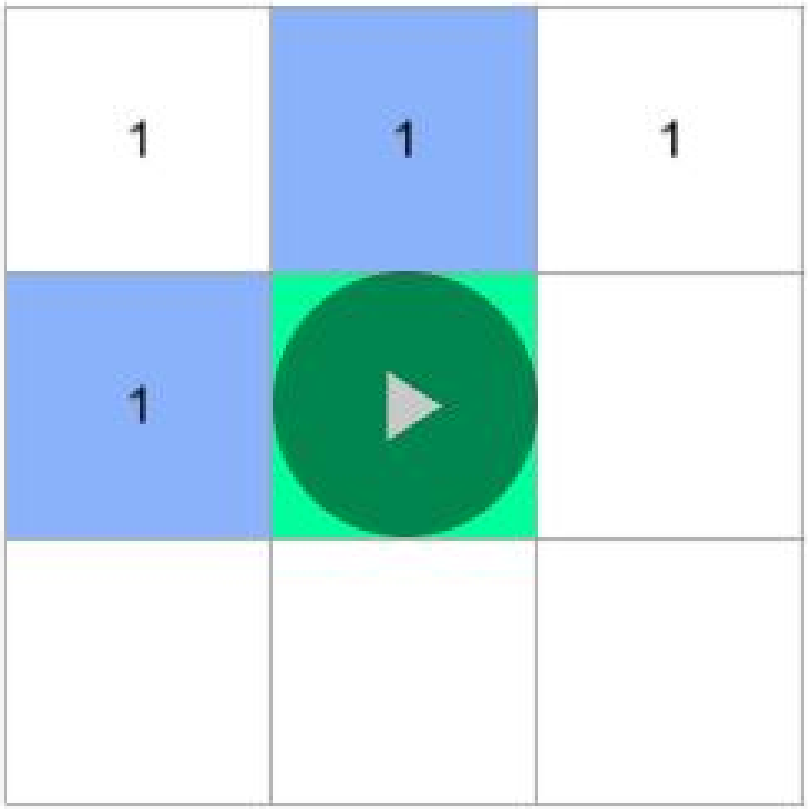
$$dp[row][col] = dp[row - 1][col] + dp[row][col - 1]$$



Green = current square
Blue = squares from the recurrence relation

Iterate row by row using the recurrence relation (squares out of bounds count as 0):

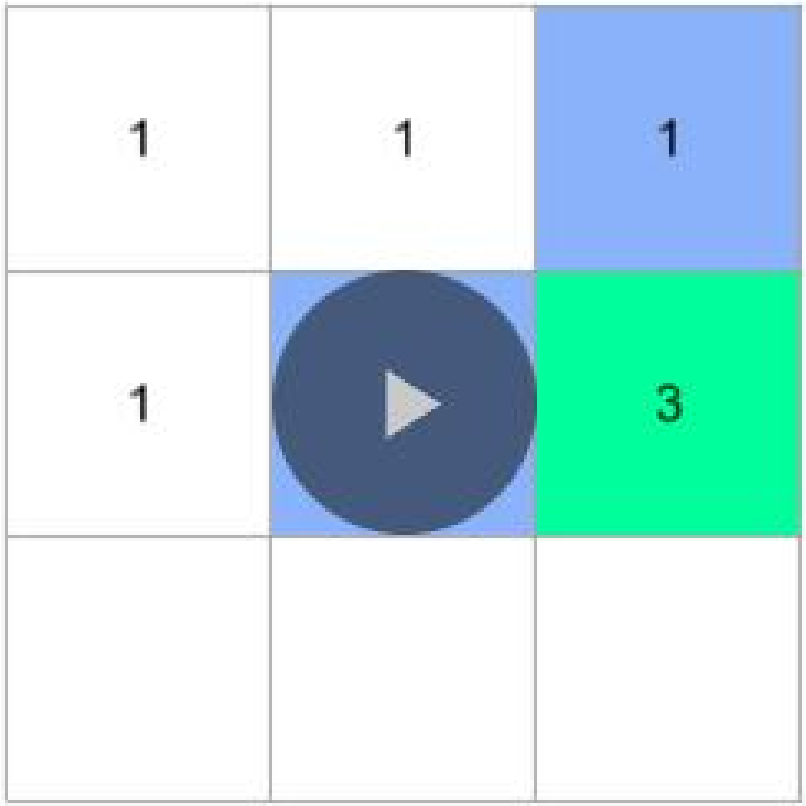
$$dp[row][col] = dp[row - 1][col] + dp[row][col - 1]$$



Green = current square
Blue = squares from the recurrence relation

Iterate row by row using the recurrence relation (squares out of bounds count as 0):

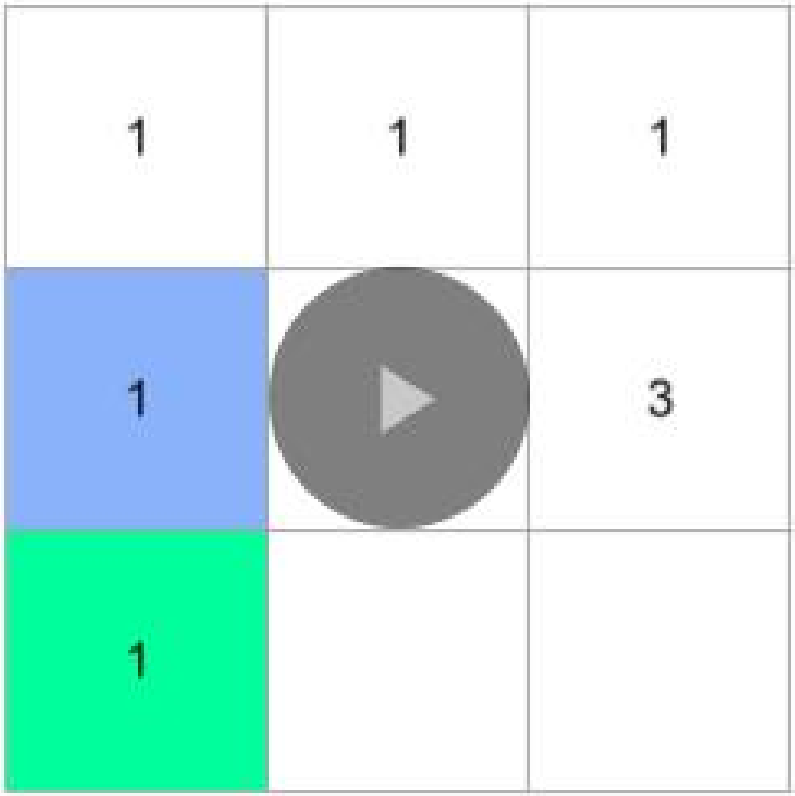
$$dp[row][col] = dp[row - 1][col] + dp[row][col - 1]$$



