A Systematic Approach to Dynamic Programming

A step-by-step method using memoization and tabulation

Fabian Robaina Follow
Aug 9, 2019 · 15 min read ★

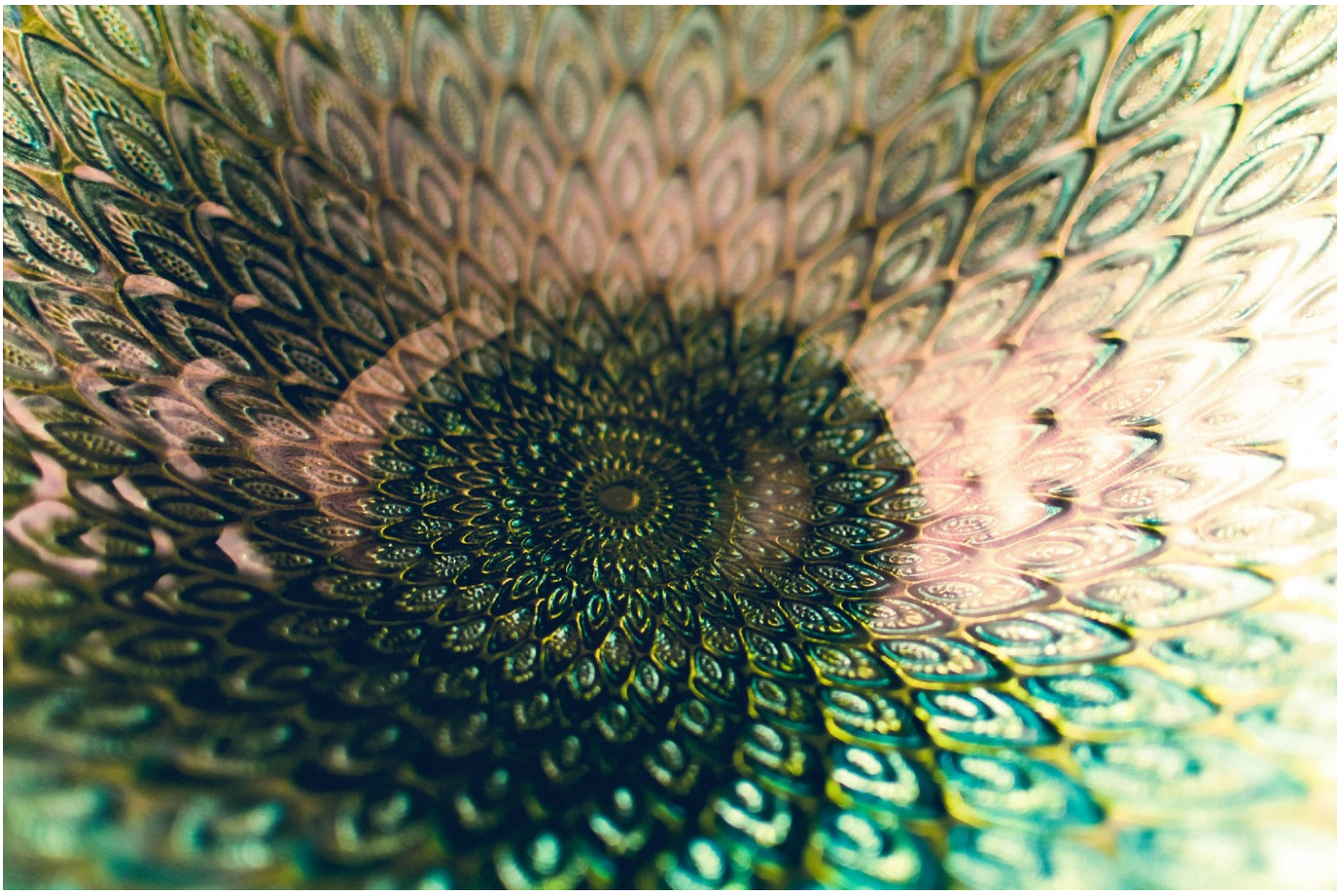


Photo by Ira Mint on Unsplash

Dynamic programming (DP) can be an intimidating concept at first. This problem-solving technique builds on non-intuitive constructs such as recursion, backtracking, and recurrence relations. The good news is that dynamic programming is also really

useful in real-life applications and highly attractive to interviewers when they're evaluating your problem-solving skills.

