Solution Review: The Catalan Numbers

In this lesson, we will review some solutions to the Catalan numbers challenge from the last lesson.

We'll cover the following

Solution 1: Simple recursion

```
def catalan(n):
  if n == 0:      # base case; C(0) = 1
    return 1
  sum = 0
  # iterate from 1...n to evaluate: C(0)*C(n-1) + C(1)*C(n-2) ... + C(n-1)*C(0)
  for i in range(n):
    sum += (catalan(i) * catalan(n-1-i))  # C(i)*C(n-1-i)
  return sum

print(catalan(4))
```

Explanation

This is a much simpler problem. The only thing we need to figure out is how to convert the following summation equation into code:

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}$$

or

$$C_n = C_0 C_{n-1} + C_1 C_{n-2} + C_2 C_{n-3} + \ldots + C_{n-1} C_0$$

So, for the sum, we simply use a for loop to iterate from `0` to `n-1`, and then we make calls to `catalan(i)` and `catalan(n-1-i)` as required by the expression (*lines 6-7*). The following is our base case:

$$C_0 = 1$$

This has been handled in *lines 2 and 3*.

Although the repeating subproblems are very apparent in this problem, let's still look at a dry run of the algorithm to get a better hang of things.

expand catalan(3)

+ \+

catalan(0) is equal to 1, expand catalan(2)

