**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*F*(0)=0 and F(1) = 1*F*(1)=1. With bottom-up, we would use these base cases to calculate F(2)*F*(2), and then use that result to calculate F(3)*F*(3), and so on all the way up to F(n)*F*(*n*).

// Pseudocode example for bottom-up

F = array of length (n + 1)

F[0] = 0

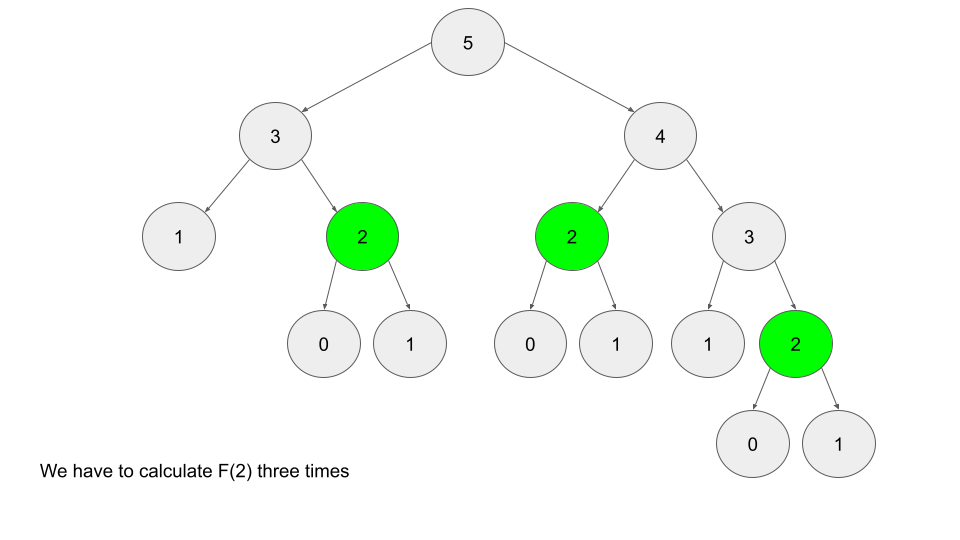
F[1] = 1

for i from 2 to n:

F[i] = F[i - 1] + F[i - 2]

Top-down (Memoization)

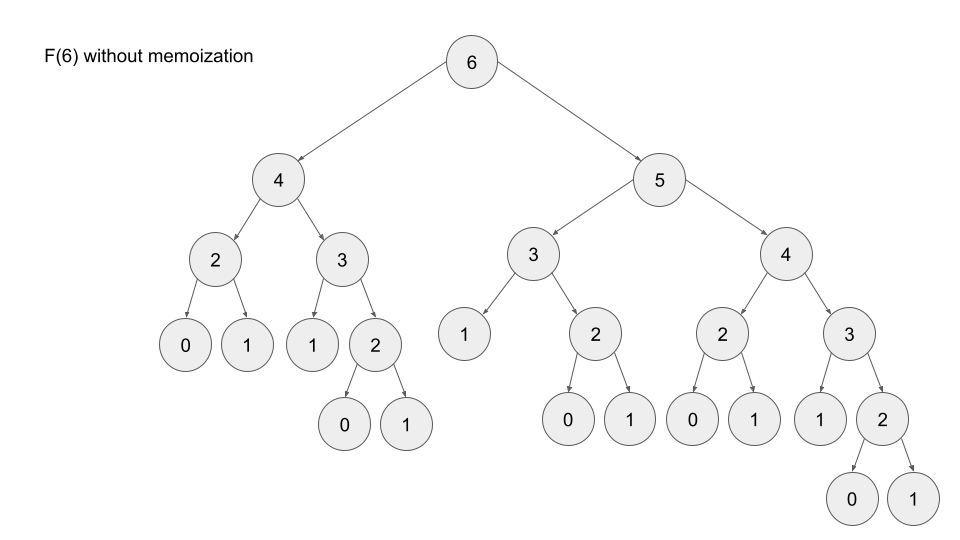
Top-down is implemented with recursion and made efficient with memoization. If we wanted to find the  *nth* Fibonacci number *F*(*n*), we try to compute this by finding  *F*(*n*−1) and *F*(*n*−2). This defines a recursive pattern that will continue on until we reach the base cases *F*(0)=*F*(1)=1. The 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):