The process of generating dynamic programming solutions can be approached systematically. Here I want to share a systematic approach I use when solving problems using dynamic programming. I hope you find this useful.

I will initially present the steps I follow when approaching problems using dynamic programming. If you're familiar with the terms described by the steps, you may want to grab a problem and test how this approach works for you. For those of you who aren't familiar, I'll explain what each step is about and why they're arranged the way they are.

From here on, I'll refer to dynamic programming as DP.

. . .

General Steps to Solving Problems Using Dynamic Programming

First of all, ask yourself: Can I solve this problem using DP?

Assuming that the problem has the necessary structure (detailed farther on) to apply DP techniques to it, then:

1. Define the state(s).
2. Define the recurrence relation(s).
3. List all the state(s) transitions with their respective conditions.
4. Define the base case(s).
5. Implement a naive recursive solution.
6. Optimize the recursive solution to caching (memoization).
7. Remove the overhead of recursion with a bottom-up approach (tabulation).

Before we start to go into the steps, a couple of details. First, this piece is a general overview of some of the topics dynamic programming covers. You'll get much more

rigorous explanations on a formal DP course. Second, memorizing DP solutions to specific problems does not help here; learning the approach to come up with the solution does. Finally, I am not a DP expert. These tips are things I found useful as a computer science student while learning to solve problems using DP.

. . .

Can I Solve This Problem Using DP?

First, let's describe what we mean by DP. In short, DP is a problem-solving technique that aims to solve complex problems by first solving smaller instances of those problems exactly once.

DP is widely used to solve problems that relate to optimization. A good trick to see if your problem is a good candidate to apply DP techniques is to find keywords that imply optimization, such as *maximize*, *minimize*, *longest*, or *shortest*.

Problems that are good DP targets are said to have optimal structure and overlapping sub-problems. In an academic setting you may get a much more elaborate description of these terms, but in short:

- To evaluate an optimal structure, ask yourself if you can describe the solution as a recurrence relation. If so, you can find an optimal solution to the problem in the function of the solutions to its sub-problems. An example is the famous Fibonacci Sequence. A Fibonacci number is the sum of the two previous Fibonacci numbers, which translates to this recurrence relation: Fib(n) = Fib(n-1) + Fib(n-2).
- To evaluate overlapping sub-problems, ask yourself if, while solving the problem, you find your program solving the same sub-problem multiple times. If so, then you have overlapping sub-problems.

The takeaway is this: If you find hints in the problem description that imply optimization, evaluate whether the problem has an optimal structure and find overlapping sub-problems. Then DP may be a path to consider while solving the problem.

. . .

Recursion and Backtracking Are Important

Recursion is a fundamental technique when creating DP solutions. In recursion, you have a function that calls itself until some base case is reached. These functions are said to be recursive.

```
def fib(n: int) -> int:

    # Base Cases
    if n == 0:
        return 0
    if n == 1:
        return 1

    # Recursive Calls
    return fib(n-1) + fib(n-2)
```

Example of a recursive function

Backtracking is a technique that uses recursion to solve exhaustive search problems. In these problems, we must evaluate multiple possibilities before arriving at an optimal solution. Because these problems usually involve permutations or combinations of choices they are also known as combinatorial search problems.

In backtracking, we explore until we find a valid solution or we reach a base case, at which point we return one level up the recursive tree, aka we backtrack.

```
12  def find_permutations(input_collection: list, current_state: list, element_used: [bool], k: int):
13      assert k <= len(input_collection)
14
15      # If the current sate has the length of the expected permutation output. Print it.
16      if len(state) == k:
17          print(state)
18          # Prevents the code from computing permutations of length > k.
19          return
20
21      # Iterates through all elements in the collection
22      # If the element is not used already (no repetition) append it to the state.
23      for (i, element) in enumerate(input_collection):
24          if not element_used[i]:
25              # Append the element to the state array.
```

```
26              current_state.append(element)
27              # Modify the array 'used' for posterior recursive calls so 'element' is marked as used.
28              element_used[i] = True
29              # Find reminding permutations where 'element' is already used in the position where it was just appended.
30              find_permutations(input_collection, current_state, used, k)
31              # All permutations with 'element' at this appended position were already computed. Pop it.
32              current_state.pop()
33              # Mark 'element' as not used so that it can be placed on another position by another branch of recursive
34              # calls.
35              element_used[i] = False
36
37
38      # Input collection.
39      collection = list([1, 2, 3])
40      # Initial empty state.
41      state = list([])
42      # Initial state all elements are not used.
43      used = [0] * len(collection)
44
45      find_permutations(collection, state, used, 3)
```

Finding all permutations of size k of a given collection of integers

The code above prints all the permutations of size k of a given collection. Notice that the recursive calls happen at line 30. Before that, we modify the `current_state` list by appending the current element and marking it as used.

