What is Dynamic Programming

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:

- The problem can be broken down into “overlapping sub-problems” - smaller versions of the original problem that are re-used multiple times.
- The problem has an “optimal substructure” - an optimal solution can be formed from optimal solutions to the overlapping sub-problems of the original problem.

The Fibonacci Sequence is a classic example used to explain DP. 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 we want to find the n^th^ Fibonacci number $F(n)$, we can break it down into smaller sub-problems - find $F(n-1)$ and $F(n-2)$ instead. Then, adding the solutions to these sub-problems 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 sub-problems. These sub-problems are also overlapping - for example, we would need $F(4)$ to calculate both $F(5)$ and $F(6)$.

Although greedy problems have optimal substructure, they do not overlapping sub-problems. Divide and conquer algorithms break a problem into sub-problems, but these sub-problems are not overlapping.

Dynamic programming is a powerful tool because it can break a complex problem into manageable sub-problems, avoid unnecessary recalculation of overlapping sub-problems, and use the results of those sub-problems 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 because it reuses the results of sub-problems rather than recalculating the results for previously seen sub-problems.

Top-down and Bottom-up

There are two ways to implement a DP algorithm:

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

Bottom-up (Tabulation)

Bottom-up is implemented with iteration and starts at the base case. Let’s use the Fibonacci sequence as an example again. The base case 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 $F(3)$ and so on all the way up to $F(n)$.

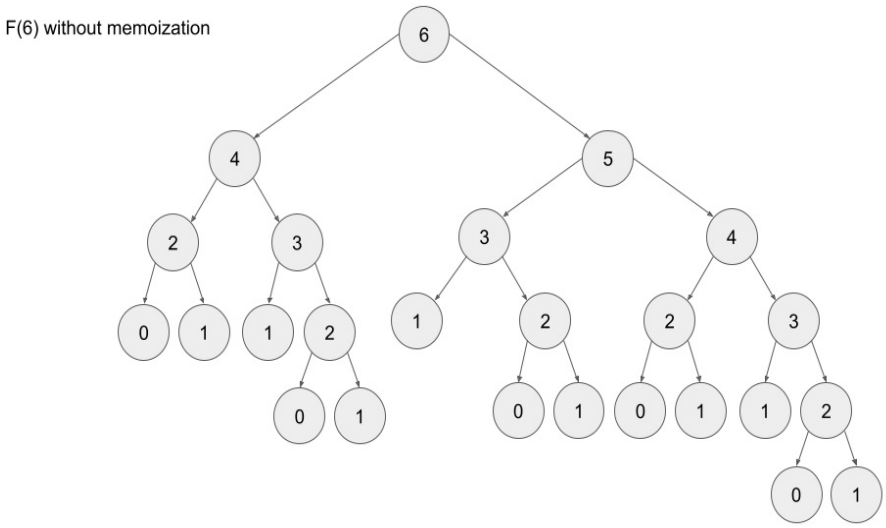
```
// Pseudocode example for bottom-up

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]
```

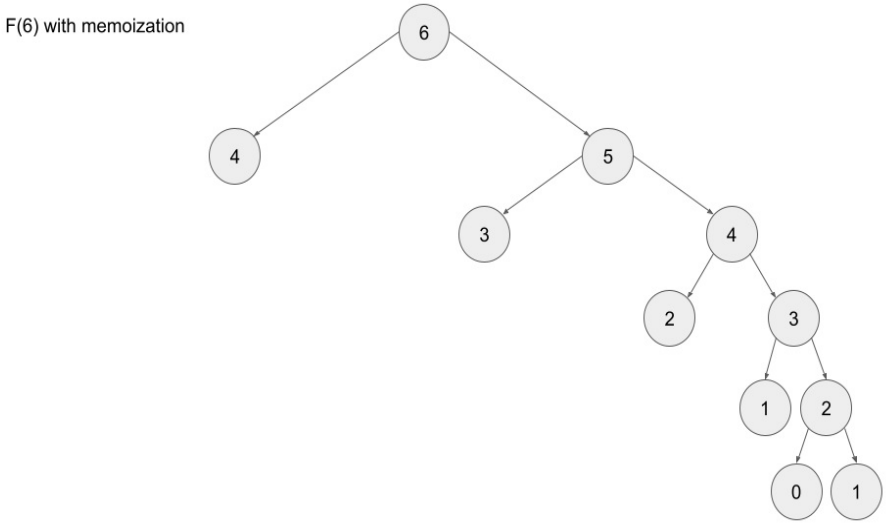
Top-down (Memoization)

Top-down is implemented with recursion and made efficient with memoization. If we wanted to find n^th^ Fibonaci $F(N)$, we could find a recursive pattern that will continue on until the base cases $F(0) = 0$ and $F(1) = 1$ are reached. The problem with just implementing it recursively is that, there is a ton of unnecessary repeated computation. To solve this, we use memoization technique.

> 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.

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 one 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 sub-problems does not matter, whereas with tabulation, we need to go through a logical ordering of solving sub-problems.

When to use DP

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 `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 `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.