Green = current square
Blue = squares from the recurrence relation

Iterate row by row using the recurrence relation (squares out of bounds count as 0):

dp[row][col] = dp[row - 1][col] + dp[row][col - 1]



Green = current square
Blue = squares from the recurrence relation

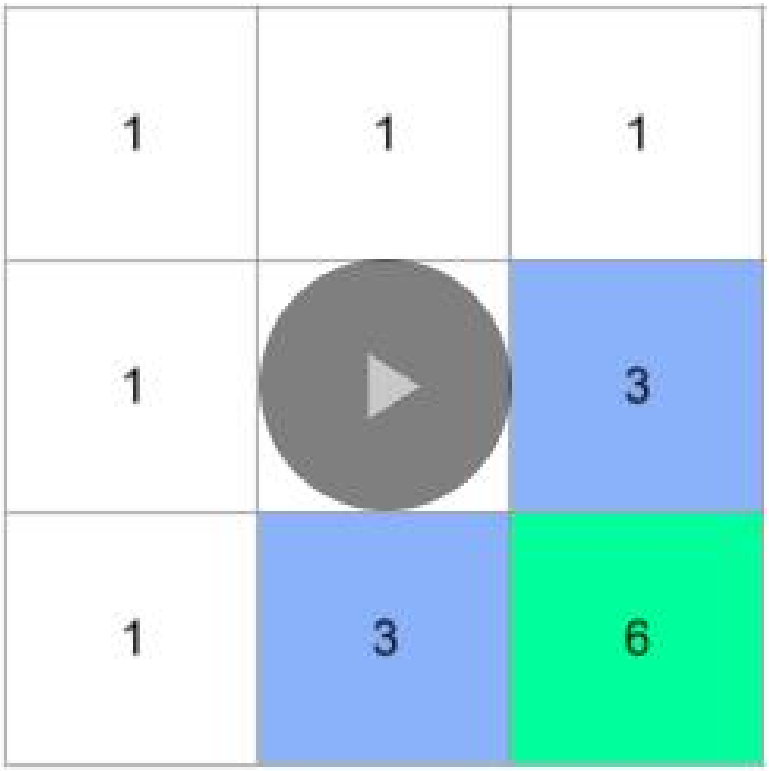
Iterate row by row using the recurrence relation (squares out of bounds count as 0):

dp[row][col] = dp[row - 1][col] + dp[row][col - 1]

Green = current square
Blue = squares from the recurrence relation

Iterate row by row using the recurrence relation (squares out of bounds count as 0):

$$dp[row][col] = dp[row - 1][col] + dp[row][col - 1]$$



There are 6 unique paths.

Top-down Implementation

As we know, one of the advantages of top-down implementations is that they are usually easier to implement. However, for some, bottom-up dynamic programming is more intuitive when it comes to pathing problems. As such, we implemented the bottom-up approach first in this example. That said, it is rare to convert bottom-up solutions to top-down solutions since the typical flow for dynamic programming problems is to come up with a top-down DP solution first, and then convert it to a bottom-up DP solution, if the bottom-up approach is more efficient. Below, for completeness, we have also included the top-down implementation.

Java | Python3 | Copy

```java
class Solution {
    private int[][] memo;

    private int dp(int row, int col) {
        if (row + col == 0) {
            return 1; // Base case
        }

        int ways = 0;
        if (memo[row][col] == 0) {
            if (row > 0) {
                ways += dp(row - 1, col);
            }
            if (col > 0) {
                ways += dp(row, col - 1);
            }

            memo[row][col] = ways;
        }

        return memo[row][col];
    }

```

```
23
24      public int uniquePaths(int m, int n) {
25          memo = new int[m][n];
26          return dp(m - 1, n- 1);
27      }
28  }
```

The time and space complexity of both solutions is $O(m \cdot n)$. We visit each square only once and do a constant amount of work. Here's a challenge to do on your own: try to use state reduction to improve the space complexity of the bottom-up approach to $O(n)$ without modifying the input.

Up next

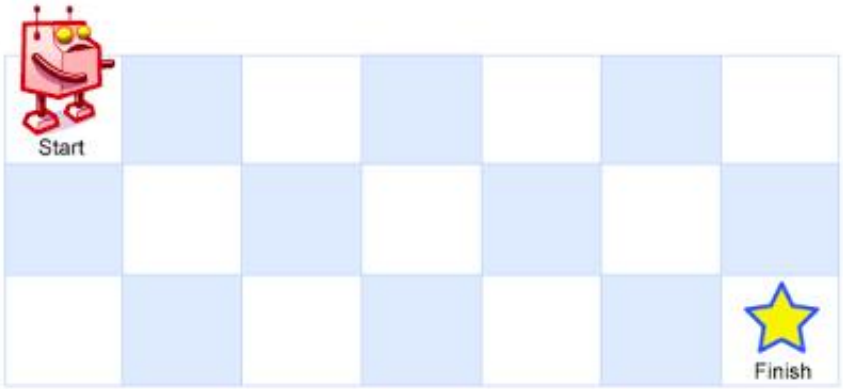
As always, try the next 3 practice problems on your own, and come back here if you need hints:

There is a robot on an `m x n` grid. The robot is initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.

Given the two integers `m` and `n`, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The test cases are generated so that the answer will be less than or equal to `2 * 10^9`.

