Dynamic Programming

By:

Dustin Smith

Let's discuss dynamic programming. What is dynamic programming? The content of this right up comes from Dynamic Programming–Learn to Solve Algorithmic Problems & Coding Challenges

Definition:

Dynamic Programming is both a mathematical optimization method and computer programming method.

With dynamic programming, we can take two approaches memoization and tabulation. We will start with examining brute force methods and how to apply memoization. First, consider the Fibonacci series. Let's refresh. The Fibonacci numbers are $0, 1, 1, 2, 3, 5, 8, \ldots$. We can define the Fibonacci numbers as

$$\begin{aligned} F_n &= F_{n-2} + F_{n-1} \\ F_0 &= 0 \\ F_1 &= 1 \end{aligned}$$

Let's look at the classic recursion method for the Fibonacci series.

```
def fin(n: int) -> int:
  if n < 2:
    return 1
  return fib(n - 2) + fib(n - 1)
```

What would be the steps for `fib(7)`? What would be the time complexity of the brute force algorithm?

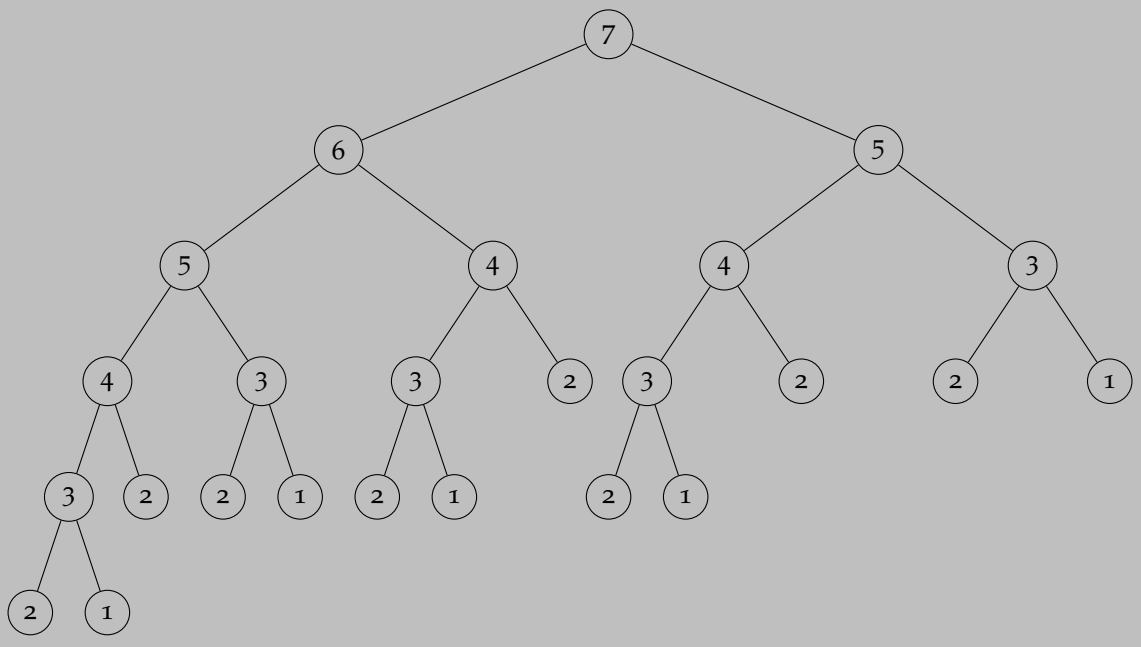

Figure 1: Steps for the binary tree of `fib(7)`.

We can see that each node has two children and the height of the tree is $n = 7$. That is, we have 2^n steps with brute force so $\theta(2^n)$, exponential time complexity. We can generalize the brute force binary steps to be $\theta(m^n)$ where n is the height of tree and m is the number of elements per child. What would be the space the complexity? Whenever we get to a leaf, we reach the end of the stack. In order to get the next leaf, we must pop stack. Therefore, we use n stack calls which is the height of the tree, $\theta(n)$, linear

space complexity. From figure 1, we see we have overlapping sub problems, namely $\texttt{fib}(5)$, $\texttt{fib}(4)$, and $\texttt{fib}(3)$. This is dynamic programming when we can decompose a problem into smaller instances of the same sub problems.