After line 30, which is when the execution is back to the current recursive call, and before we backtrack to the previous call, we make sure the element is removed from the current position and marked as not used so that it can be placed somewhere else by later recursive calls.

At first, that may seem confusing, but don't worry about that code now. The point of showing the code here is to show that backtracking often requires some cleaning to be done before we actually backtrack to previous recursive calls.

Take time to practice recursion and backtracking. Practicing is the only way to get comfortable with the idea of functions calling themselves. Once you're comfortable with it, you start to see recursive solutions everywhere. At that point, you have new problems.

. . .

Approaches to DP

The two main approaches to dynamic programming are memoization (the top-down approach) and tabulation (the bottom-up approach).

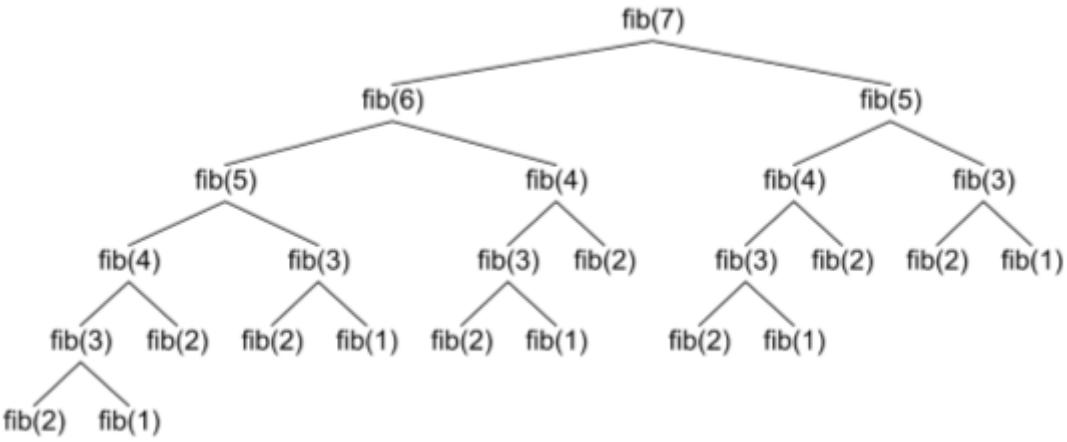
So far we've seen that recursion and backtracking are important when applying the DP premise of breaking a complex problem into smaller instances of itself. However, none of the code snippets above classify as DP solutions even though they use recursion and backtracking.

For a naive recursive solution to apply as a DP solution it should optimize to caching the results of computed sub-problems. In the short definition of DP above, the emphasis is on solving smaller instances only once — with a strong emphasis on "only once."

Memoization

Memoization = Recursion + Caching

Our framework for a dynamic-programming-worthy problem said that it usually contains overlapping sub-problems. Remember the Fibonacci code above? If we create a recursive tree to compute the seventh Fibonacci number we get this:



Source: Fibonacci recursion tree

Notice how many times we solve the same sub-problem. For example, Fib(3) is computed five times, and every fib(3) call recursively calls two more fib. That's ten function calls solving the same fib(3) sub-problem.

Now we start talking DP! Instead of solving the same problem multiple times, why don't we solve it just once and store it on some data structure in case we need it later? That is memoization!

```
def fib_with_memoization(n: int, memo: [int]) -> int:
```

```
    # Base cases
    if n == 0:
        return 0
    if n == 1:
        return 1
    # Check if sub-problem was already solved
    if memo[n] != 0:
        return memo[n]
    # Solve sub-problem and cache it.
    memo[n] = fib(n - 1) + fib(n - 2)
    return memo[n]
```

Fib code optimized to caching.

This approach is the easiest of the two DP approaches presented here. Once you can get a recursive solution to the problem, just make sure you cache the solutions to the sub-problems. Before you make recursive calls to solve a sub-problem, check if it was already solved. Notice that here we do a trade: To archive time efficiency, we're willing to give up memory space to allocate all computed sub-problems' solutions.