Example 1:




```
Input: m = 3, n = 7
Output: 28
```

C++

```cpp
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<vector<int>> dp(2, vector<int>(n,1));
        for(int i = 1; i < m; i++)
            for(int j = 1; j < n; j++)
                dp[i & 1][j] = dp[(i-1) & 1][j] + dp[i & 1][j-1];   // <- &  used to alternate between rows
        return dp[(m-1) & 1][n-1];
    }
};
```

Custom Testcase (Contribute)

Run Code Submit

Submission Result: Accepted More Details >

Unique Paths II

Solution

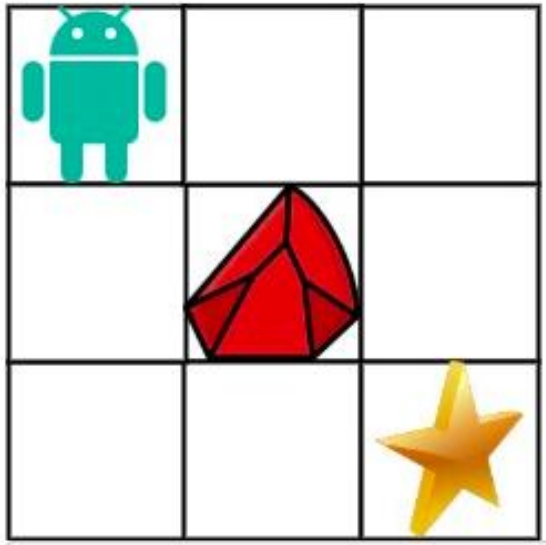
A robot is located at the top-left corner of a `m x n` grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

Now consider if some obstacles are added to the grids. How many unique paths would there be?

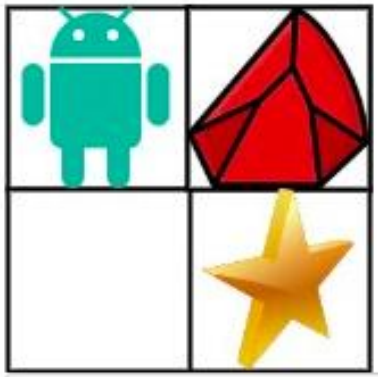
An obstacle and space is marked as `1` and `0` respectively in the grid.

Example 1:



Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
Output: 2
Explanation: There is one obstacle in the middle of the 3x3 grid above.
There are two ways to reach the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right

Example 2:



Input: obstacleGrid = [[0,1],[0,0]]
Output: 1

Constraints:

- `m == obstacleGrid.length`
- `n == obstacleGrid[i].length`
- `1 <= m, n <= 100`
- `obstacleGrid[i][j]` is `0` or `1`.

```cpp
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
    int row = obstacleGrid.size();
    int col = obstacleGrid[0].size();
    vector<int> dp(col,0);
    dp[0]=1;
    for(int i=0;i<row;i++)
        for(int j=0;j<col;j++)
            if(obstacleGrid[i][j]) dp[j]=0;
            else if(j>0) dp[j] += dp[j-1];
    return dp[col-1];
}
};
```

Custom Testcase (Contribute !)

Run Code Submit

Submission Result: Accepted More Details >

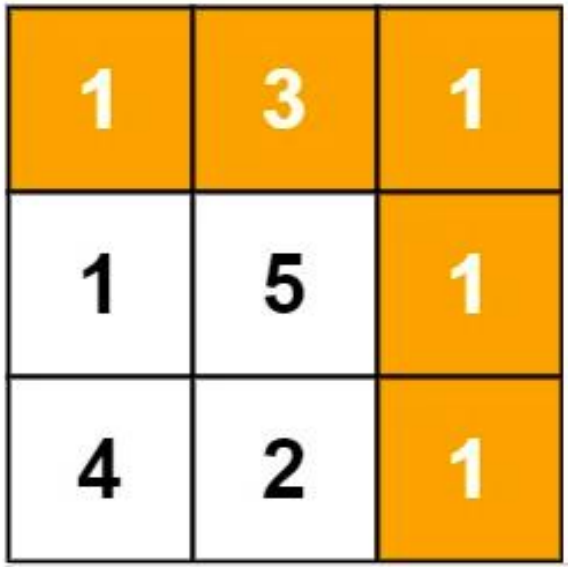
Minimum Path Sum

Solution

Given a `m x n grid` filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Example 1:




```
Input: grid = [[1,3,1],[1,5,1],[4,2,1]]
Output: 7
Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.
```

C++

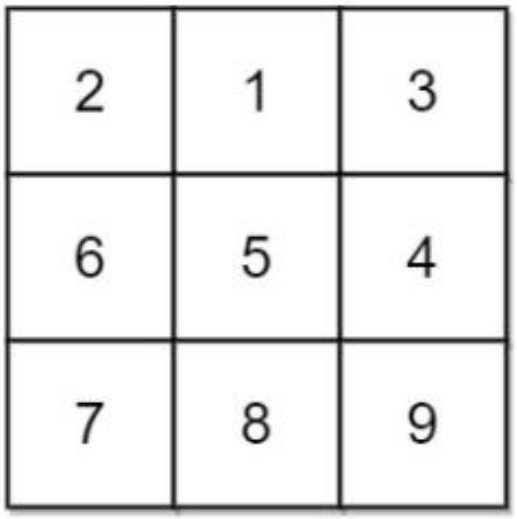
```cpp
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid)
    {
        int m = grid.size();
        int n = grid[0].size();
        vector<int> cur(m, grid[0][0]);
        for (int i = 1; i < m; i++)
            cur[i] = cur[i - 1] + grid[i][0];
        for (int j = 1; j < n; j++)
        {
            cur[0] += grid[0][j];
            for (int i = 1; i < m; i++)
                cur[i] = min(cur[i - 1], cur[i]) + grid[i][j];
        }
        return cur[m - 1];
    }
};
```

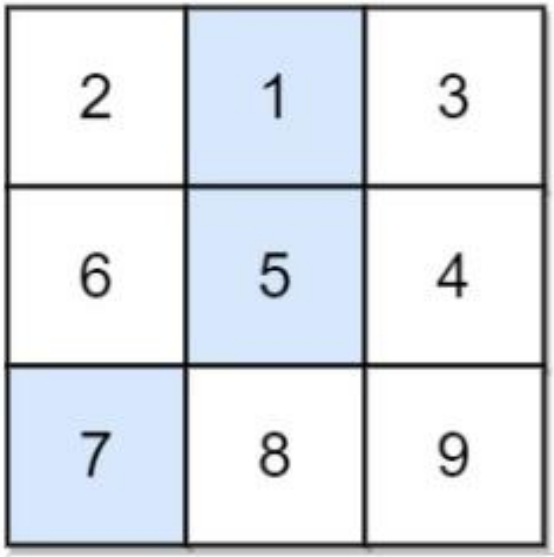
Minimum Falling Path Sum

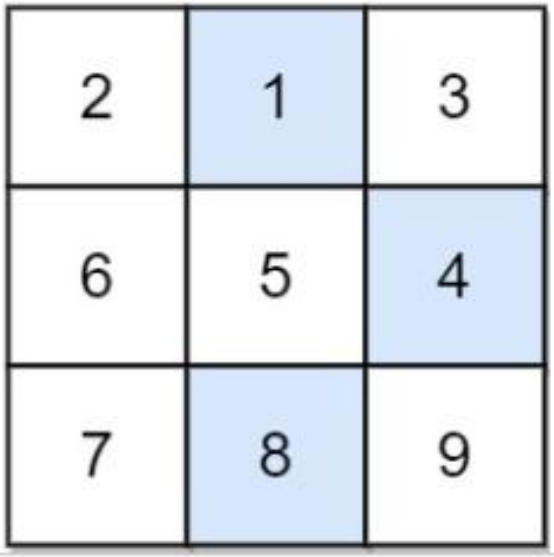
Given an `n x n` array of integers `matrix`, return *the* ***minimum sum*** *of any* ***falling path*** *through* `matrix`.