Let's implement our Fibonacci function but now with memoization.

```
def fib_memo(n: int, memo: Dict[int, int]=None) -> int:
  if memo is None:
    memo = {}
  if n in memo:
    return memo[n]
  if n < 2:
    return 1

  memo[n] = fib_memo(n - 2, memo) + fib_memo(n - 1, memo)
  return memo[n]
```

How does our binary tree change in the case of memoization? We now have roughly $2 \cdot n$ nodes. That

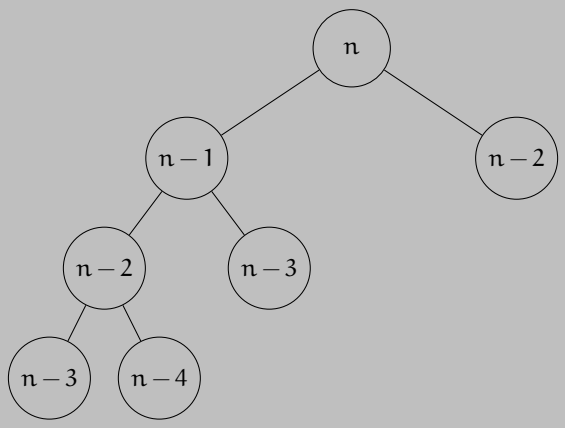

Figure 2: Steps for the binary tree of $\texttt{fib}(n)$ memoization.

is, our time complexity is now linear of $\theta(n)$ with space complexity of $\theta(n)$ for the dictionary. We were able to go from exponential to linear.

Let's no look at a traveler on a 2D grid. We begin in the top-left corner and end in the bottom-right corner. We can only move down or to the right. In how many ways, can we travel to the goal on a grid with dimensions $m \times n$? First, we will consider some base cases. If either m or n are zero, then we don't have a grid so we would have zero ways. What if our grid was 1×1? There is only one way since the start is the finish. We can represent our grid traveler problem as tree where each node is the coordinate pair of position and root of the tree would be the size of the grid. The leaf nodes figure 3 are related to our base cases of 0 for a grid with with a zero column or row and 1 for a 1×1 square grid. Starting from the leafs, we can label the number of ways with either $+0$ or $+1$ for each node. The parent nodes will be the sum of the number of ways of the children up to the root. I have left of the leafs that already have a 0 for an m or n. However, we can see the height of the tree would be $m + n$. Each node will have at most two children so $\theta(2^{m+n})$.

```
def grid_traveler(m: int, n: int) -> int:
  if m == 1 and n == 1:
    return 1
  if m == 0 or n == 0:
```

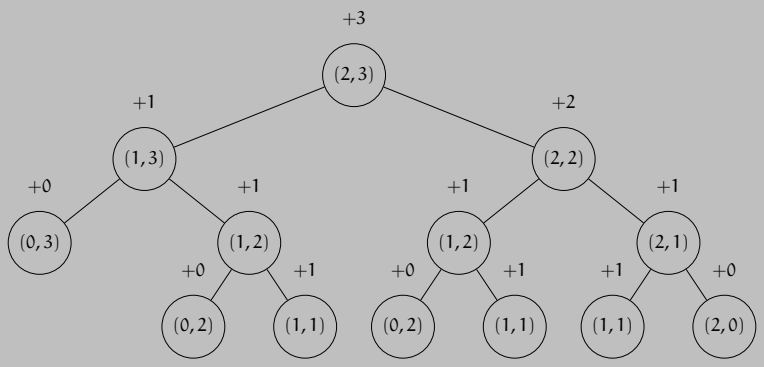

Figure 3: Binary tree for our grid traveler problem of a 2×3 grid.

```
    return 0

  return grid_traveler(m - 1, n) + grid_traveler(m, n - 1)
```

For space complexity, the stack would at most have $m + n$ calls so $\theta(n + m)$ is our complexity.