Dynamic programming usually trades memory space for time efficiency.

When caching your solved sub-problems you can use an array if the solution to the problem depends only on one state. For example, in the fib code above, the solution to a sub-problem is the 'nth Fibonacci' number. We can use n as an index on a 1D array, where its value represents the solution to the fib(n) sub-problem.

Sometimes the solution to the problem may depend on two states. In this case, you can cache the results using a 2D array, where columns represent one state and rows represent the other. For example, in the famous Knapsack problem (which we'll explore later) we want to optimize for total value, given a maximum weight constraint and a list of items. A knapsack sub-problem may look like this: KS(W, i) → (Max value), where we interpret it as: "What is the maximum value I can get with a weight 'W' and considering the 'ith' item?." Therefore if we want to cache this solution, we need to take both states into account, and that can be accomplished using a 2D array.

Memoization is great — we have the elegance of a problem described recursively, and we're solving overlapping sub-problems only once. Well, not everything is that great. We're still making a bunch of recursive calls. Recursion is expensive both on processor time and memory space. Most recursive functions will consume call stack memory linearly with the number of recursive calls needed to complete the task.

There are special types of recursive functions, known as tail-recursion, that don't necessarily increase the call stack linearly if optimized correctly. These can execute on a constant call stack space. Without going into many details, tail-recursive functions perform the recursive call at the end of its execution, meaning that its stack frame is useless thereafter. The same stack memory space can be reused to hold the state for the next recursive call. The problem that arises is in dealing with the return address. We want to make sure that after the recursive tree ends, you return to the instruction that started the series of recursive calls. Feel free to do some research on this topic.

Recursive functions always carry the weight of potential stack overflows issues. The following is a Python command to check the recursion depth limit. If I try to use Python with recursion to solve a problem whose solution involves a recursive depth of more than 1000 calls, I'll get a stack overflow exception. That quantity can be increased, but we get into language-specific topics.

```
>>> import sys
>>> sys.getrecursionlimit()
1000
```

Recursion depth limit in Python

In defense of recursive programming, we must say that recursive functions are often easier to formally prove. Recursive functions provide you with the same repetitive behavior of raw loops but without in-block state changes, which is a common source of bugs. States in recursive programming are updated by passing new parameters to new recursive calls, instead of being modified as the loop progresses.

Tabulation

Tabulation aims to solve the same kind of problems but completely removes recursion. Removing recursion means we don't have to worry about stack overflow issues, as well

as the common overhead of recursive functions. In the tabulation approach to DP (also known as the table-filling method) we solve all sub-problems and store their results on a matrix. these results are then used to solve larger problems that depend on the previously computed results. Because of this, the tabulation approach is also known as a bottom-up approach. It's like starting at the lower level of your recursive tree and moving your way up.

Tabulation can be much more counterintuitive than recursive-plus-cached memoization solutions. But it's also much more efficient in terms of time complexity and space complexity if we take into account the call stack memory which increases linearly with the number of recursive calls — again, assuming it's not tail-recursion optimized.

If you go back to the steps initially presented in this piece, you'll find that tabulation is the last step in the systematic approach to DP. This is because it's easier to get to a tabulation solution by first solving the problem with recursion and backtracking, then optimizing it to caching with memoization techniques, if necessary, and finally making a few adjustments to update it to a final bottom-up solution. Later you will see a few tricks to achieve that. But first, let's see how tabulation works.

We've been talking about states a lot, but we still do not have a formal definition of what we mean by states on our DP context. What I understand by states are parameters that affect the outcome of a recursive call. States are what differentiate one call from another and allow us to explore different choices and get an optimal result. We'll get some practice defining states at the end of this article.

Since tabulation proposes a bottom-up approach, we must first solve all sub-problems that a larger problem may depend on. If we don't solve the smaller problem, we can't move on to solve the larger one. In tabulation, we use one for-loop for every state of the problem. But where do I make it start? Where do I make it end? To answer that, let's explore the following recurrence relation.

Let's say we have a function that solves some optimization problem — call it optimal or OP for short. And let's assume that the nature of the problem makes OP(n) depend on OP(n-1), so OP(n) = OP(n-1).