A **falling path** starts at any element in the first row and chooses the element in the next row that is either directly below or diagonally left/right. Specifically, the next element from position `(row, col)` will be `(row + 1, col - 1)`, `(row + 1, col)`, or `(row + 1, col + 1)`.

Example 1:

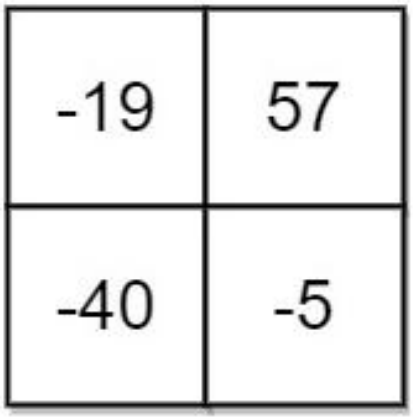


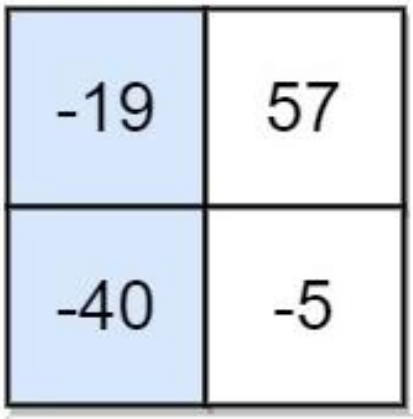




Input: matrix = [[2,1,3],[6,5,4],[7,8,9]]
Output: 13
Explanation: There are two falling paths with a minimum sum as shown.

Example 2:






```
Input: matrix = [[-19,57],[-40,-5]]
Output: -59
Explanation: The falling path with a minimum sum is shown.
```

Constraints:

- `n == matrix.length == matrix[i].length`
- `1 <= n <= 100`
- `-100 <= matrix[i][j] <= 100`

C++

```cpp
class Solution{
    public:
    int minFallingPathSum(vector<vector<int>>& A) {
    vector<vector<int>> dp(A.size(),vector<int>(A.size(),INT_MAX));
    for(int i=0;i<A.size();i++) dp[0][i] = A[0][i];
    for(int i=1;i<A.size();i++)
        for(int j=0;j<A.size();j++)
            if(j>0 and j<A.size()-1){
                dp[i][j] = min(dp[i-1][j-1]+A[i][j],dp[i-1][j]+A[i][j]);
                dp[i][j] = min(dp[i-1][j+1]+A[i][j],dp[i][j]);
            }
            else if(j==0) dp[i][j] = min(dp[i-1][j+1]+A[i][j],dp[i-1][j]+A[i][j]);
            else if(j == A.size()-1) dp[i][j] = min(dp[i-1][j-1]+A[i][j],dp[i-1][j]+A[i][j]);
    int ans = INT_MAX;
    for(auto& x:dp[dp.size()-1]) ans = min(ans,x);
    return ans;
    }
};
```

Custom Testcase (Contribute)

Run Code Submit

Submission Result: Accepted More Details >

Best Time to Buy and Sell Stock with Transaction Fee

Solution

You are given an array `prices` where `prices[i]` is the price of a given stock on the `i^th` day, and an integer `fee` representing a transaction fee.

Find the maximum profit you can achieve. You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: prices = [1,3,2,8,4,9], fee = 2
Output: 8
Explanation: The maximum profit can be achieved by:
- Buying at prices[0] = 1
- Selling at prices[3] = 8
- Buying at prices[4] = 4
- Selling at prices[5] = 9
The total profit is ((8 - 1) - 2) + ((9 - 4) - 2) = 8.
```

Example 2:

```
Input: prices = [1,3,7,5,10,3], fee = 3
Output: 6
```

Constraints:

- `1 <= prices.length <= 5 * 10^4`
- `1 <= prices[i] < 5 * 10^4`
- `0 <= fee < 5 * 10^4`

Hide Hint #1

Consider the first K stock prices. At the end, the only legal states are that you don't own a share of stock, or that you do. Calculate the most profit you could have under each of these two cases.

< Back | 2 solutions, 2 states DP solutions, clear explanation!

Joy4fun ★ 874 Last Edit: October 25, 2018 11:44 PM 21.3K VIEWS

379

Given any `day I`, its max profit status boils down to one of the two status below:

(1) buy status:
`buy[i]` represents the **max profit** at `day i` in **buy status**, given that **the last action you took is a buy action** at `day K`, where `K<=i`. **And you have the right to sell at** `day i+1`, **or do nothing.**
(2) sell status:
`sell[i]` represents the **max profit** at `day i` in **sell status**, given that **the last action you took is a sell action** at `day K`, where `K<=i`. **And you have the right to buy at** `day i+1`, **or do nothing.**

Let's walk through from **base case**.

Base case:
We can start from **buy status**, which means we buy stock at `day 0`.
`buy[0]=-prices[0]`;
Or we can start from **sell status**, which means we sell stock at `day 0`.
Given that we don't have any stock at hand in day 0, we set sell status to be 0.
`sell[0]=0`;

Status transformation:
At `day i`, we may **buy** stock (from **previous sell status**) or do nothing (from **previous buy status**):
```
buy[i] = Math.max(buy[i - 1], sell[i - 1] - prices[i]);
```
Or
At `day i`, we may **sell** stock (from previous **buy status**) or keep holding (from **previous sell status**):
```
sell[i] = Math.max(sell[i - 1], buy[i - 1] + prices[i]);
```

Finally:
We will return `sell[last_day]` as our result, which represents the max profit at the last day, given that you took sell action at any day before the last day.

We can apply transaction fee at either buy status or sell status.

So here come our two solutions:

Solution I -- pay the fee when buying the stock:

```
public int maxProfit(int[] prices, int fee) {
        if (prices.length <= 1) return 0;
        int days = prices.length, buy[] = new int[days], sell[] = new int[days];
        buy[0]=-prices[0]-fee;
        for (int i = 1; i<days; i++) {
            buy[i] = Math.max(buy[i - 1], sell[i - 1] - prices[i] - fee); // keep the same as day i-1, or buy from sell status at day i-1
            sell[i] = Math.max(sell[i - 1], buy[i - 1] + prices[i]); // keep the same as day i-1, or sell from buy status at day i-1
        }
        return sell[days - 1];
    }
```

Solution II -- pay the fee when selling the stock:

```
    public int maxProfit(int[] prices, int fee) {
        if (prices.length <= 1) return 0;
        int days = prices.length, buy[] = new int[days], sell[] = new int[days];
        buy[0]=-prices[0];
        for (int i = 1; i<days; i++) {
            buy[i] = Math.max(buy[i - 1], sell[i - 1] - prices[i]); // keep the same as day i-1, or buy from sell status at day i-1
            sell[i] = Math.max(sell[i - 1], buy[i - 1] + prices[i] - fee); // keep the same as day i-1, or sell from buy status at day i-1
        }
        return sell[days - 1];
    }
```

EnjoyJourney ★ 20 October 24, 2021 6:08 AM

Great solution! Yet we can optimize the Space to O(1) by this:

```
public int maxProfit(int[] prices, int fee) {
    int soldi = 0;
    int buyi = -prices[0];
    for(int i = 1;i<prices.length;i++){
        int prevBuyi = buyi;
        buyi = Math.max(buyi,soldi-prices[i]);
        soldi = Math.max(soldi,prevBuyi+prices[i]-fee);
    }
    return soldi;
}
```

< Back | Most consistent ways of dealing with the series of stock problems

fun4LeetCode ★ 20270 Last Edit: October 27, 2018 3:12 AM 117.3K VIEWS

3.1K

Note: this is a repost of my original post here with updated solutions for this problem (714. Best Time to Buy and Sell Stock with Transaction Fee). If you are only looking for solutions, you can go directly to each section in part `II -- Applications to specific cases`.

Up to this point, I believe you have finished the following series of stock problems:

1. 121. Best Time to Buy and Sell Stock
2. 122. Best Time to Buy and Sell Stock II
3. 123. Best Time to Buy and Sell Stock III
4. 188. Best Time to Buy and Sell Stock IV
5. 309. Best Time to Buy and Sell Stock with Cooldown
6. 714. Best Time to Buy and Sell Stock with Transaction Fee

For each problem, we've got a couple of excellent posts explaining how to approach it. However, most of the posts failed to identify the connections among these problems and made it hard to develop a consistent way of dealing with this series of problems. Here I will introduce the most generalized solution applicable to all of these problems, and its specialization to each of the six problems above.

`I -- General cases`

The idea begins with the following question: **Given an array representing the price of stocks on each day, what determines the maximum profit we can obtain?**

Most of you can quickly come up with answers like "it depends on which day we are and how many transactions we are allowed to complete". Sure, those are important factors as they manifest themselves in the problem descriptions. However, there is a hidden factor that is not so obvious but vital in determining the maximum profit, which is elaborated below.

First let's spell out the notations to streamline our analyses. Let `prices` be the stock price array with length `n`, `i` denote the `i-th` day (`i` will go from `0` to `n-1`), `k` denote the maximum number of transactions allowed to complete, `T[i][k]` be the maximum profit that could be gained at the end of the `i-th` day with at most `k` transactions. Apparently we have base cases: `T[-1][k] = T[i][0] = 0`, that is, no stock or no transaction yield no profit (note the first day has `i = 0` so `i = -1` means no stock). Now if we can somehow relate `T[i][k]` to its subproblems like `T[i-1][k], T[i][k-1], T[i-1][k-1], ...`, we will have a working recurrence relation and the problem can be solved recursively. So how do we achieve that?

The most straightforward way would be looking at actions taken on the `i-th` day. How many options do we have? The answer is three: **buy**, **sell**, **rest**. Which one should we take? The answer is: we don't really know, but to find out which one is easy. We can try each option and then choose the one that maximizes our profit, provided there are no other restrictions. However, we do have an extra restriction saying no multiple transactions are allowed at the same time, meaning if we decide to **buy** on the `i-th` day, there should be `0` stock held in our hand before we buy; if we decide to **sell** on the `i-th` day, there should be exactly `1` stock held in our hand before we sell. The number of stocks held in our hand is the hidden factor mentioned above that will affect the action on the `i-th` day and thus affect the maximum profit.

Therefore our definition of `T[i][k]` should really be split into two: `T[i][k][0]` and `T[i][k][1]`, where the **former** denotes the maximum profit at the end of the `i-th` day with at most `k` transactions and with `0` stock in our hand AFTER taking the action, while the **latter** denotes the maximum profit at the end of the `i-th` day with at most `k` transactions and with `1` stock in our hand AFTER taking the action. Now the base cases and the recurrence relations can be written as:

1. Base cases:
   ```
   T[-1][k][0] = 0, T[-1][k][1] = -Infinity
   T[i][0][0] = 0, T[i][0][1] = -Infinity
   ```
2. Recurrence relations:
   ```
   T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])
   T[i][k][1] = max(T[i-1][k][1], T[i-1][k-1][0] - prices[i])
   ```

For the base cases, `T[-1][k][0] = T[i][0][0] = 0` has the same meaning as before while `T[-1][k][1] = T[i][0][1] = -Infinity` emphasizes the fact that it is impossible for us to have `1` stock in hand if there is no stock available or no transactions are allowed.

For `T[i][k][0]` in the recurrence relations, the actions taken on the `i-th` day can only be **rest** and **sell**, since we have `0` stock in our hand at the end of the day. `T[i-1][k][0]` is the maximum profit if action **rest** is taken, while `T[i-1][k][1] + prices[i]` is the maximum profit if action **sell** is taken. Note that the maximum number of allowable transactions remains the same, due to the fact that a transaction consists of two actions coming as a pair -- **buy** and **sell**. Only action **buy** will change the maximum number of transactions allowed (well, there is actually an alternative interpretation, see my comment below).

For `T[i][k][1]` in the recurrence relations, the actions taken on the `i-th` day can only be **rest** and **buy**, since we have `1` stock in our hand at the end of the day. `T[i-1][k][1]` is the maximum profit if action **rest** is taken, while `T[i-1][k-1][0] - prices[i]` is the maximum profit if action **buy** is taken. Note that the maximum number of allowable transactions decreases by one, since buying on the `i-th` day will use one transaction, as explained above.

To find the maximum profit at the end of the last day, we can simply loop through the `prices` array and update `T[i][k][0]` and `T[i][k][1]` according to the recurrence relations above. The final answer will be `T[i][k][0]` (we always have larger profit if we end up with `0` stock in hand).

`II -- Applications to specific cases`

The aforementioned six stock problems are classified by the value of `k`, which is the maximum number of allowable transactions (the last two also have additional requirements such as "cooldown" or "transaction fee"). I will apply the general solution to each of them one by one.

Case I: `k = 1`

For this case, we really have two unknown variables on each day: `T[i][1][0]` and `T[i][1][1]`, and the recurrence relations say:

