

Where to Use Dynamic Programming


In this lesson, we will learn about some problem characteristics in dynamic programming.

We'll cover the following

- Dynamic programming, the magic wand
- Dynamic programming, the 'selective' magic wand
- Optimal substructure
 - Count characters in a string
 - N queens problem
- Overlapping subproblems

Dynamic programming, the magic wand

Do you remember the run time complexity of the Fibonacci algorithm from [earlier lessons](#)? We even saw how bad that algorithm performed as we tried to calculate a slightly higher (~50) Fibonacci number. We have modified that algorithm to remember the values it has already evaluated to avoid re-evaluations. Run this code with different values to see how much time the algorithm takes. For comparison, we have reproduced the simple recursion-based solution below as well as in the other tab.

 DynamicProgrammingSolution

 SimpleRecursion

```
import time
import matplotlib.pyplot as plt

calculated = {}

def fib(n):
    if n == 0: # base case 1
        return 0
    if n == 1: # base case 2
        return 1
    elif n in calculated:
        return calculated[n]
    else: # recursive step
        calculated[n] = fib(n-1) + fib(n-2)
        return calculated[n]
```



```
showNumbers = True  
numbers = 20
```



Magical! Look at the almost linear time complexity curve formed now instead of an exponential curve. Try updating the value now with numbers as big as 500. When trying bigger numbers, set the `showNumbers` variable to `False` so that the graph does not become messy. This is just a primer of how wonderful dynamic programming is. We are going to discuss the Fibonacci numbers example in depth in later chapters.

Dynamic programming, the ‘selective’ magic wand

As amazing as dynamic programming is, you cannot use it on every kind of problem. There are a number of problems that entail an exponential complexity, but their complexity cannot be reduced even after using dynamic programming. In fact, a problem needs to meet very strict prerequisites before it can be solved using dynamic programming. These prerequisites are **optimal substructure** and **overlapping subproblems**.

Optimal substructure

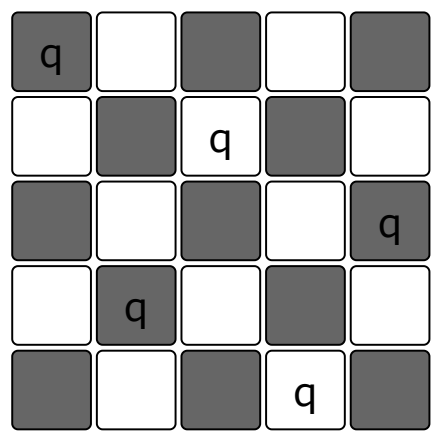
This property means that an optimal solution to a problem can be built using the optimal solutions to that problem’s subproblems. Essentially, we should be able to specify the problem in terms of its subproblems in such a way that if we know the optimal answer to the subproblem, the evaluation of the problem can simply use that answer. This might seem straightforward, but it is a little tricky. Let’s look at two examples to see what we mean.

Count characters in a string

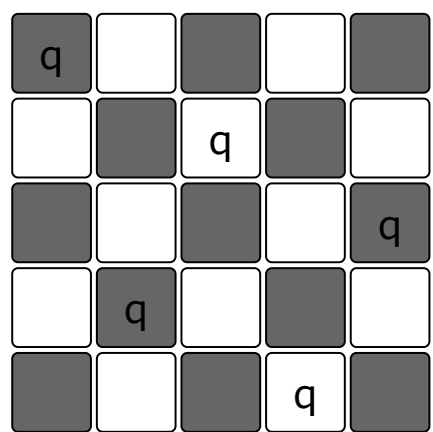
We saw this [problem](#) in the second lesson of this course where we were counting characters in a string. Let’s say we already know there are `k` number of `char`s in the string of length `n`. If we were to construct another string by appending a character to `str`, we could use the result of `str` to count character `char` in the new string. The count would simply be `k+1` if the new character was equal to `char` and `k` otherwise.