3 of 7

+ +

+

expanding both catalan(1) in the middle

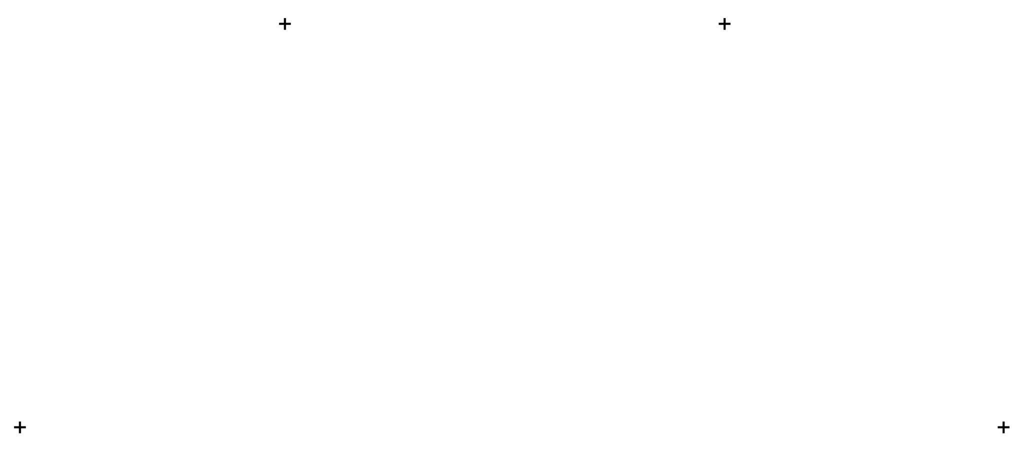
4 of 7

+ +

+

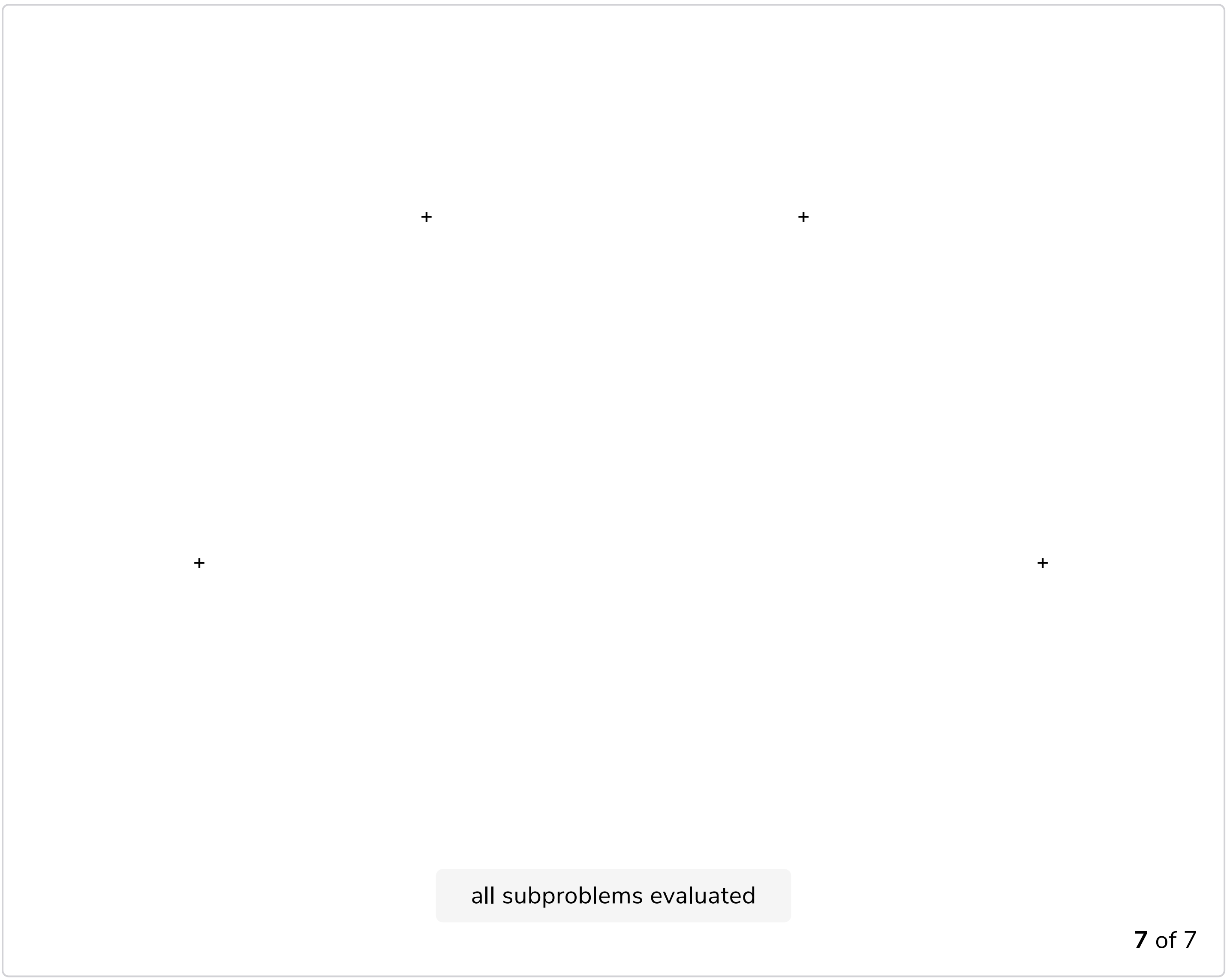
expand other catalan(2) as well

5 of 7



catalan(0) has been evaluated again, expanding catalan(1)'s

6 of 7



Time complexity

The time complexity here is in order of factorials. If we solve the recurrence relation of Catalan numbers given by the two expressions, we get the following closed-form equation:

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

This further evaluates to

$$C_n = \frac{2n!}{(n+1)!n!}$$

Thus, the runtime complexity becomes **O(n!)**.

In the above slides, we can already see a number of repeating computations. This shows there is a lot of room for improvement if we use dynamic programming.

Solution 2: Top-down approach

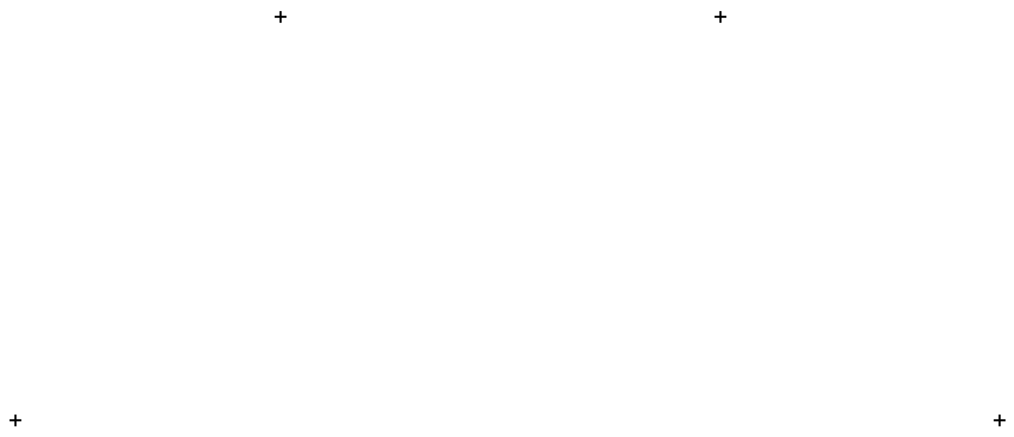
Before using dynamic programming, let’s see if this problem satisfies both conditions of dynamic programming.

