**Introduction to Dynamic Programming**

*Borrowed with minor formatting changes from:* [*www.techiedelight.com/introduction-dynamic-programming/*](http://www.techiedelight.com/introduction-dynamic-programming/)

**Dynamic programming** is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions using a memory-based data structure (array, map, etc). Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup. So the next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time. This technique of storing solutions to subproblems instead of recomputing them is called **memoization**.

Here’s brilliant explanation given by [Jonathan Paulson](https://www.quora.com/profile/Jonathan-Paulson) on [Quora](https://www.quora.com/How-should-I-explain-dynamic-programming-to-a-4-year-old/answer/Jonathan-Paulson) on concept of Dynamic Programming to a kid.

*\*writes down “1+1+1+1+1+1+1+1 =” on a sheet of paper\*  
“What’s that equal to?”  
\*counting\* “Eight!”  
\*writes down another “1+” on the left\*  
“What about that?”  
\*quickly\* “Nine!”  
“How’d you know it was nine so fast?”  
“You just added one more”  
“So you didn’t need to recount because you remembered there were eight! Dynamic Programming is just a fancy way to say ‘remembering stuff to save time later'”*

There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping sub-problems.

1. Optimal substructure

Dynamic programming simplifies a complicated problem by breaking it down into simpler sub-problems in a recursive manner. A problem that can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems is said to have optimal substructure. In other words, solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub-problems.

For example, the shortest path *p* from a vertex *u* to a vertex *v* in a given graph exhibits optimal substructure: take any intermediate vertex *w* on this shortest path *p*. If *p* is truly the shortest path, then it can be split into sub-paths*p1* from *u* to *w* and*p2* from *w* to*v* such that these, in turn, are indeed the shortest paths between the corresponding vertices.

1. Overlapping sub-problems

A problem is said to have **overlapping subproblems** if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems.

For example, the problem of computing the Fibonacci sequence exhibits overlapping subproblems. The problem of computing the *n*th Fibonacci number *F*(*n*), can be broken down into the subproblems of computing *F*(*n* – 1) and *F*(*n* – 2), and then adding the two. The subproblem of computing *F*(*n* – 1) can itself be broken down into a subproblem that involves computing *F*(*n* – 2). Therefore the computation of *F*(*n* – 2) is reused, and the Fibonacci sequence thus exhibits overlapping subproblems. Dynamic programming takes account of this fact and solves each sub-problem only once. This can be achieved in either of two ways –

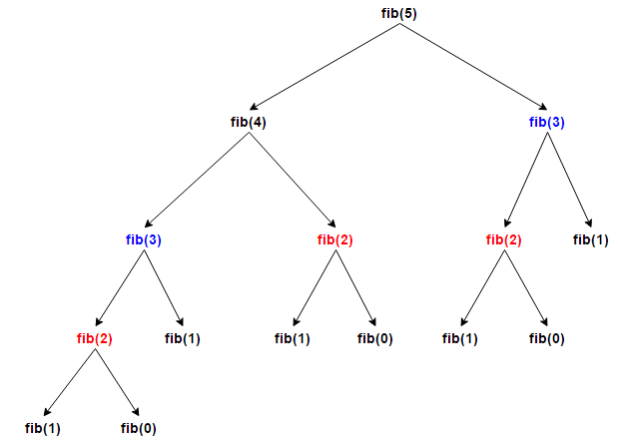
1. **Top-down approach (Memoization):** This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its sub-problems, and if its sub-problems are overlapping, then one can easily memoize or store the solutions to the sub-problems in a table. Whenever we attempt to solve a new sub-problem, we first check the table to see if it is already solved. If the sub-problem is already solved, we can use its solution directly, otherwise we solve the sub-problem and add its solution to the table.
2. **Bottom-up approach (Tabulation):** Once we formulate the solution to a problem recursively as in terms of its sub-problems, we can try reformulating the problem in a bottom-up fashion: try solving the sub-problems first and use their solutions to build-on and arrive at solutions to bigger sub-problems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger sub-problems by using the solutions to small sub-problems. For example, if we already know the values of *F*(*i* – 1) and *F*(*i* – 2), we can directly calculate the value of *F*(*i*).

If a problem can be solved by combining optimal solutions to non-overlapping sub-problems, the strategy is called “divide and conquer” instead. This is why merge sort and quick sort are not classified as dynamic programming problems.

Let’s consider a naive implementation of a function finding the nth member of the Fibonacci sequence-

|  |
| --- |
| // Function to find n'th Fibonacci number  int fib(int n)  {      if (n <= 1)          return n;        return fib(n - 1) + fib(n - 2);  } |

Notice that if we call, say, fib(5), we produce a call tree that calls the function on the same value many different times:

In particular, fib(3) was calculated two times and fib(2) was calculated three times from scratch. In larger examples, many more subproblems are recalculated, leading to an exponential time algorithm.

As you've already seen, the technique of saving values that have already been calculated is called memoization; this is the top-down approach, since we first break the problem into subproblems and then calculate and store values.

In the dynamic programming bottom-up approach, we calculate the smaller values of fib first, then build larger values from them. This method also uses O(n) time since it contains a loop that repeats n-1 times, but it only takes constant O(1) space, in contrast to the top-down approach which requires O(n) space to store the map.

public int fibDP(int n)

{

int[] result = new int[n + 2];

result[0] = 0; //base case 1

result[1] = 1; //base case 2

for (int i = 2; i <= n; i++)

result[i] = result[i – 2] + result[i -1];

return result[result.length];

}

In both memoization and tabulation, we only calculate fib(2) once, and then use it to calculate both fib(4) and fib(3), instead of computing it every time either of them is evaluated. The difference is the bottom-up dynamic programming (tabulation) trades space for time, solving the simplest subproblems *first*, then building subsequent solutions from previous solutions.

Memoization is actually a more generally applicable technique, as you don't have to calculate every subproblem as in tabulation – you only calculate those subproblems as are applicable. In addition, memoization doesn't require you to re-learn the algorithm – you simply store previous solutions and look them up in the data structure of your choice.

However, dynamic programming (tabulation) can result in beautiful, simple, and very fast algorithms given problems of the right nature. Just like recursion, it takes a LOT of practice to get good at!