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?

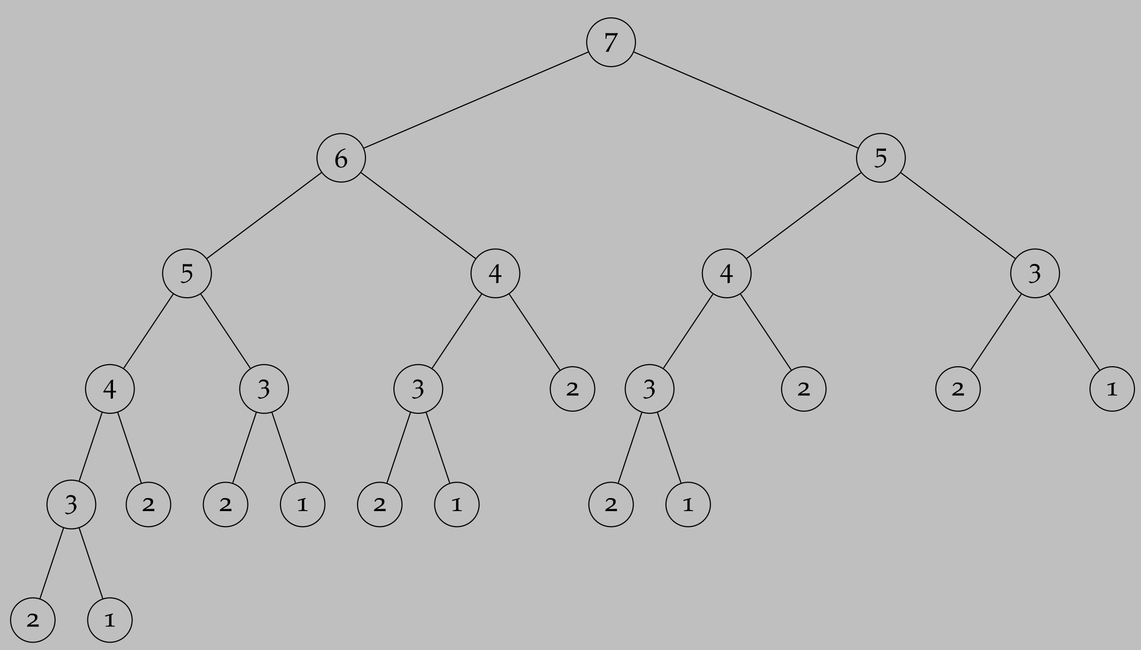

Figure 1: Steps for the binary tree of fib(7).

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 $\mathtt{fib}(5)$, $\mathtt{fib}(4)$, and $\mathtt{fib}(3)$. 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

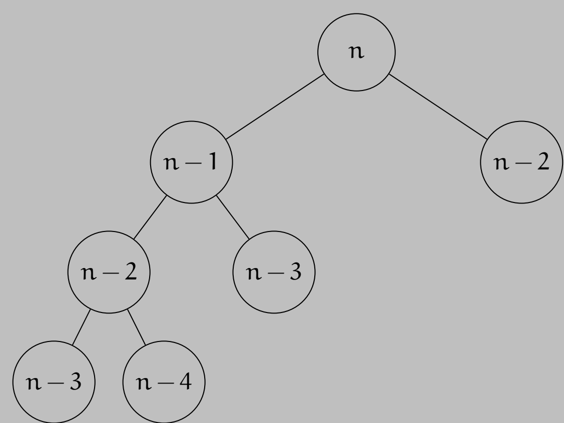

Figure 2: Steps for the binary tree of $\mathtt{fib}(n)$ memoization.

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:
```

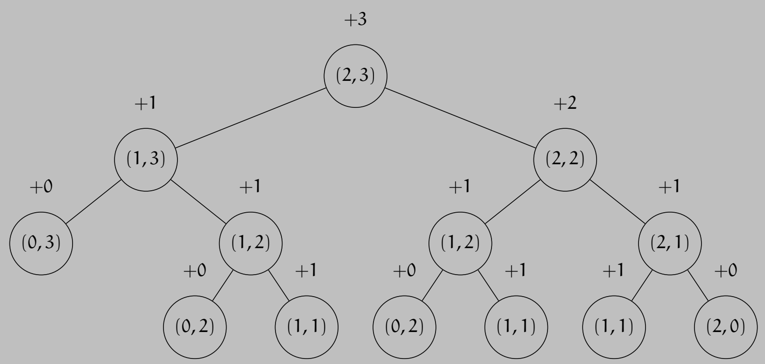

Figure 3: Binary tree for our grid traveler problem of a 2×3 grid.

