Methodology:

Dynamic programming is inherently tied to optimization, so there are a number of ways to approach the rod-cutting problem depending on the solution you are optimizing for. We are looking for the maximum sales price of a piece i in units long. Two popular approaches involve using either recursion(top-down), or by solving each smaller sub-problem only once, using nested loops (bottom-up). Given that I have seen the top-down implementation, I wanted to try to implement the bottom-up version from scratch, using the initial conditions to lay out the algorithm and then to be compared to the top-up version for analysis. This is a bottom-up rod cutting algorithm implemented in Python 3.

<u>Conditions:</u>

-Solve the smaller subproblem before moving to the next smaller subproblem(i<j)

-i is smaller subproblem than j if i<j

-from i to j(i<j), from j = 0,1,..n

-We can break this down into two loops

-We need a new array to save the results of the subproblems, i.e, the revenue per rod length.

-r[0] = 0 rod length

- For every smaller subproblem, we need to compute the max q:

-This max must consider the previous max q, and the price per length i and max revenue r = max(p_i+ r_{n-i}): j is less than or equal to n.

-Let j $\leq n$

-We need to save the max q into revenue.

-After solving every smaller subproblem, the solution is calculated and returned as r[n], the last index of the maxRevenue array.

Given that this algorithm considers two nested loops as a function of n, it is easy to assume that the time efficiency is that of n^2.

$$Theoretical\ Time\ Efficiency = \Theta(n^2)$$

This algorithm returns the last index of the maxRev[] array, saving the maximum revenue for each number of cut pieces i=1,2…n. This index will always be equal to n. We also stated in the condition, let j<= n. This implementation in python only appends to the array from 0…j<=n (inner loop). Therefore, the space efficiency is:

$$Theoretical\ Space\ Efficiency = \Theta(n)$$

Testing:

Utilizing the time library, we can analyze the execution time of this rod-cutting function and compare this to the theoretical efficiency.

To test this algorithm further, we will increase the sizes of n to verify that my implementation in Python 3 matches the theoretical time efficiency stated in methodologies.

This analysis will start at n=10, and then increase tenfold for five separate test cases. These timed values will be recorded as experimental values.

The theoretical values will also be calculated starting at n =10 increasing tenfold. An average of each will then be calculated and compared against each other to check for the percent error between the experimental and theoretical values.

Results:

```
TestCase#1:
maxRevenue: 11 Time 0.00010350599950470496 n =  10
_______________
TestCase#2:
maxRevenue: 20 Time 0.002760080000037109 n =  100
_______________
TestCase#3:
maxRevenue: 200 Time 0.4925601650011231 n =  1000
_______________
TestCase#4:
maxRevenue: 2000 Time 53.689861536000535 n =  10000
_______________
```

n	$\Theta(n^2)$	**Theoretical**	**Experimental**	**Expected**
10	**100**	**.000103 seconds**	**.0001035 seconds**	**x1**
100	**10000**	**.001035 seconds**	**.0027600 seconds**	**10^2 x Longer**
1000	**1000000**	**.13505 seconds**	**0.492560 seconds**	**10^2 x Longer**
10000	**100000000**	**13.505 seconds**	**53.689 seconds**	**10^2 x Longer**

$$\boldsymbol{TheoreticalAvg. = \frac{.000103 + .001035 + .13505 + 13.505}{4} = 3.41}$$

$$\boldsymbol{ExperimentalAvg. = \frac{.000103 + .0027600 + .492560 + 53.689}{4} = 13.64}$$

$$\boldsymbol{Percent\ Error(\%) = \frac{|3.41 - 13.64|}{3.41} * 100 = 300\%\ error}$$

Conclusion:

The downside to the perf_Counter() function from the Time() library is that the returned value is approximated and not entirely accurate. A better approach is running the testable function N times and calculating an average between the runtimes returned. With this being said, perf_Counter() can be valuable for some quick insight and analysis when there is not enough time to setup a more complex testing environment.

Theoretically, the execution time for the RodCut() function should grow by a factor of n^2. We can summarize from this that the time efficiency of the experimental implementation should increase by a factor of 10^2, with n=10. This is not observable between the 1st and 2nd test case, but the other test cases increase by a factor $\geq 10^2$.

The calculated percent error is far too off to be considered valid, but there is an observable pattern between the experimental time efficiency and n. There is a direct relationship between the two variables, and the time efficiency of the bottom-up Rod Cutting algorithm grows exponentially as a function of n. Considering the runtime environment of Python 3 and the Repl.it IDE, the first recorded experimental value may be too small(negligible) to be considered for calculations against our theoretical values. The first recorded number is less than a millisecond for example, and jumping between the scales will add loss to precision.

Therefore, the experimental time efficiency of this bottom-up Rod Cutting implementation follows a noticeable and observable pattern. Increasing by factors of 10^2 for values > .1 millisecond.