```
T[i][1][0] = max(T[i-1][1][0], T[i-1][1][1] + prices[i])
T[i][1][1] = max(T[i-1][1][1], T[i-1][0][0] - prices[i]) = max(T[i-1][1][1], -prices[i])
```

where we have taken advantage of the base case `T[i][0][0] = 0` for the second equation.

It is straightforward to write the `O(n)` time and `O(n)` space solution, based on the two equations above. However, if you notice that the maximum profits on the `i-th` day actually only depend on those on the `(i-1)-th` day, the space can be cut down to `O(1)`. Here is the space-optimized solution:

```java
public int maxProfit(int[] prices) {
    int T_i10 = 0, T_i11 = Integer.MIN_VALUE;

    for (int price : prices) {
        T_i10 = Math.max(T_i10, T_i11 + price);
        T_i11 = Math.max(T_i11, -price);
    }

    return T_i10;
}
```

Now let's try to gain some insight of the solution above. If we examine the part inside the loop more carefully, `T_i11` really just represents the maximum value of the negative of all stock prices up to the `i-th` day, or equivalently the minimum value of all the stock prices. As for `T_i10`, we just need to decide which action yields a higher profit, sell or rest. And if action sell is taken, the price at which we bought the stock is `T_i11`, i.e., the minimum value before the `i-th` day. This is exactly what we would do in reality if we want to gain maximum profit. I should point out that this is not the only way of solving the problem for this case. You may find some other nice solutions here.

Case II: `k = +Infinity`

If `k` is positive infinity, then there isn't really any difference between `k` and `k - 1` (wonder why? see my comment below), which implies `T[i-1][k-1][0] = T[i-1][k][0]` and `T[i-1][k-1][1] = T[i-1][k][1]`. Therefore, we still have two unknown variables on each day: `T[i][k][0]` and `T[i][k][1]` with `k = +Infinity`, and the recurrence relations say:

```
T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])
T[i][k][1] = max(T[i-1][k][1], T[i-1][k-1][0] - prices[i]) = max(T[i-1][k][1], T[i-1][k][0] - prices[i])
```

where we have taken advantage of the fact that `T[i-1][k-1][0] = T[i-1][k][0]` for the second equation. The `O(n)` time and `O(1)` space solution is as follows:

```java
public int maxProfit(int[] prices) {
    int T_ik0 = 0, T_ik1 = Integer.MIN_VALUE;

    for (int price : prices) {
        int T_ik0_old = T_ik0;
        T_ik0 = Math.max(T_ik0, T_ik1 + price);
        T_ik1 = Math.max(T_ik1, T_ik0_old - price);
    }

    return T_ik0;
}
```

(**Note**: The caching of the old values of `T_ik0`, that is, the variable `T_ik0_old`, is unnecessary. Special thanks to 0x0101 and elvina for clarifying this.)

This solution suggests a greedy strategy of gaining maximum profit: as long as possible, buy stock at each local minimum and sell at the immediately followed local maximum. This is equivalent to finding increasing subarrays in `prices` (the stock price array), and buying at the beginning price of each subarray while selling at its end price. It's easy to show that this is the same as accumulating profits as long as it is profitable to do so, as demonstrated in this post.

Case III: `k = 2`

Similar to the case where `k = 1`, except now we have four variables instead of two on each day: `T[i][1][0]`, `T[i][1][1]`, `T[i][2][0]`, `T[i][2][1]`, and the recurrence relations are:

```
T[i][2][0] = max(T[i-1][2][0], T[i-1][2][1] + prices[i])
T[i][2][1] = max(T[i-1][2][1], T[i-1][1][0] - prices[i])
T[i][1][0] = max(T[i-1][1][0], T[i-1][1][1] + prices[i])
T[i][1][1] = max(T[i-1][1][1], -prices[i])
```

where again we have taken advantage of the base case `T[i][0][0] = 0` for the last equation. The `O(n)` time and `O(1)` space solution is as follows:

```java
public int maxProfit(int[] prices) {
    int T_i10 = 0, T_i11 = Integer.MIN_VALUE;
    int T_i20 = 0, T_i21 = Integer.MIN_VALUE;

    for (int price : prices) {
        T_i20 = Math.max(T_i20, T_i21 + price);
        T_i21 = Math.max(T_i21, T_i10 - price);
        T_i10 = Math.max(T_i10, T_i11 + price);
        T_i11 = Math.max(T_i11, -price);
    }

    return T_i20;
}
```

Case IV: `k is arbitrary`

This is the most general case so on each day we need to update all the maximum profits with different `k` values corresponding to `0` or `1` stocks in hand at the end of the day. However, there is a minor optimization we can do if `k` exceeds some critical value, beyond which the maximum profit will no long depend on the number of allowable transactions but instead will be bound by the number of available stocks (length of the `prices` array). Let's figure out what this critical value will be.

A profitable transaction takes at least two days (buy at one day and sell at the other, provided the buying price is less than the selling price). If the length of the `prices` array is `n`, the maximum number of profitable transactions is `n/2` (integer division). After that no profitable transaction is possible, which implies the maximum profit will stay the same. Therefore the critical value of `k` is `n/2`. If the given `k` is no less than this value, i.e., `k >= n/2`, we can extend `k` to positive infinity and the problem is equivalent to `Case II`.

The following is the `O(kn)` time and `O(k)` space solution. Without the optimization, the code will be met with TLE for large `k` values.

