# **Dynamic Programming Introduction**

* A lot of people make a big deal about dynamic programming problems but there is really no need to be afraid of them.
* There are often common patterns we can utilize to simplify DP problems.

## **Dynamic Programming Defined**

* Dynamic Programming is a general approach to solving recursive problems.
* There are other, similar, recursive methods such as “divide-and-conquer”.
* Except that unlike divide-and-conquer, Dynamic Programming problems involve

1. Taking a recursive algorithm and finding the ***overlapping subproblems*** (that is, the repeated calls/work).
2. Eliminating/removing these repeated calls to cut down on runtime by ***caching*** results we have already calculated for future recursive calls.

## **How to Approach Dynamic Programming Problems**

* A good way to approach DP problems is often to
  1. Implement a normal recursive solution (naïve solution)
  2. Identify any overlapping subproblems
  3. Add a cache to remove repetition
* Where a naïve approach may take O(2n) exponential time to solve a problem, we can use these optimizations to reduce the runtime to O(n2) or O(n3).

**Implementing Recursive Solutions**

* How do we implement a recursive solution?
* By definition, recursive solutions are built off of **solutions to subproblems**.
* There are many ways you might divide a problem into subproblems.
* Many times, this will mean simply to compute f(n) by
  + adding something,
  + removing something,
  + halving something,
  + or
  + otherwise changing the solution for f(n-1)
* In some cases, you might solve the problem for the first half of the data set, then the second half, and then merge those results.
* Unlike divide-and-conquer (as in mergesort or quicksort) it is OK if our subproblems overlap, so long as there are not too many of them.
* Three of the most common approaches to develop a recursive algorithm are

1. **Bottom-Up Approach**
   1. The bottom-up approach is often the most intuitive.
   2. We start with knowing how to solve the problem for a simple base case.
   3. Then we figure out how to solve break the problem down into a smaller and smaller set of subproblems. First for two elements, then for three elements, and so on.
   4. The key here is to think about how you can build the solution for one case off of the previous case (or multiple previous cases).
2. **Top-Down Approach** 
   1. The top-down approach can be more complex since it's less concrete. But sometimes, it's the best way to think about the problem.
   2. In these problems, we think about how we can divide the problem for case N into subproblems.
   3. Be careful of overlap between the cases.
3. **Half-and-Half Approach** 
   1. In addition to top-down and bottom-up approaches, it's often effective to divide the data set in half.
   2. For example, binary search works with a "half-and-half" approach. When we look for an element in a sorted array, we first figure out which half of the array contains the value. Then we recurse and search for it in that half.
   3. Merge sort is also a "half-and-half" approach. We sort each half of the array and then merge together the sorted halves.