Now, let's implement the grid traveling problem using memoization.

```
def grid_traveler_memo(m: int, n: int, memo: Dict[str, int]=None) -> int:
  if memo is None:
    memo = {}

  key = f"{m}, {n}"
  if key in memo:
    return memo[key]

  if m == 1 and n == 1:
    return 1
  if m == 0 or n == 0:
    return 0

  memo[key] = grid_traveler_memo(m - 1, n, memo) + \
    grid_traveler_memo(m, n - 1, memo)
  return memo[key]
```

What would our time and space complexities be? We would have $m \cdot n$ combinations now as opposed to 2^{m+n} steps. Thus, our new time complexity $\theta(m \cdot n)$. Our space complexity would still be $\theta(m + n)$. We have been able to improve greatly from exponential time complexity.

Now that we have seen a few examples of turning a recursive brute force algorithm into a the dynamic programming variant of memoization, we should discuss some guidelines for using memoization.

1. Determine a recursive solution.
2. Make it efficient.

What does this mean? We need to visualize it as a tree, implement the tree using recursion where the leaves are our base cases, and test it. Once we do this, we can move onto adding a memo, add the base cases to return memo values, and store return values in the memo.

Next, we write a function to determine if we can make the desired the sum given a target value and an array of integers which we can reuse. We will return a boolean if we can or cannot make the desired target value. We can assume $\forall x \in$ array $x \geqslant 0$. Consider the following inputs of a target value of 7 and an array of $[5, 4, 3, 7]$. Let's first visualize our problem using a tree. By now, it should be clear

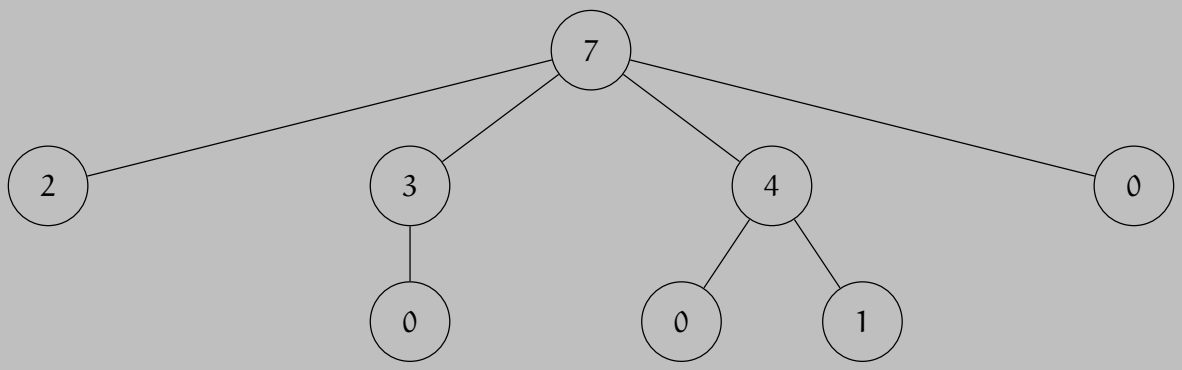

Figure 4: The tree for a target of 7 and an array of $[5, 4, 3, 7]$. The tree is pruned of any leaves that would be less than 0.

what the height and number of children per parent node would be. In this brute force implementation, let m be the height of the tree which corresponds to the target and n be number of children, we have that the time complexity is $\theta(n^m)$. The stack space for this algorithm would be m as well so the space complexity is $\theta(m)$. What would be our base case(s)? If we get a node of zero, we are able to construct the sum, and if our number goes negative, we are not.

```
def can_sum(target: int, arr: List[int]) -> bool:
  if target == 0:
    return True
  if target < 0:
    return False

  for num in arr:
    remain = target - num
    if can_sum(remain, arr):
      return True

  return False
```