```java
public int maxProfit(int k, int[] prices) {
    if (k >= prices.length >>> 1) {
        int T_ik0 = 0, T_ik1 = Integer.MIN_VALUE;

        for (int price : prices) {
            int T_ik0_old = T_ik0;
            T_ik0 = Math.max(T_ik0, T_ik1 + price);
            T_ik1 = Math.max(T_ik1, T_ik0_old - price);
        }

        return T_ik0;
    }

    int[] T_ik0 = new int[k + 1];
    int[] T_ik1 = new int[k + 1];
    Arrays.fill(T_ik1, Integer.MIN_VALUE);

    for (int price : prices) {
        for (int j = k; j > 0; j--) {
            T_ik0[j] = Math.max(T_ik0[j], T_ik1[j] + price);
            T_ik1[j] = Math.max(T_ik1[j], T_ik0[j - 1] - price);
        }
    }

    return T_ik0[k];
}
```

The solution is similar to the one found in this post. Here I used backward looping for the `T` array to avoid using temporary variables. It turns out that it is possible to do forward looping without temporary variables, too.

Case V: `k = +Infinity but with cooldown`

This case resembles `Case II` very much due to the fact that they have the same `k` value, except now the recurrence relations have to be modified slightly to account for t

Case V: `k = +Infinity but with cooldown`

This case resembles `Case II` very much due to the fact that they have the same `k` value, except now the recurrence relations have to be modified slightly to account for the "**cooldown**" requirement. The original recurrence relations for `Case II` are given by

```
T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])
T[i][k][1] = max(T[i-1][k][1], T[i-1][k][0] - prices[i])
```

But with "cooldown", we cannot buy on the `i-th` day if a stock is sold on the `(i-1)-th` day. Therefore, in the second equation above, instead of `T[i-1][k][0]`, we should actually use `T[i-2][k][0]` if we intend to buy on the `i-th` day. Everything else remains the same and the new recurrence relations are

```
T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])
T[i][k][1] = max(T[i-1][k][1], T[i-2][k][0] - prices[i])
```

And here is the `O(n)` time and `O(1)` space solution:

```java
public int maxProfit(int[] prices) {
    int T_ik0_pre = 0, T_ik0 = 0, T_ik1 = Integer.MIN_VALUE;

    for (int price : prices) {
        int T_ik0_old = T_ik0;
        T_ik0 = Math.max(T_ik0, T_ik1 + price);
        T_ik1 = Math.max(T_ik1, T_ik0_pre - price);
        T_ik0_pre = T_ik0_old;
    }

    return T_ik0;
}
```

dietpepsi shared a very nice solution here with thinking process, which turns out to be the same as the one above.

Case VI: `k = +Infinity but with transaction fee`

Again this case resembles `Case II` very much as they have the same `k` value, except now the recurrence relations need to be modified slightly to account for the "**transaction fee**" requirement. The original recurrence relations for `Case II` are given by

```
T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])
T[i][k][1] = max(T[i-1][k][1], T[i-1][k][0] - prices[i])
```

Since now we need to pay some fee (denoted as `fee`) for each transaction made, the profit after buying or selling the stock on the `i-th` day should be subtracted by this amount, therefore the new recurrence relations will be either

```
T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])
T[i][k][1] = max(T[i-1][k][1], T[i-1][k][0] - prices[i] - fee)
```

or

```
T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i] - fee)
T[i][k][1] = max(T[i-1][k][1], T[i-1][k][0] - prices[i])
```

Note we have two options as for when to subtract the `fee`. This is because (as I mentioned above) each transaction is characterized by two actions coming as a pair - - **buy** and **sell**. The fee can be paid either when we buy the stock (corresponds to the first set of equations) or when we sell it (corresponds to the second set of equations). The following are the `O(n)` time and `O(1)` space solutions corresponding to these two options, where for the second solution we need to pay attention to possible overflows.

Solution I -- pay the fee when buying the stock:

```java
public int maxProfit(int[] prices, int fee) {
    int T_ik0 = 0, T_ik1 = Integer.MIN_VALUE;

    for (int price : prices) {
        int T_ik0_old = T_ik0;
        T_ik0 = Math.max(T_ik0, T_ik1 + price);
        T_ik1 = Math.max(T_ik1, T_ik0_old - price - fee);
    }

    return T_ik0;
}
```

Solution II -- pay the fee when selling the stock:

```java
public int maxProfit(int[] prices, int fee) {
    long T_ik0 = 0, T_ik1 = Integer.MIN_VALUE;

    for (int price : prices) {
        long T_ik0_old = T_ik0;
        T_ik0 = Math.max(T_ik0, T_ik1 + price - fee);
        T_ik1 = Math.max(T_ik1, T_ik0_old - price);
    }

    return (int)T_ik0;
}
```

`III -- Summary`

In summary, the most general case of the stock problem can be characterized by three factors, the ordinal of the day `i`, the maximum number of allowable transactions `k`, and the number of stocks in our hand at the end of the day. I have shown the recurrence relations for the maximum profits and their termination conditions, which leads to the `O(nk)` time and `O(k)` space solution. The results are then applied to each of the six cases, with the last two using slightly modified recurrence relations due to the additional requirements. I should mention that peterleetcode also introduced a nice solution here which generalizes to arbitrary `k` values. If you have a taste, take a look.

Hope this helps and happy coding!

Comments: 182

Best | Most Votes | Newest to Oldest | Oldest to Newest

Type comment here... (Markdown is supported)

Post

Dico ★ 4842 February 25, 2019 3:32 AM

Damn, 3D DP makes me feel I am out of IQ.

187 Show 2 replies Reply

shuhua ★ 242 October 30, 2017 10:28 AM

Excellent writing! This post seems like a paper.

97 Reply

sha256pki ★ 869 October 24, 2017 7:24 PM

Thank you very much! this is the best post on leetcode so far.

113 Show 2 replies Reply Share Report

Paint House III

There is a row of `m` houses in a small city, each house must be painted with one of the `n` colors (labeled from `1` to `n`), some houses that have been painted last summer should not be painted again.

A neighborhood is a maximal group of continuous houses that are painted with the same color.

- For example: `houses = [1,2,2,3,3,2,1,1]` contains `5` neighborhoods `[{1}, {2,2}, {3,3}, {2}, {1,1}]`.

Given an array `houses`, an `m x n` matrix `cost` and an integer `target` where:

- `houses[i]`: is the color of the house `i`, and `0` if the house is not painted yet.
- `cost[i][j]`: is the cost of paint the house `i` with the color `j + 1`.

Return *the minimum cost of painting all the remaining houses in such a way that there are exactly* `target` *neighborhoods*. If it is not possible, return `-1`.

Example 1:

```
Input: houses = [0,0,0,0,0], cost = [[1,10],[10,1],[10,1],[1,10],[5,1]], m = 5, n = 2, target = 3
Output: 9
Explanation: Paint houses of this way [1,2,2,1,1]
This array contains target = 3 neighborhoods, [{1}, {2,2}, {1,1}].
Cost of paint all houses (1 + 1 + 1 + 1 + 5) = 9.
```

Example 2:

C++

```cpp
class Solution {
public:
    int dp[101][101][21] = {};
    int dfs(vector<int>& houses, vector<vector<int>>& cost, int i, int target, int last_clr) {
        if (i >= houses.size() || target < 0) return target == 0 ? target : 1000001;
        if (houses[i] != 0) // painted last year.
        return dfs(houses, cost, i + 1, target - (last_clr != houses[i]), houses[i]);
        if (dp[i][target][last_clr]) return dp[i][target][last_clr];
        auto res = 1000001;
        for (auto clr = 1; clr <= cost[i].size(); ++clr)
        res = min(res, cost[i][clr - 1]  + dfs(houses, cost, i + 1, target - (last_clr != clr), clr));
        return dp[i][target][last_clr] = res;
    }
    int minCost(vector<int>& houses, vector<vector<int>>& cost, int m, int n, int target) {
        auto res = dfs(houses, cost, 0, target, 0);
        return res > 1000000 ? -1 : res;
    }
};

```

Count Vowels Permutation

Solution

Given an integer `n`, your task is to count how many strings of length `n` can be formed under the following rules:

- Each character is a lower case vowel (`'a'`, `'e'`, `'i'`, `'o'`, `'u'`)
- Each vowel `'a'` may only be followed by an `'e'`.
- Each vowel `'e'` may only be followed by an `'a'` or an `'i'`.
- Each vowel `'i'` **may not** be followed by another `'i'`.
- Each vowel `'o'` may only be followed by an `'i'` or a `'u'`.
- Each vowel `'u'` may only be followed by an `'a'.`

Since the answer may be too large, return it modulo `10^9 + 7.`

Example 1:

```
Input: n = 1
Output: 5
Explanation: All possible strings are: "a", "e", "i" , "o" and "u".
```

Example 2:

```
Input: n = 2
Output: 10
Explanation: All possible strings are: "ae", "ea", "ei", "ia", "ie", "io", "iu", "oi", "ou" and "ua".
```

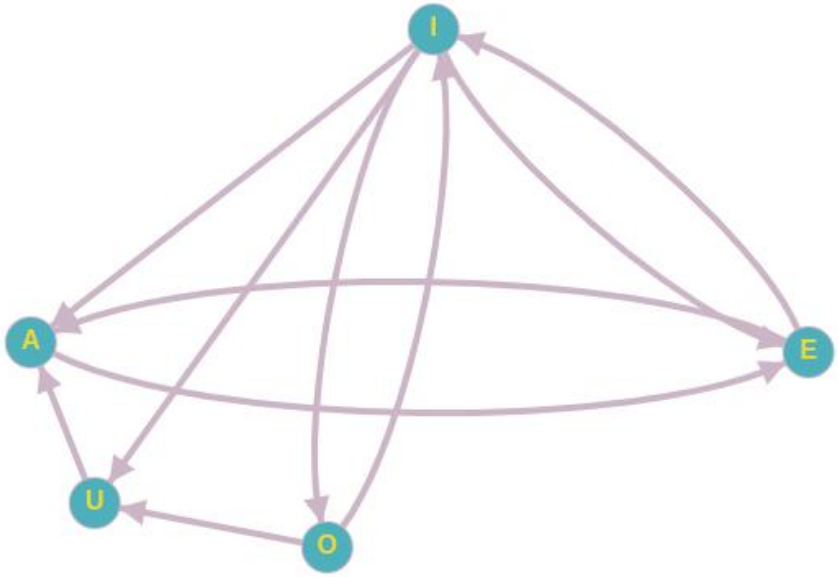
< Back | Detailed Explanation using Graphs [With Pictures] [O(n)]

Just__a__Visitor ★ 2904 Last Edit: October 11, 2019 3:42 PM 9.1K VIEWS

175

Intuition

Let us visualize this as a graph problem. From the above rules, we can create a directed graph where an edge between characters `first` and `second` imply that it is permissible to write `second` immediately after first. Hence, the question converts to, **Given a directed graph, how many paths of length `n` are there?**



Now, Let us say that `dp[n][char]` denotes the number of directed paths of length `n` which end at a particular vertex `char`. Then, we know that the last vertex in our path was `char`. However, let's focus on the last second vertex. It could have been any of the vertex which has a direct edge to `char`. Hence, if we can find the number of paths of length `n-1` ending at these vertices, then we can append `char` at the end of every path and we would have exhausted all possibilites.

Hence, `dp[n+1][x] = sum of all dp[n][y]` such that there is a directed edge from `y` to `x`.

```cpp
class Solution
{
public:
    int countVowelPermutation(int n);
};

int Solution :: countVowelPermutation(int n)
{
    vector<vector<long>> dp(n+1, vector<long>(5, 0));

    int MOD = 1e9 + 7;

    /* dp[i][j] denotes the number of valid strings of length i */

    for(int i = 0; i < 5; i++)
        dp[1][i] = 1;

    for(int i = 1; i < n; i++)
    {
        dp[i+1][0] = (dp[i][1] + dp[i][2] + dp[i][4]) %MOD;

        dp[i+1][1] = (dp[i][0] + dp[i][2]) % MOD;

        dp[i+1][2] = (dp[i][1] + dp[i][3]) % MOD;

        dp[i+1][3] = dp[i][2];

        dp[i+1][4] = (dp[i][2] + dp[i][3]) % MOD;
    }

    int res = 0;
    for(int i = 0; i < 5; i++)
        res = (res + dp[n][i]) % MOD;
```

```
    int res = 0;
    for(int i = 0; i < 5; i++)
        res = (res + dp[n][i]) % MOD;

    return res;
}
```