N queens problem

Let's say we already know the placement of n queens on an $n \times n$ board. We cannot use this information to find the answer for an $(n+1) \times (n+1)$ board. Look at the visualization below.

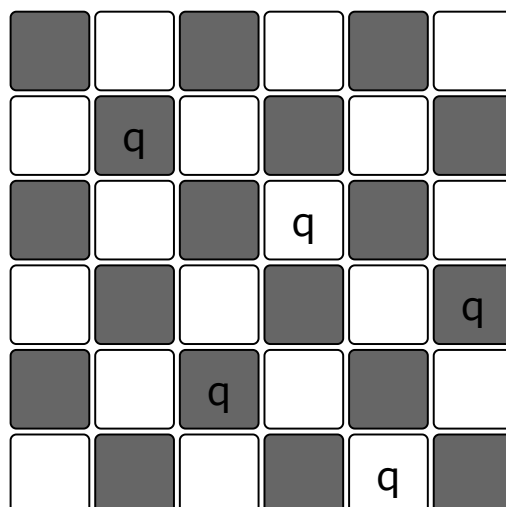


A 5x5 board with 5 queens placed correctly.



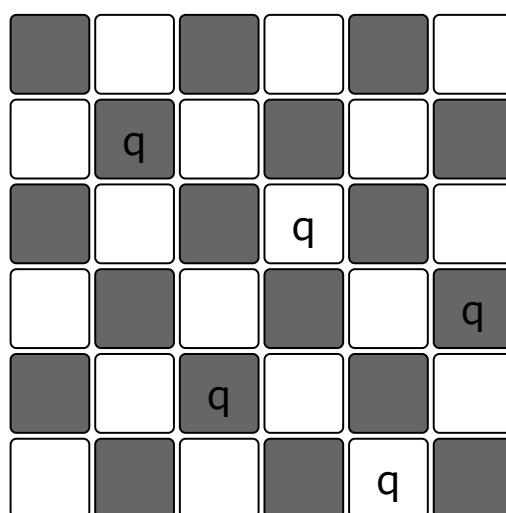
Let's add a row and column to make it a 6x6 board.

2 of 5



Let's add a row and column to make it a 6x6 board.

3 of 5



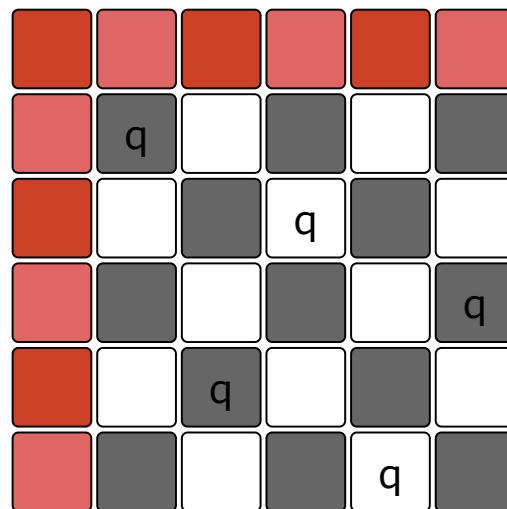
Try to place a queen anywhere.

4 of 5

Diagonal shared

Columns shared

Rows shared



You would see it shares row, column or diagonal with other queens.

5 of 5

—

[]

This also shows that the ability to express an algorithm recursively does not mean that that algorithm has an optimal substructure.

Overlapping subproblems

We now know dynamic programming is all about avoiding recomputations. Therefore, dynamic programming will only be beneficial to those problems that have repeating subproblems. For example, we have already seen how the Fibonacci numbers algorithm has so many overlapping subproblems.

In summary, a problem only benefits from dynamic programming if it has a solution that builds from the solution of subproblems and if these subproblems are being evaluated over and over again.

We have now covered the primer on dynamic programming. We will conclude the chapter with a small quiz in the next lesson.

