Dynamic Programming: Introduction

Let's go over the Dynamic Programming pattern, its real-world applications, and some problems we can solve with it.

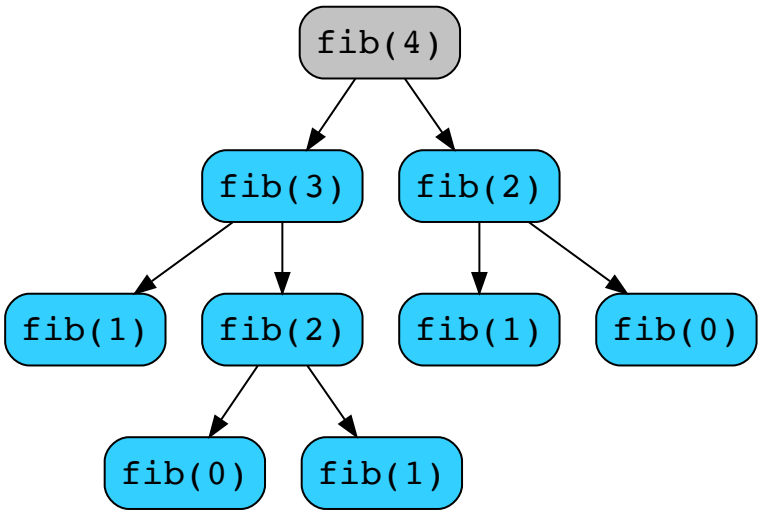
We'll cover the following

Overview

Many computational problems are solved using a divide-and-conquer approach recursively. In these problems, we see an *optimal substructure*, i.e., the solution to a smaller problem helps us solve the bigger one. For example, is the string "rotator" a palindrome? We can start by observing that the first and the last characters match, so the string *might* be a palindrome. We shave off these two characters and try to answer the same question for the smaller string "otato". We keep doing this until we are able to say a definite *yes* or *no*.

In some of the problems that can be solved with the above approach, there are many overlapping sub-problems. That is, we find ourselves solving the same sub-problem over and over. The recursive solution to the *palindrome* problem does not have this characteristic, but solutions to many other problems do have overlapping sub-problems. An example is

the recursive computation of the nth Fibonacci number. Here's the recursion tree for the solution to this problem with n = 4.



We can see above that $fib(2)$ is evaluated twice, $fib(1)$ is evaluated thrice, and $fib(0)$ is evaluated twice. These are repeated overlapping subproblems. We can use the dynamic programming pattern to save the result once and use it wherever the subproblem is repeated.

We've discussed that we need to save the computations, but how can we save and use them? There are two approaches in dynamic programming that we can use to solve the problems:

- Top-down approach with memoization
- Bottom-up approach with tabulation

Top-down approach with memoization

In the top-down approach, we store all the results of solved subproblems in an array or hash map. We use the results of these stored subproblems wherever we encounter an overlapping subproblem during the recursion.

In the Fibonacci problem tree shown earlier in the lesson, we calculate **fib(2)** for the first time in the left sub-tree of the root node. At that time, we store this value. We simply look it up later, rather than recursively re compute it, in the right sub-tree of the root node.

Bottom-up approach with tabulation

The bottom-up approach allows us to not use recursion as we used it in the top-down approach. It uses an n-dimensional array to store the results of solved subproblems. We use the results of these stored subproblems wherever we encounter an overlapping subproblem during the iteration of the array.

For the Fibonacci example shown above, we would create an array of size $n + 1$ in which the ith value will eventually hold the ith Fibonacci number. We initialize the indices 0 and 1 with the values 0 and 1, respectively. Then, we compute the values for indices 2, 3, ..., n as the sum of the values at the previous two indices.

The bottom-up approach on some problems requires computing two-dimensional arrays. Often, we can optimize the space complexity in such problems by using a one-dimensional array. The idea is to use the same array for the previous and the next iteration. But this optimized approach is a bit tricky and might not be applicable to all dynamic programming pattern problems.

Examples

The following examples illustrate some problems that can be solved with this approach:

Subarray sum equals K

Given an array of integers and an integer k, return the total number of subarrays whose sum equals to k.

To solve this problem, we can use the bottom-up dynamic programming approach. To fill the 2-D array, we can use the following formula:
dp[index][sum] = dp[index-1][sum] + dp[index-1][sum-num[index]],
where sum ranges from 0 - K.

num	[1, 1, 2, 3]

K	4

K →

i	num[i]	0	1	2	3	4
0	1	1	1	0	0	0
1	1	1	2	1	0	0
2	2	1	2	2	2	1
3	3	1	2	2	3	3

Total subsets: 3

Does my problem match this pattern?