Now we will construct the memoization algorithm for can sum.

```
def can_sum_memo(
    target: int,
    arr: List[int],
    memo: Dict[int, bool]=None
    ) -> bool:
  if memo is None:
    memo = {}
  if target in memo:
    return memo[target]

  if target == 0:
```

```
    return True
  if target < 0:
    return False

  for num in arr:
    remain = target - num
    if can_sum_memo(remain, arr, memo):
      memo[target] = True
      return True

  memo[target] = False
  return False
```

For the memoized function, what is our time and space complexity? For the space complexity, our stack will consist of m, the height of tree (target), so $\theta(m)$ is our space complexity. The number of combinations we have now are m by n so we will have $\theta(m \cdot n)$ time.

The next question is now how can we make the target sum. In this example, given a target and an array of integers, how can we make the target sum from array with replacement. If we cannot make the sum, we just return null, and the array of values otherwise. If the target is 0, we just need to return an empty array []. Let's consider the same example of a target value of 7 and an array of $[5, 4, 3, 7]$. If

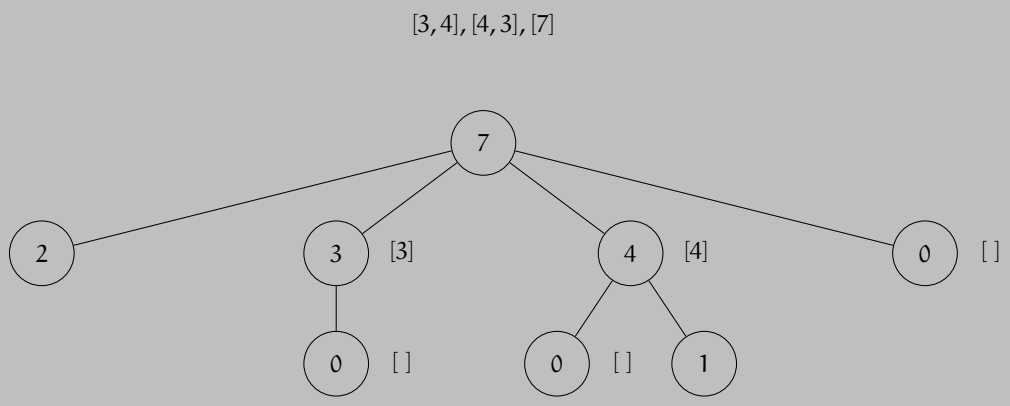

Figure 5: The binary tree with array updates.

we reach zero, we know we can create the sum. As we move up the stack with out empty array, we need to push the difference between the node values into the array. When we reach the root, we will have our arrays of values on how to sum. However, for this problem, we just need to return a single solution. We don't need to keep track of all the solutions.

```
def how_sum(target: int, arr: List[int]) -> List[int]:
  if target == 0:
    return []
  if target < 0:
    return None

  for num in arr:
    remain = target - num
    result = how_sum(remain, arr)
```

```
    if res is not None:
      return result + [num]

  return None
```

What would be our time and space complexity here? Similarly, we would have n^m steps where n is the length of the array and m is the height or the target. However, now we have to update an array that could potentially be of size m, an array of m ones. Therefore, our time complexity is $\theta(n^m \cdot m)$. For space, we have the same potential of an array of size m so our space complexity is $\theta(m)$.

As we have done before, we will now memoize the how sum function.

```
def how_sum_memo(
    target: int,
    arr: List[int],
    memo: Dict[int, List[int]]=None
    ) -> List[int]:
  if memo is None:
    memo = {}
  if target in memo:
    return memo[target]

  if target == 0:
    return []
  if target < 0:
    return None

  for num in arr:
    remain = target - num
    result = how_sum_memo(remain, arr, memo)
    if res is not None:
      new_arr = result.copy()
      new_arr.append(num)
      memo[target] = new_arr
      return new_arr

  memo[target] = None
  return None
```

Same as with other functions, we will have a combination of $m \cdot n$ but this time we also have to iterate through an array