Explore

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 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.**

> 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.