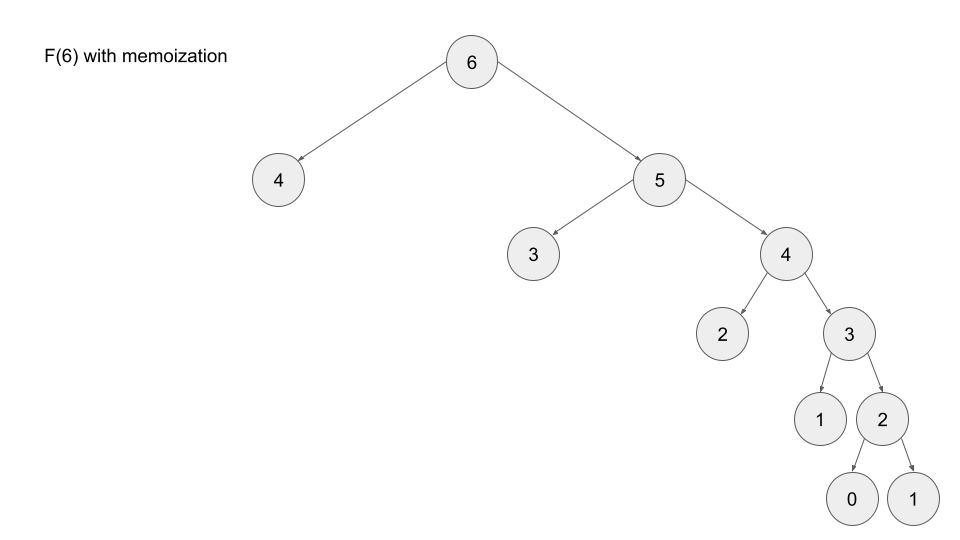


Notice that we need to calculate *F*(2) three times. This might not seem like a big deal, but if we were to calculate *F*(6), this **entire image** would be only one child of the root. Imagine if we wanted to find *F*(100) - the amount of computation is exponential and will quickly explode. The solution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





// 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.

We'll be talking more about these two options throughout the card. For now, all you need to know is that top-down uses recursion, and bottom-up uses iteration.

**When to Use DP**

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](https://leetcode.com/problems/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 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](https://leetcode.com/problems/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 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 10,00010,000 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.

**Multidimensional DP**

The dimensions of a DP algorithm refer to the number of state variables used to define each state. So far in this explore card, all the algorithms we have looked at required only one state variable - therefore they are **one-dimensional**. In this section, we're going to talk about problems that require multiple dimensions.

Typically, the more dimensions a DP problem has, the more difficult it is to solve. Two-dimensional problems are common, and sometimes a problem might even require [five dimensions](https://leetcode.com/problems/maximize-grid-happiness/). The good news is, the framework works regardless of the number of dimensions.

The following are common things to look out for in DP problems that require a state variable:

* An index along some input. This is usually used if an input is given as an array or string. This has been the sole state variable for all the problems that we've looked at so far, and it has represented the answer to the problem if the input was considered only up to that index - for example, if the input is nums = [0, 1, 2, 3, 4, 5, 6], then dp(4) would represent the answer to the problem for the input nums = [0, 1, 2, 3, 4].
* A second index along some input. Sometimes, you need two index state variables, say i and j. In some questions, these variables represent the answer to the original problem if you considered the input starting at index i and ending at index j. Using the same example above, dp(1, 3) would solve the problem for the input nums = [1, 2, 3], if the original input was [0, 1, 2, 3, 4, 5, 6].
* Explicit numerical constraints given in the problem. For example, "you are only allowed to complete k transactions", or "you are allowed to break up to k obstacles", etc.
* Variables that describe statuses in a given state. For example "true if currently holding a key, false if not", "currently holding k packages" etc.
* Some sort of data like a tuple or bitmask used to indicate things being "visited" or "used". For example, "bitmask is a mask where the *ith* bit indicates if the *ith* city has been visited". Note that mutable data structures like arrays cannot be used - typically, only immutable data structures like numbers and strings can be hashed, and therefore memoized.