- Yes, if the problem exhibits both of these characteristics:
  - Overlapping subproblems, that is, we can use the results of one subproblem when solving another, possibly larger subproblem.
  - Optimal substructure, that is, if the final solution can be constructed from the optimal solutions to its subproblems.
- No, if either of these conditions is fulfilled:
  - The problem has non-overlapping subproblems.
  - The optimal substructure property is violated.

Many problems in the real world use the dynamic programming pattern

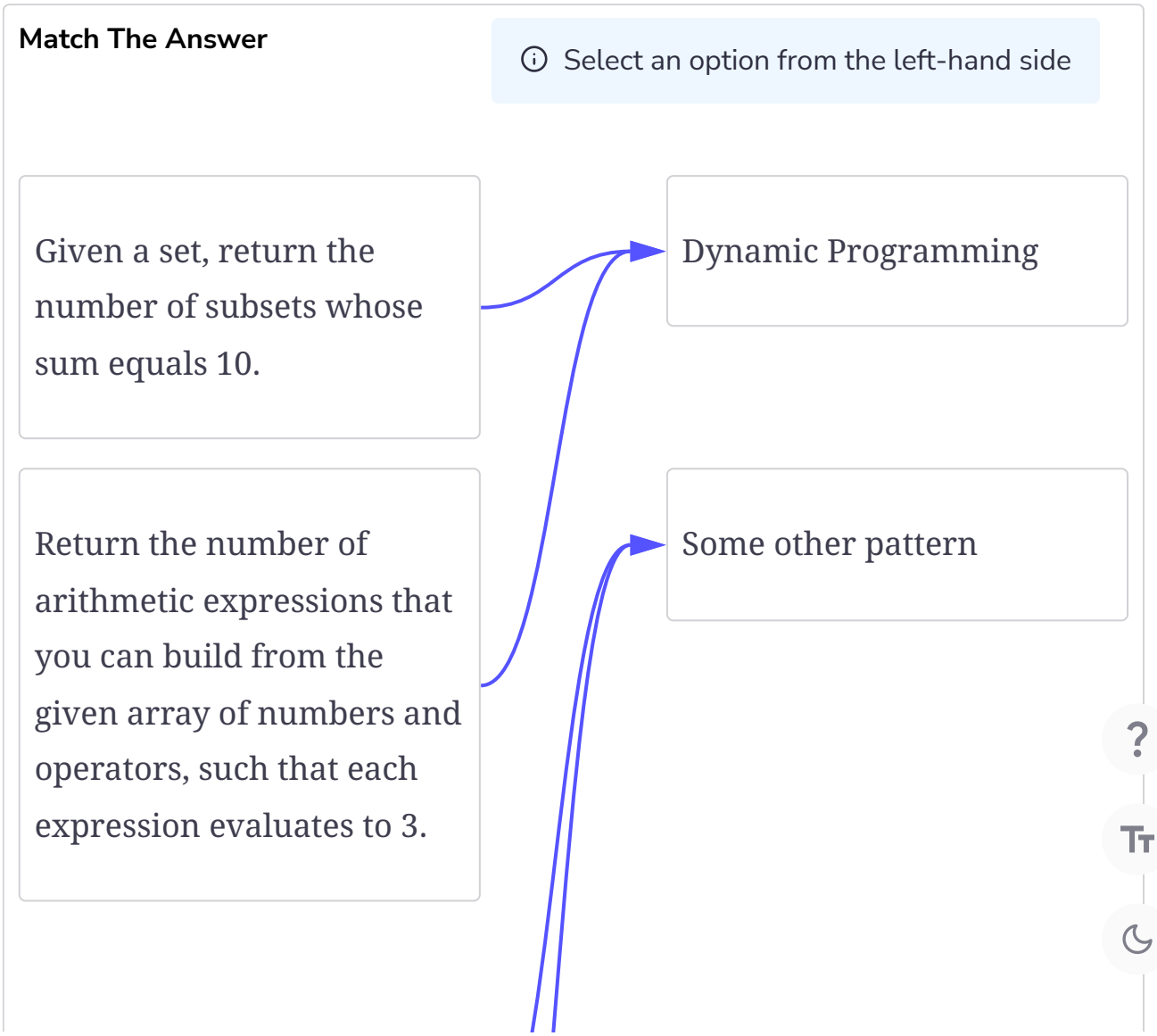
Let's look at some examples.

- **Load Balancer:** Find the optimal way to handle a given workload by using servers with different workload handling capacities.
- **Search Engine:** Check if white spaces can be added to a query to create valid words in case the original query does not get any hits on the web. Find all the possible queries that can be created by adding white spaces to the original query.

Strategy time!

Match the problems that can be solved using the dynamic programming pattern.

> **Note:** Select a problem in the left-hand column by clicking it, and then click one of the two options in the right-hand column.



Find all the ways to safely place 5 queens on a 5×5 chessboard.

Maximize your heist without robbing neighboring houses.

Reset | Show Solution | Submit