Dynamic Programming

In both **dynamic programming** and **divide-and-conquer**, the problem to be solved has an **optimal substructure**, meaning that the solution to a problem is obtained by the combination of solutions to its subproblems (often found recursively).

In **dynamic programming**, the subproblems are often **overlapping**, meaning that some subproblems may be solved repeatedly as a part of building up solutions to various other subproblems. In other words, multiple subproblems may rely on one particular recursive call.

As an example of **overlapping** subproblems, suppose we had a program to calculate the n^{th} Fibonacci number based on the recursive calculation: Fib(n) = Fib(n-1) + Fib(n-2). Calculating the 50^{th} Fibonacci number requires two recursive calls since Fib(50) = Fib(49) + Fib(48). And in carrying out the recursion, you would need to calculate Fib(49) = Fib(48) + Fib(47) and also Fib(48) = Fib(47) + Fib(46). Notice that in these recursive calculations, there are multiple recursive calls to Fib(48) and Fib(47), that is, the subproblems are overlapping.

To contrast, in divide-and-conquer, the subproblems are non-overlapping and typically the problem is split into subproblems by a fixed and regular procedure. For example, mergeSort requires splitting an array into two non-overlapping halves and solving (sorting) those subproblems recursively, then combining (merging) the sorted halves.

To improve upon the efficiency of recursion with overlapping subproblems, dynamic programming only solves each subproblem once and then stores the solutions to subproblems in a table (or array); any future references to that subproblem are simply looked up in the table. There are two approaches for building these subsolutions and storing them:

- **top-down approach**: This approach often works directly from the recursive formulation for the problem to be solved. The recursive formulation breaks the problem into subproblems, but to avoid repeated calls to the subproblems, we use **memoization**. That is, whenever an attempt is made to solve a subproblem, we first check the table to see if it has already been solved. If so, we simply return that value; if not, we solve the subproblem (recursively) and then store the solution to that subproblem in the table.

- **bottom-up approach**: This approach solves the smallest subproblems first and stores them in a table. Then we use the solutions to the smallest sub-problems to build solutions to larger and larger sub-problems, eventually arriving at the solution for the whole problem.

Think: top-down breaks down, bottom-up builds up.

NOTE: Dynamic programming can frequently improve efficiency of a recursive algorithm, but the **trade-off is memory**. Furthermore, the use of a dynamic programming algorithm may be restricted to certain ("small") input sizes because the algorithm requires a table of values whose size is related to the input size.