Multi-dimensional problems make us think harder about deciding what our function or array will represent, as well as what the recurrence relation should look like. In the next article, we'll walk through another example using the framework with a 2D DP problem.

**Top-down to Bottom-up**

As we've said in the previous chapter, **usually** a top-down algorithm is easier to implement than the equivalent bottom-up algorithm. With that being said, it is useful to know how to take a completed top-down algorithm and convert it to bottom-up. There's a number of reasons for this: first, in an interview, if you solve a problem with top-down, you may be asked to rewrite your solution in an iterative manner (using bottom-up) instead. Second, as we mentioned before, bottom-up **usually** is more efficient than top-down in terms of runtime.

**Steps to convert top-down into bottom-up**

1. Start with a completed top-down implementation.
2. Initialize an array dp that is sized according to your state variables. For example, let's say the input to the problem was an array nums and an integer k that represents the maximum number of actions allowed. Your array dp would be 2D with one dimension of length nums.length and the other of length k. The values should be initialized as some default value opposite of what the problem is asking for. For example, if the problem is asking for the maximum of something, set the values to negative infinity. If it is asking for the minimum of something, set the values to infinity.
3. Set your base cases, same as the ones you are using in your top-down function. Recall in House Robber, dp(0) = nums[0] and dp(1) = max(nums[0], nums[1]). In bottom-up, dp[0] = nums[0] and dp[1] = max(nums[0], nums[1]).
4. Write a for-loop(s) that iterate over your state variables. If you have multiple state variables, you will need nested for-loops. These loops should **start iterating from the base cases**.
5. Now, each iteration of the inner-most loop represents a given state, and is equivalent to a function call to the same state in top-down. Copy the logic from your function into the for-loop and change the function calls to accessing your array. All dp(...) changes into dp[...].
6. We're done! dp is now an array populated with the answer to the original problem for all possible states. Return the answer to the original problem, by changing return dp(...) to return dp[...].

Let's try a quick example using the House Robber code from before. Here's a completed top-down solution:

**class** **Solution**:

**def** **rob**(self, nums: List[*int*]) -> *int*:

**def** **dp**(i: *int*) -> *int*:

***# Base cases***

            if i == **0**:

                return nums[**0**]

            elif i == **1**:

                return max(nums[**0**], nums[**1**])

            if i not in memo:

***# Use recurrence relation to calculate dp[i].***

                memo[i] = max(dp(i - **1**), dp(i - **2**) + nums[i])

            return memo[i]

        memo = {}

        return dp(len(nums) - **1**)

First, we initialize an array dp sized according to our state variables. Our only state variable is i which can take n values.

**class** **Solution**:

**def** **rob**(self, nums: List[*int*]) -> *int*:

        n = len(nums)

        dp = [**0**] \* n

        return dp[n - **1**]

Second, we should set our base cases. dp[0] = nums[0] and dp[1] = max(nums[0], nums[1]). To avoid index out of bounds, we should also just return nums[0] if theres only one house.

**class** **Solution**:

**def** **rob**(self, nums: List[*int*]) -> *int*:

        n = len(nums)

        if n == **1**:

            return nums[**0**]

        dp = [**0**] \* n

***#Base Cases***

        dp[**0**] = nums[**0**]

        dp[**1**] = max(nums[**0**], nums[**1**])

        return dp[n - **1**]

Next, write a for-loop to iterate over the state variables, starting from the base cases.

**class** **Solution**:

**def** **rob**(self, nums: List[*int*]) -> *int*:

        n = len(nums)

        if n == **1**:

            return nums[**0**]

        dp = [**0**] \* n

***#Base Cases***

        dp[**0**] = nums[**0**]

        dp[**1**] = max(nums[**0**], nums[**1**])

        for i in range(**2**, n):

            pass

        return dp[n - **1**]

Lastly, copy the recurrence relation over from the top-down solution and put it in the for-loop. Return dp[n - 1].