This recurrence relation is telling you that you cannot know what OP(n) is if you don't know what OP(n-1) is. That means we need to start at the lowest value of n, say 0, and solve every sub-problem all the way to n. That's the trick: If your recurrence relation shows that your states are decreasing, then your loops should be increased so you compute every sub-problem that larger problems depend on. Remember *bottom-up*.

This will become clear when we apply all the strategies learned to real problems. And guess what? That will start now.

. . .

The Knapsack Problem

We'll start applying our systematic approach to DP on the famous 0/1 Knapsack problem. Our task is the following: We are given a bag with a discrete number of items, where each item has a value and a weight. We want to grab items such that we maximize the value we take home. But here's the condition: We can only carry a limited amount of weight with us. So the given is n items with values of [v0,v2,v3,…,v(n-1)] and weights [w0,w1,w2,…w(n-1)] respectively, and a weight limit (W). What is the maximum value we can take home?

Should I use DP to solve this problem?

Did you notice? The problem asks us to maximize the result of some problem based on conditions. I can smell optimization already. But does it have an optimal structure?

Yes, the solution to the problem of how much value I can get with items [0, n] and their respective values [v0, …,v(n-1)] and weights [w0, …, w(n-1)] and a total weight limit W can be broken down into the maximum value I can get by either adding the nth item and evaluating the optimal outcome I get with the remaining weight and the remaining items, or not adding it, and studying the optimal result I get with the same weight and the remaining items.

Wait, what? I know, lots of words but mathematics will explain that better when we define the recurrence relation and state transitions.

Step 1: Define the states

What are the parameters that will affect the outcome of our recursive calls? In the problem, we have two factors that affect our results: the weight limit and the item that we're considering whether to add to our bag or not. The state of a recursive call will be defined by its available weight W and the index of the item being considered at the moment 'i'.

Steps 2 and 3: Define the recurrence relation and examine the state transitions

The cost function we're trying to optimize will be KS(W, i), which should give us the maximum weight that we can get with the items [0, i] and remaining weight W.

We have two choices: We can get the item or not. If we get the item, we affect the state of the next recursive call because it will have less available weight and one less item to consider adding. We only consider adding an item if it fits. We compress all that information mathematically as follows:

$KS(W, i) = MAX(val[i] + KS(W- w[i], i-1), KS(W, i-1))$ **if** $W \geq w[i]$

$KS(W, i) = KS(W, i-1)$ else

Step 4: Define the base cases

When should we stop considering options? One stopping condition will be if we have no more space ($W == 0$), another will be if we have no more items to consider ($i == -1$). Both base cases add no value to us so:

return 0 if $W == 0$ or $i == -1$

Step 5: Implement a naive recursive solution

We have all it takes to come up with our recursive solution. We have a recurrence relation, we have defined the state transitions, and we have base cases.

```
# 0/1 Knapsack problem
def KS(W: int, i: int):
    global items

    # Base Cases.
    if W == 0 or i == -1: return 0

    value_item_added, value_item_not_added = 0, 0
```

```
    # State transitions:
    # Evaluating result of adding the ith item.
    if W >= items[i][1]:
        #                 |value of i|         |weight of i|
        value_item_added = items[i][0] + KS(W - items[i][1], i - 1)

    # Evaluating result of not adding the ith item.
    value_item_not_added = KS(W, i - 1)

    # Optimizing the cost function.
    return max(value_item_added, value_item_not_added)


# Items are represented as a collection of tuples (value, weight)
items = [(5, 5), (2, 1), (3, 2), (10, 5)]
# Weight limit
W_limit = 12

print(KS(W_limit, len(items) - 1))
```

Recursive solution to the Knapsack problem. **Result = 18**

Step 6: Optimize the recursive solution to caching (memoization)

In my experience, it is sometimes hard to see if the problem has overlapping sub-problems without going through an example of the problem being solved recursively. This can be done by running your code or simply drawing on paper the sequence of recursive calls. Let's update our code to cache computed sub-problems.

```
# 0/1 Knapsack problem
def KS(W: int, i: int, memo: [[int]]):
    global items

    # Base Cases.
    if W == 0 or i == -1: return 0

    # Checked if sub-problem was already solved.
    if memo[W][i] != 0:
        return memo[W][i]

    value_item_added, value_item_not_added = 0, 0

    # State transitions:
    # Evaluating result of adding the ith item.
    if W >= items[i][1]:
```

```
    #                    |value of i|          |weight of i|
    value_item_added = items[i][0] + KS(W - items[i][1], i - 1, memo)

# Evaluating result of not adding the ith item.
value_item_not_added = KS(W, i - 1, memo)

# Stored the result of this problem solved.
memo[W][i] = max(value_item_added, value_item_not_added)
return memo[W][i]
```

Memoization = Recursion + Caching.