Optimal substructure

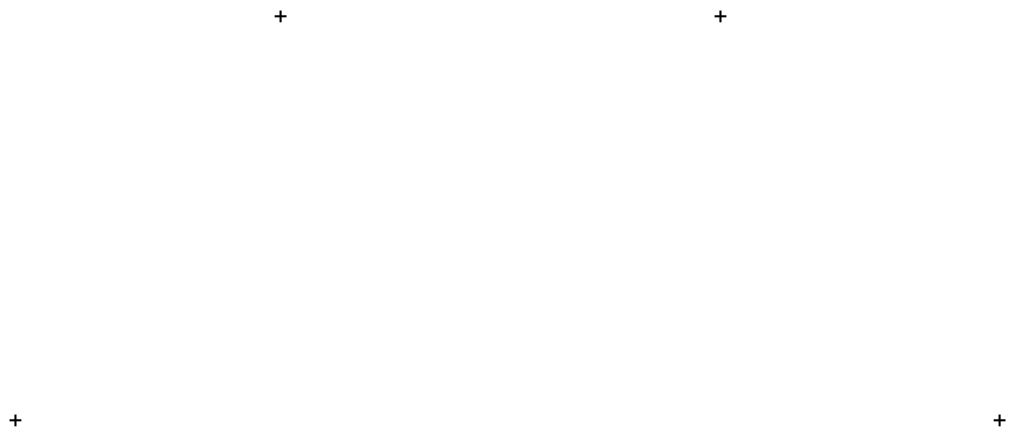
Every Catalan number is constructed using all the previous Catalan numbers. Thus, if I had `n` Catalan numbers available, I could construct `n+1`th by simply using these `n` Catalan numbers. This shows the problem has an optimal substructure.

Overlapping subproblem

You can already see many repeating subproblems in the formula of the Catalan numbers. Still, to get a better idea, look at the visualization below to see the number of overlapping subproblems.



catalan(2) repeating twice



repeating computation due to catalan(2)

3 of 5

catalan(1) repeating 6 times

4 of 5

repeating computation due to catalan(1)

5 of 5

This asks for results to be memoized, so let's look at an improved recursive solution with memoization i.e., top-down dynamic programming solution.

```
def catalan_memo(n, memo):
  if n == 0:          # base case; C(0) = 1
    return 1
  elif n in memo:       # if n already evaluated, return from dp
    return memo[n]
  sum = 0
  # iterate from 1...n to evaluate: C(0)*C(n-1) + C(1)*C(n-2) ... + C(n-1)*C(0)
  for i in range(n):
    sum += (catalan_memo(i, memo) * catalan_memo(n-1-i, memo))  # C(i)*C(n-1-i)
  memo[n] = sum         # store result in dp
  return memo[n]


def catalan(n):
  memo = {}
  return catalan_memo(n, memo)

print(catalan(400))
```

Great! Now we can run our algorithm on slightly larger numbers.

Explanation

There's nothing fancy here: just a quick look at `memo` before evaluation (*line 4*) and storing in `memo` after evaluation (*line 10*).

Time and space complexity

We have avoided recomputations by storing all the results in the `memo`. Since each Catalan number is evaluated by using all the previous Catalan numbers, our algorithm will have a runtime complexity of **$O(n^2)$**. Let's look at the intuition behind this. If we wanted to evaluate the n^{th} Catalan number, we are going to need the sum of all previous n-1 numbers. To evaluate the $(n\text{-}1)^{th}$ number we will need to sum up all previous n-2 numbers. This way we would end up with the following number of steps:

$$= (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1$$

$$= \frac{n(n-1)}{2}$$

$$\in O(n^2)$$

The size of our `memo` dictionary is equal to `n` since there are `n` total subproblems. Thus, the space complexity is **O(n)**.