**class** **Solution**:

**def** **rob**(self, nums: List[*int*]) -> *int*:

        n = len(nums)

        if n == **1**:

            return nums[**0**]

        dp = [**0**] \* n

***#Base Cases***

        dp[**0**] = nums[**0**]

        dp[**1**] = max(nums[**0**], nums[**1**])

        for i in range(**2**, n):

***# Use recurrence relation to calculate dp[i].***

            dp[i] = max(dp[i - **1**], dp[i - **2**] + nums[i])

        return dp[n - **1**]

**Time and Space Complexity**

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*⋅*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 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 holding is a boolean variable. What will the time complexity be for a DP algorithm that solves this problem? Let n = nums.length and K be the maximum actions possible given in the input. i can be from 0 to nums.length, k can be from 0 to K, and holding }can be true or false. Therefore, there are  n⋅K⋅2 states. If computing each state is *O*(1), then the time complexity will be *O*(*n*⋅*K*⋅2)=*O*(*n*⋅*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)*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.

**Common Patterns in DP**

**Iteration in the recurrence relation**

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](https://leetcode.com/problems/min-cost-climbing-stairs/). The recurrence relation was:

dp(i)=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:

dp(i)=min(dp(j) + cost[j]) for all (i - k)≤j<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.

**State Transition by Inaction**

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 ( ?dp(i) = 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 dp(i, j) = 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 dp(i) = dp(i - 1).

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

**State Reduction**

In an earlier chapter when we used the framework to solve [Maximum Score from Performing Multiplication Operations](https://leetcode.com/explore/learn/card/dynamic-programming/631/strategy-for-solving-dp-problems/4100/), 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:

**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.

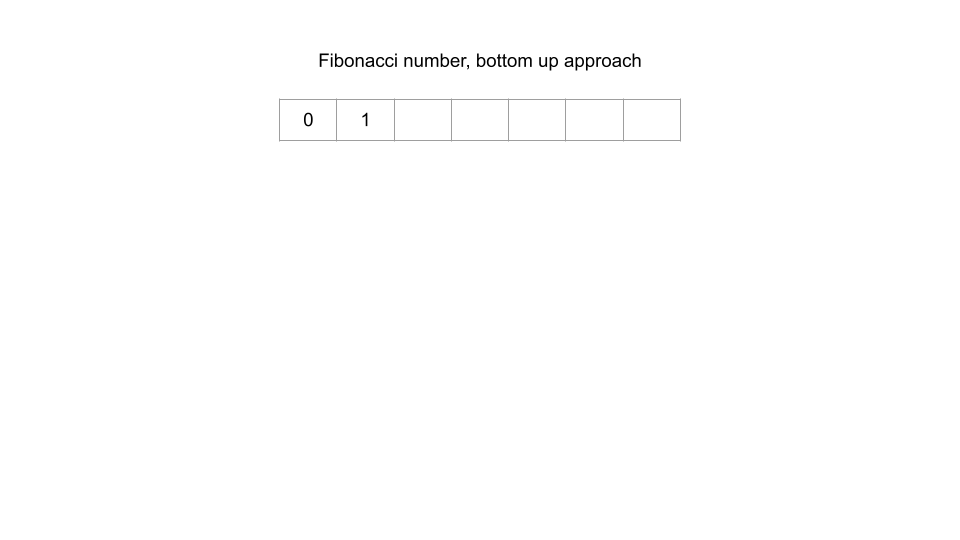
When it comes to reducing state variables, it's hard to give any general advice or blueprint. The best advice is to try and think if any of the state variables are related to each other, and if an equation can be created among them. If a problem does not require iteration, there is usually some form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