Step 7: Remove the overhead of recursion with a bottom-up approach (tabulation)

As we discussed earlier, it's easy to optimize to a tabulation solution once we're at this stage. There will only be a few changes to our code.

Instead of:

KS(W, i) = MAX(val[i] + KS(W- w[i], i-1), KS(W, i-1)) if W ≥ w[i]

KS(W, i) = KS(W, i-1) else

we have:

KS[W][i] = MAX(val[i] + KS[W-w[i]][i-1], KS[W][i-1]) if W ≥ w[i]

KS[W][i] = KS[W][i-1] else

We have modified our recurrence relation to matrix notation since we are now filling a matrix.

Remember the two important points in tabulation:

- One for-loop for each state.
- If your states decrease in value then your loops increase in value, and vice-versa.

In this case, we have two states and both of them decrease in value as described in the recurrence relation. The updated code looks like this:

```
# 0/1 Knapsack problem
```

```python
def KS(W: int, items: [(int, int)]) -> int:

    # Initialize the tabulation table.
    # There are len(items)+1 columns and W+1 rows all initialized to 0.
    ks = [[0 for col in range(len(items) + 1)] for row in range(W + 1)]

    # Since both w = 0 and i = 0 will be dummy row/col both
    # loops start at 1.
    for w in range(1, W + 1):
        for i in range(1, len(items) + 1):
            value_item_added, value_item_not_added = 0, 0

            # State transitions:
            # Evaluating result of adding the ith item.
            if w >= items[i - 1][1]:
                #                   |value of i|             |weight of i|
                value_item_added = items[i - 1][0] + ks[w - items[i - 1][1]][i - 1]

            # Evaluating result by not adding the ith item.
            value_item_not_added = ks[w][i - 1]

            # Stored the optimal result of this sub-problem.
            ks[w][i] = max(value_item_added, value_item_not_added)

    return ks[W][len(items)]
```

0/1 Knapsack Tabulation solution.

Notice the things that changed:

- Our function is no longer recursive. We don't need to pass modified states around. KS now receives the weight limit and the collection of items and returns the optimal result. In this case, we can say that KS is now purely functional.
- In our tabulation approach, base cases are hard-coded into the table. Sometimes this requires us to consider adding dummy rows/columns. If you see our base case for i, it was i==-1. However, on a matrix, we cannot have a negative index. To solve this, we do an index shifting. What used to be (i==-1) will now be (i==0) so there we'll store the values of the base case.
- Let's take a moment to analyze index shifting further because it's a very common practice. Our recurrence relation shows:

KS[W][i] = MAX(val[i] + KS[W-w[i]][i-1], KS[W][i-1]) if W ≥ w[i]

KS[W][i] = KS[W][i-1] else

Notice the [i-1]. If we start our loop at i = 0, that will result in an out-of-bound exception. But we said that i = 0 and W = 0 are our base cases. There's no reason to compute them, so let's start at 1. However, we need to modify how we refer to val[i] and w[i] since now (i = 1) should map to the first element (that at position 0). We simply balance the effect of the index shifting by using (i-1) whenever we want to index an element in our input collection.

- Notice that the only major changes were the removal of return statements use for recursion, and changing all our recursive notation to matrix form. We wrapped the same logic for state transitions inside loops that we easily know how to generate.
- There is only one return statement left in our function, it gives us the final result to our original problem. That is it!

. . .

Most Important Things to Remember

I know this has been a lot of information. Let's try to highlight some of the most important details:

- DP is a problem-solving technique that aims to solve complex problems by first solving smaller instances of those problems exactly once.
- Problems that are good DP targets are those that have optimal structure and overlapping sub-problems.
- The two main approaches to dynamic programming are memoization (top-down approach) and tabulation (bottom-up approach).
- Memoization = Recursion + Caching
- Recursion is expensive both in processor time and memory space.
- In the tabulation approach to DP, we solve all sub-problems and store their results on a matrix. then we continue to solve the larger problems that depend on these computed sub-problems.

- Practice, Practice, Practice.

Thanks for reading.

Programming Dynamic Programming Computer Science Python