```
    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

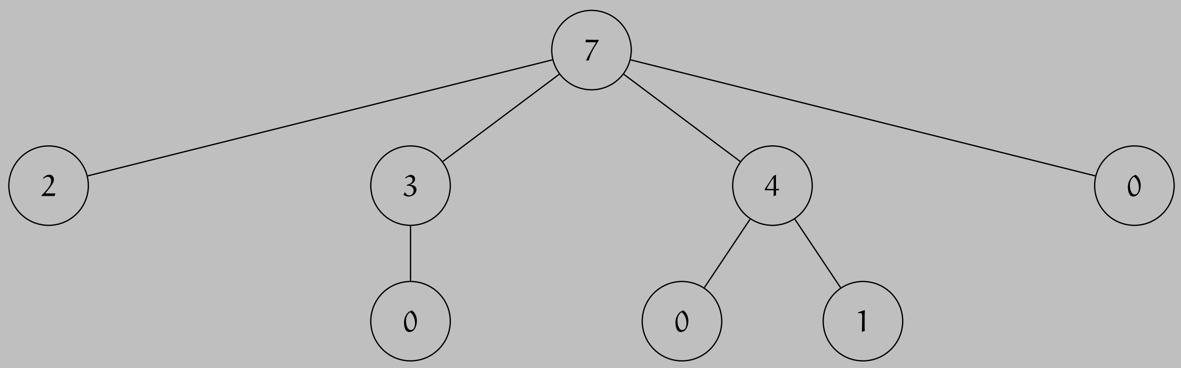

Figure 4: The tree for a target of 7 and an array of $[5, 4, 3, 7]$. The tree is pruned of any leaves that would be less than 0.

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:
```

```python
    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

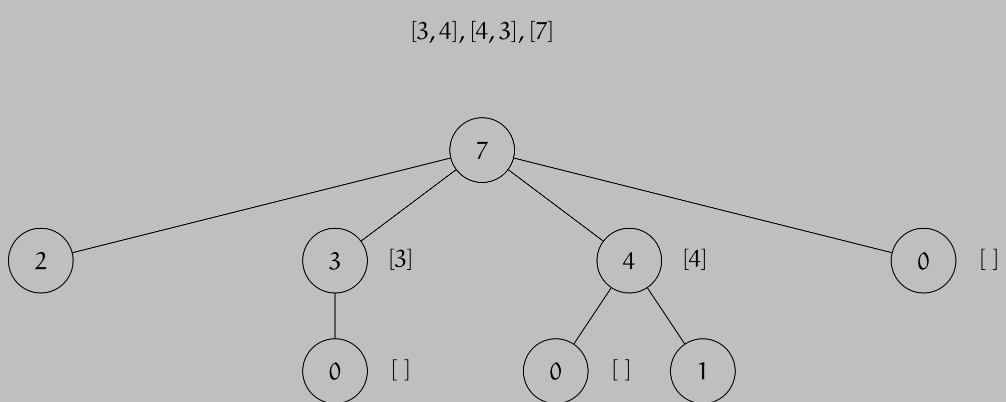

Figure 5: The binary tree with array updates.

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.

```python
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 similar to last time we might have to create an array of size m. Our time complexity is $\theta(m^2 \cdot n)$. However, unlike the brute force method where we only need to track the array, we must also track the dictionary so our space complexity is $\theta(m \cdot m) = \theta(m^2)$.

In this iteration, we want to find the best sum. We will define the best sum as the array that consists of the elements that sum to the target with the least elements. Again, let's consider a target of 7 and an array of $[5, 4, 3, 7]$. From figure 5, this time we need to keep track of all the solutions in order to select the array with the least elements. If there is more than one answer, just return one.

```
def best_sum(target, arr: List[int]) -> List[int]:
  if target == 0:
    return []
```

```
  if target < 0:
    return None

  shortest = None

  for i in arr:
    remain = target - i
    res = best_sum(remain, arr)

    if res is not None:
      res.append(i)
      if shortest is None or len(shortest) > len(res):
        shortest = res

  return shortest
```

We will have the same time complexity as how_sum here, $\theta(n^m \cdot m)$. For the space, we will have a stack space of m and an array max space of m, $\theta(m^2)$.

Now, let's construct the memoized version of best_sum.