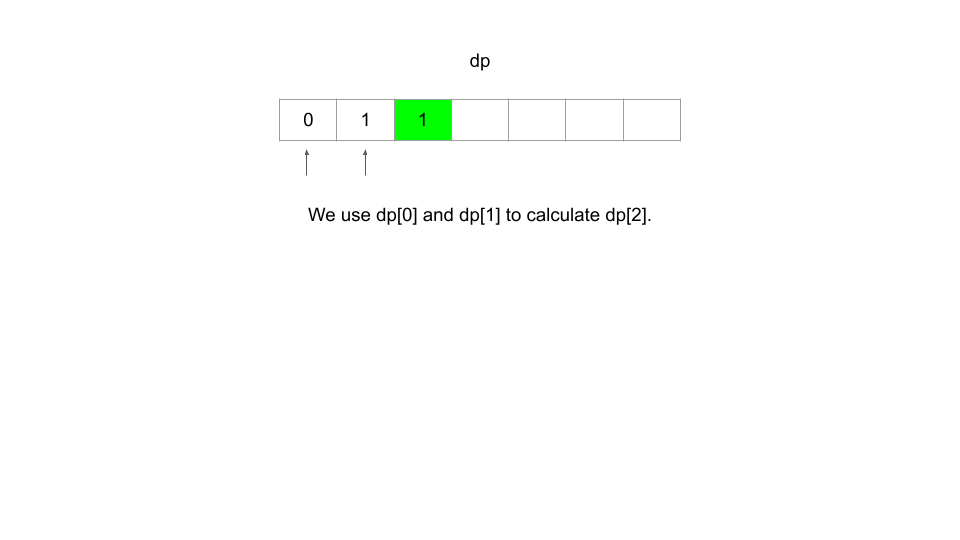
Another common scenario where we can improve space complexity is when the recurrence relation is static (no iteration) along one dimension. Let's look back at where we started - [Fibonacci](https://leetcode.com/problems/fibonacci-number/). Recall that the  *ith* Fibonacci number can be calculated with the recurrence relation:

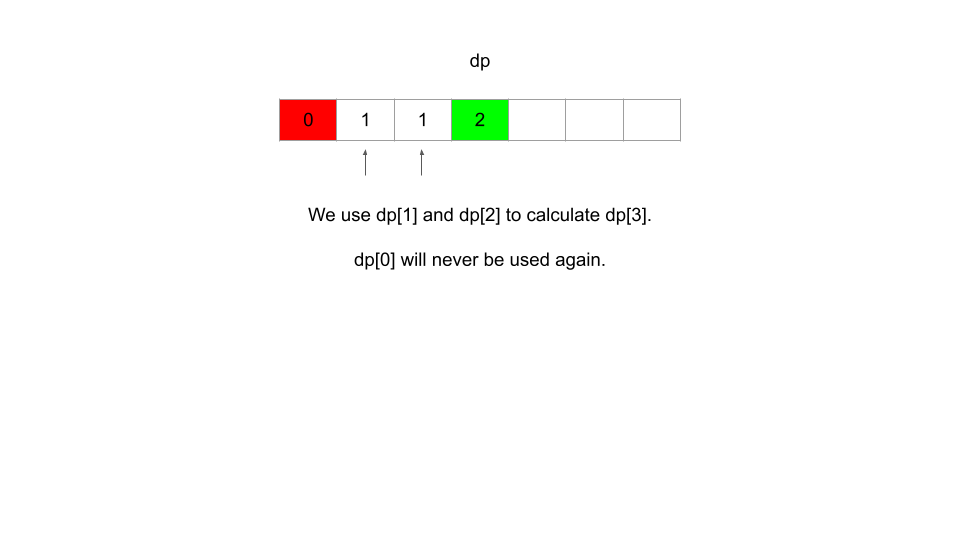
*F*(*i*)=*F*(*i*−1)+*F*(*i*−2)

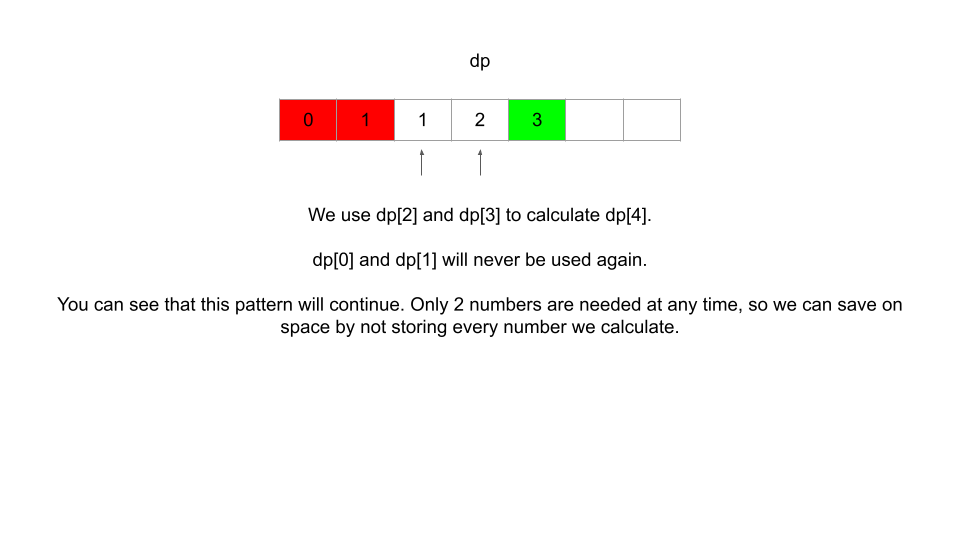
Because this recurrence relation is static, to calculate the *ith* Fibonacci number, we only ever care about the previous two numbers. That means if we are using a bottom-up approach to find the *nth* Fibonacci number and start from the base cases, we don't actually need to use an array and remember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