Solution 3: Bottom-up dynamic programming

If you plug a bigger number (say 1000) into solution two, it will not evaluate due to `RecursionError`. Also, we have seen that recursion can be expensive as well, so let's look at a bottom-up solution to this problem.

```
def catalan(n):
  table = [None] * (n+1)  # tabulating
  table[0] = 1            # handling the base case
  for i in range(1,n+1):  # iterating to fill up the tabulation table
    table[i] = 0          # initializing the i-th value to 0
    # iterate from 0 to i; according to formula of catalan i.e.
    # C0*Ci + C1*Ci-1 + ... Ci*C0
    for j in range(i):
      table[i] += (table[j] * table[i-j-1]) # C(j) * C(i-j-1)
  return table[n]

print(catalan(1000))
```

Explanation

We start by creating a list of size `n+1`, called `table`, for tabulation and set element at 0th index to 1 to satisfy base case i.e

$$C_0 = 1$$

Next, we start filling `table`, starting with the second element. For each entry in `table`, we simply plug in all the previous entries into the formula to evaluate the current entry. Evaluation of the ith Catalan Number happens in *lines 8-9*.

Look at the following visualization for the dry run of the algorithm.

catalan(3)

Evaluate catalan(3)

1 of 14

i	0	1	2	3
C(i)				

table

construct table of size n+1 i.e. 4

2 of 14

i	0	1	2	3
C(i)	1			

table

base case: update 0th index to cater for C(0)

3 of 14

i	0	1	2	3
C(i)	1			

table

start filling the table beginning with C(1)

4 of 14

i	0	1	2	3
C(i)	1			

table

C(1) = C(0) * C(0)

evaluating C(0)

5 of 14

i	0	1	2	3
C(i)	1			

table

C(1) = C(0) * C(0)
C(1) = 1 * 1

C(0) is in the table

6 of 14

i	0	1	2	3
C(i)	1	1		

table

C(1) = C(0) * C(0)
C(1) = 1 * 1
C(1) = 1

Update value of C(1) in table

7 of 14

i	0	1	2	3
C(i)	1	1		

table

C(2) = C(1) * C(0) + C(0) * C(1)

evaluating C(2)

8 of 14

i	0	1	2	3
C(i)	1	1		

table

C(2) = C(1) * C(0) + C(0) * C(1)
C(2) = 1 * 1 + 1 * 1

C(0) and C(1) are in the table

9 of 14

i	0	1	2	3
C(i)	1	1	2	

table

C(2) = C(1) * C(0) + C(0) * C(1)
C(2) = 1 * 1 + 1 * 1
C(2) = 2

Update value of C(2) in table

10 of 14

i	0	1	2	3
C(i)	1	1	2	

table

C(3) = C(2) * C(0) + C(1) * C(1) + C(0) * C(2)

evaluating C(3)

11 of 14

i	0	1	2	3
C(i)	1	1	2	

table

C(3) = C(2) * C(0) + C(1) * C(1) + C(0) * C(2)
C(3) = 2 * 1 + 1 * 1 + 1* 2

C(0) , C(1) and C(2) are in the table

12 of 14

i	0	1	2	3
C(i)	1	1	2	5

table

C(3) = C(2) * C(0) + C(1) * C(1) + C(0) * C(2)
C(3) = 2 * 1 + 1 * 1 + 1* 2
C(3) = 5

Update value of C(3) in table

13 of 14

i	0	1	2	3
C(i)	1	1	2	5

table

C(3) = C(2) * C(0) + C(1) * C(1) + C(0) * C(2)
C(3) = 2 * 1 + 1 * 1 + 1* 2
C(3) = 5

C(3) evaluated return it

14 of 14

Time and space complexity

The time complexity will be **O(n^2)** due to the same logic we saw in solution two. Space complexity would be **O(n)**, the same as solution two.

However, if you see the visualization, both these time complexities become easier to visualize. Space complexity would be **O(n)** because the size of the `table` is `n+1`. For time complexity look at slides 5, 8, and 11 of the above visualization. Notice how the length of formulas uniformly increases. Two values in first, then four, and then six. `C(1)` is using the result of just `C(0)`, `C(2)` is using the results of `C(1)` and `C(0)`, whereas `C(3)` uses `C(2)`, `C(1)` and `C(0)`. This way, if we extend to `C(n)`, we will form the series we saw in the time complexity explanation of solution two, making it an **O(n^2)** algorithm.

In the next lesson, you will solve another coding challenge.