```
def best_sum_memo(
    target,
    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

  shortest = None

  for i in arr:
    remain = target - i
    res = best_sum_memo(remain, arr, memo)

    if res is not None:
      # copy the array due to python issues with immutability
       new_arr = res.copy()
       new_arr.append(i)
       if shortest is None or len(shortest) > len(new_arr):
         shortest = new_arr

  memo[target] = shortest
```

```
  return shortest
```

We will have the same time and space complexity as how_sum_memo. That is, our complexities our $\theta(m^2 \cdot n)$ for time and $\theta(m^2)$ for space.

In the last three problems can_sum, how_sum, and best_sum, we solved

1. a decision problem with can_sum,
2. a combinatoric problem how_sum, and
3. an optimization problem with best_sum.

Next, we will look at dynamic program problems in the scope of string inputs.

In this example, we will create the function can_construct that takes as inputs a target string and an array of strings. We will return a boolean result if we can or cannot construct the target from the array. We may reuse the strings inside the array. Consider a target string of "abcdef" and an array of ["ab", "abc", "cd", "def", "abcd"]. In these types of problems, we want to start from the beginning of the string parse left to right, breaking a string in the middle can cause unintended consequences.

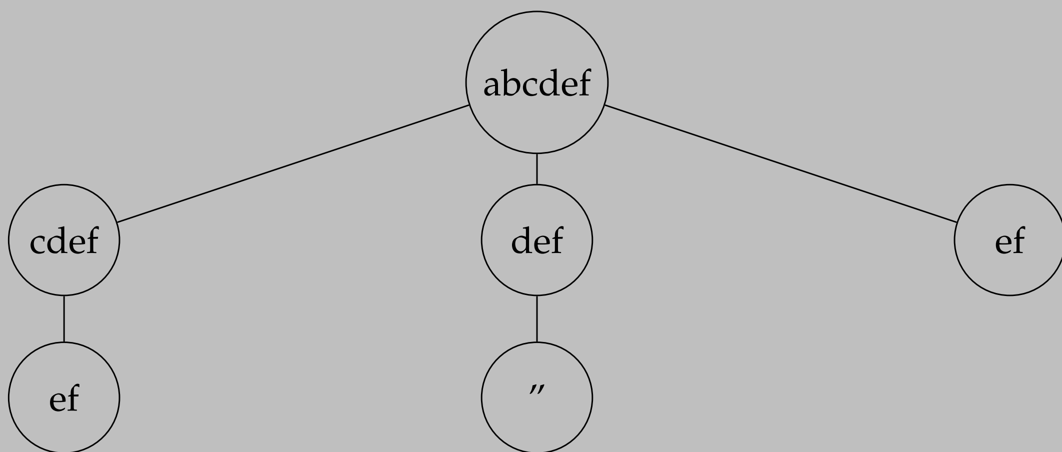

Figure 6: With strings, we are showing the remaining string after slicing the beginning of the string by the array characters. That is, the left most node is after "ab".

```
def can_construct(s: str, arr: List[str]) -> bool:
  if s == "":
    return True

  for chars in arr:
    if s.startswith(chars):
      new_s = s[len(chars):]
      if can_construct(new_s, arr):
        return True

  return False
```

In the worst case, the height of tree would be the length of the target and the number of nodes the size of the array. Let n be the height and m the number of nodes. Then our time complexity would be $\theta(m^n)$. However, in our code, we will have to slice the string which means and this will take $\theta(n)$ so our time complexity is actually $\theta(m^n \cdot n)$. As for space, we have $\theta(m^2)$ where the additionally m comes from the new_s memory.

Next, let's memoize our can_construct method.

```python
def can_construct_memo(
    s: str,
    arr: List[str],
    memo: Dict[str, bool]=None
    ) -> bool:
  if memo is None:
    memo = {}
  if s in memo:
    return memo[s]

  if s == "":
    return True

  for chars in arr:
    if s.startswith(chars):
      new_s = s[len(chars):]
      if can_construct_memo(new_s, arr, memo):
        memo[s] = True
        return True

  memo[s] = False
  return False
```

For our memoized case, we have $n \cdot m$ combinations and another n for string slicing so that leaves us with $\theta(n^2 \cdot m)$. The space will be the same as our brute force at $\theta(n^2)$.

For the next few, I will simply state the problem statement, show the brute force, memoization, and the time and space complexity but leave the explanation up to the readers to determine. Everything needed to analyze the problems has been discussed previously so let's get started.

Write a function that accepts a target string and an array of strings. In this case, we want to return the number of ways the target string can be constructed by the array of strings. Again, we can reuse elements.

```python
def count_construct(s: str, arr: List[str]) -> int:
  if s == "":
    return 1

  cnt = 0
  for chars in arr:
    if s.startswith(chars):
      new_s = s[len(chars):]
      num = count_construct(new_s, arr)
      cnt += num

  return cnt
```

```python
def count_construct_memo(
    s: str,
    arr: List[str],
    memo: Dict[str, int]=None
    ) -> int:
  if memo is None:
    memo = {}
  if s in memo:
    return memo[s]

  if s == "":
    return 1

  cnt = 0
  for chars in arr:
    if s.startswith(chars):
      new_s = s[len(chars):]
      num = count_construct_memo(new_s, arr, memo)
      cnt += num

  memo[s] = cnt
  return cnt
```

The time and space complexity for our brute force algorithm is $\theta(m^n \cdot n)$ and $\theta(n^2)$. As for the memoization, we have $\theta(n^2 \cdot m)$ and $\theta(n^2)$.

Write a function that takes a target string and an array of strings. We want to return all the ways we can construct the target string. We may use the strings in the array.