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.

**class** **Solution**:

**def** **fib**(self, n: *int*) -> *int*:

        if n <= **1**: return n

        one\_back = **1**

        two\_back = **0**

        for i in range(**2**, n + **1**):

            temp = one\_back

            one\_back += two\_back

            two\_back = temp

        return one\_bac

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.

**Kadane's Algorithm**

[Kadane's Algorithm](https://en.wikipedia.org/wiki/Maximum_subarray_problem#Kadane's_algorithm) is an algorithm that can find the [maximum sum subarray](https://leetcode.com/problems/maximum-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

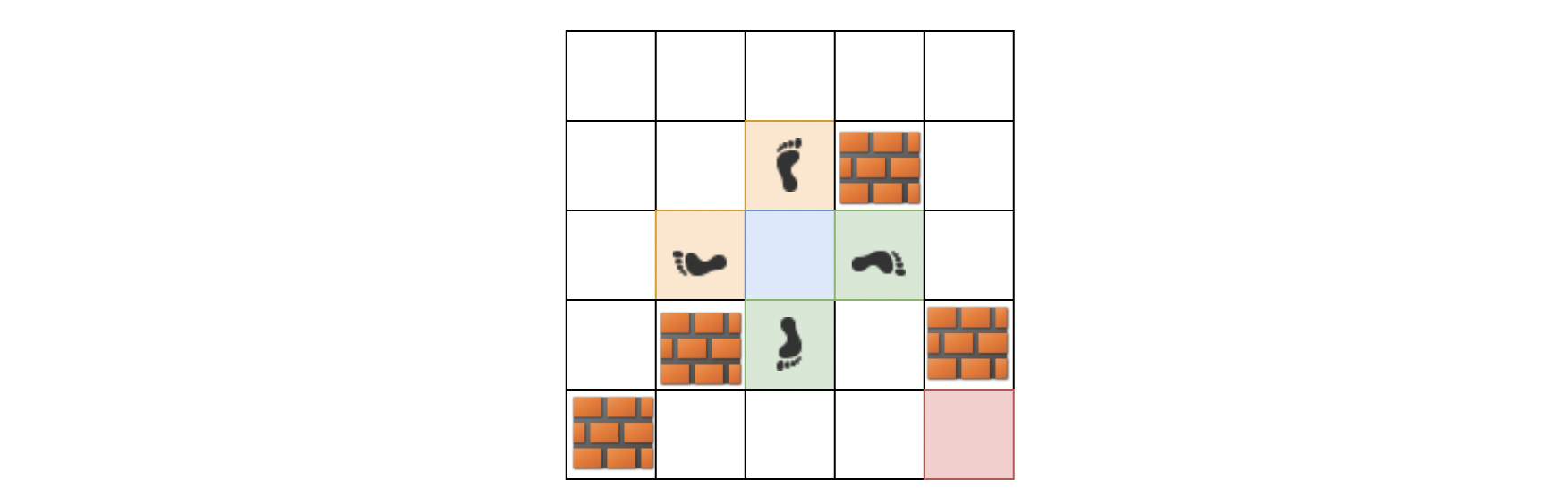
Line 3.1 of the pseudocode is where the magic happens. If current has become less than 0 from including too many or too large negative numbers, the algorithm "throws it away" and resets.

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).

**DP for Paths in a Matrix**

**Pathing Problems**

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.