```python
def all_construct(s: str, arr: List[str]) -> List[List[str]]:
  if s == "":
    return [[]]

  result = []
  for chars in arr:
    if s.startswith(chars):
        new_s = s[len(chars):]
        result += [[chars] + i for i in all_construct(new_s, arr)]

  return result


def all_construct_memo(
    s: str,
    arr: List[str],
    memo: Dict[str, List[List[str]]]=None
    ) -> List[List[str]]:
  if memo is None:
    memo = {}
```

```
  if s in memo:
    return memo[s]

  if s == "":
    return [[]]

  result = []
  for chars in arr:
    if s.startswith(chars):
      new_s = s[len(chars):]
      result += [[chars] + i for i in all_construct_memo(new_s, arr)]

  memo[s] = result
  return result
```

The time and space complexity for our brute force algorithm is $\theta(m^n \cdot n)$ and $\theta(n^2)$. As for the memoization, we have $\theta(n^2 \cdot m)$ and $\theta(n^2)$.

In the previous examples, we looked at the dynamic programming technic of memoization. However, this isn't the only way we can do dynamic programming. In the following examples, we will use the tabulation method. Again, let's start with our Fibonacci series. With the tabulation method, we will need to setup up an array. Let's consider fib(6). Therefore, we will need an array of length ending at 6 but indexed at 0 so length 7. Our base cases are $0, 1$ for $n = 1, 2$, respectively. With tabulation, we will use three pointers. Two for the indices we are summing in the array and the third to store the new value. How do we go about solving this problem? Conceptually, what we do is we start our two

0	1	0	0	0	0	0

Table 1: For tabulation, we set up an $n+1$ length array initialized at zero and with our base cases.

pointers on our base indices 0 and 1 with the third pointer on 2. We sum our two pointers and update index 2. Afterwards, we slide our pointer over one unit and repeat. Since we are iterating through an

0	1	1	2	3	5	8

Table 2: After sliding our pointers and summing, in table 1, we get the following array.

array of size n, our space and time complexity are both $\theta(n)$.

```
def fib_tab(n: int) -> int:
  table = [0 for i in range(n + 1)]
  table[1] = 1

  for i in range(n):
    if i + 1 < n and i + 2 <= n:
      table[i + 1] += table[i]
      table[i + 2] += table[i]
    if i + 1 <= n and i + 2 > n:
      table[i + 1] += table[i]
  return table[n]
```

Next, let's go back to our grid traveler problem. In this case, we will need an 2D array of size $m \times n$. That is, both our space and time complexity will be $\theta(m \cdot n)$.

```
def grid_traveler_tab(m: int, n: int) -> int:
  table = [[0] * (n + 1) for i in range(m + 1)]
  # starting position 1, 1 due to padding
  # the top row and left column is padded with zeros
  table[1][1] = 1

  for i in range(m + 1):
    for j in range(n + 1):
      current = table[i][j]
      if i + 1 <= m:
        table[i + 1][j] += current
      if j + 1 <= n:
        table[i][j + 1] += current

  table[m][n] = table[m - 1][n] + table[m][n - 1]
  return table[m][n]
```

Let's now look at the rules for tabulation.

1. Visualize it as a table.
2. Size the table based on the inputs.
3. Initialize the table with default values like 0 or False, for example.
4. Seed the the base cases; where we automatically know the answer.
5. Iterate through the table.
6. Fill further positions based on current position.

Lastly, we will look at the can_sum problem before and leave the rest for the reader to tabulate. Given a target value of 7 and array $[5, 3, 4,]$, return true if we can sum to the target.

T	F	F	F	F	F	F	F

Table 3: Initialized array with seed values.

```
def can_sum_tab(target: int, arr: List[int]) -> bool:
  table = [False for i in range(target + 1)]
  table[0] = True

  for i in range(target + 1):
    if table[i] == True:
      for val in arr:
        if i + val <= target:
          # add the val in array to current position and
          # replace with true
          table[i + val] = True
```

```
    return table[target]
```

Why add current and array value? Let's walk through this example.

1. Starting index 0: $0 + 5$, $0 + 3$, and $0 + 4$
2. Index 3: $3 + 5 > 7$, $3 + 3$, $3 + 4$; True now at end
3. Index 4: $4 + 5 > 7$, $4 + 3$, $4 + 4 > 7$
4. Index 5: all greater than 7

By doing this, we obtain True at the target index if any sums add up to it. We do this in $\theta(m \cdot n)$ time with space $\theta(m$ when $m =$ the target value.