# **Chapter 3: Dynamic Programming**

## **Introduction**

When you use the divide and conquer method of problem solving, you can often end up solving the same problem more than once. For example, consider the recursive fibonacci algorithm:

**int fibonacci (int n)**

**{**

**if (n == 0 || n == 1)**

**return 1;**

**return fibonacci (n - 1) + fibonacci (n - 2);**

**}**

What recursive calls are made when we call this to find the 4th fibonacci number?  You can see from the tree that we calculate the value of fibonacci (2) two times.   If you draw the tree for initially passing in and index of 5, you'll see that the value of fibonacci (3) is calculated two times, and the value of fibonacci (2) is calculated three times.  As you calculate higher and higher fibonacci numbers, **you end up with a large amount of duplicated work.**

In algorithm terms, **we say that the solutions are overlapping**.  If the solutions have a high degree of overlap, divide and conquer performs more work than it really needs to.  This results in the divide and conquer fibonacci algorithm being **O(2n),** which is quite inefficient.

Dynamic programming takes a different approach.  Rather than calculate from the top-down, like the divide and conquer approach, **it calculates from the bottom up**.   **Each fibonacci number is calculated exactly once**.  Here's the dynamic programming approach to calculating the fibonacci sequence:

**int fibonacci (int n)  
{  
    int array [n + 1];**

**array [0] = 0;  
    array [1] = 1;**

**for (i = 2; i <= n; i++)  
        array [i] = array [i - 1] + array [i - 2];**

**return array [n];  
}**

This approach calculates starting at the start of the fibonacci numbers, and builds up to the one we want.  The divide and conquer approach started at the one we wanted, and calculated downward**.  We calculate every fibonacci number exactly once, so this is a linear, O(n), algorithm.  Much more efficient than the original divide and conquer algorithm.**

**Dynamic programming** always involves using an array or list to store intermediate calculations while we build a final result.

Before you can use dynamic programming, the principle of optimality must be true.   **This essentially means that an optimal solution to the entire problem will consist of optimal solutions to all the pieces.**  We can then ignore non-optimal solutions to the pieces rather than calculate them and discard them.  The Principle of Optimality is **NOT TRUE** for all problems, but if it is true for a problem then we may be able to use dynamic programming to solve it in an efficient manner.

When you are trying to use dynamic programming, you follow these steps:

1. **Establish a recursive solution to the problem** (this is essentially the same as what you do when you use **divide and conquer**).
2. **Compute the value of an optimal solution by working on the smallest pieces of it first**, making certain that each piece is solved only once.  This results in an iterative approach to solve the problem.
3. Construct an optimal solution for the entire problem by **combining the optimal solutions for the pieces, starting with the smallest piece first**.  This step may use a recursive algorithm.

## **Floyd's Shortest Path Algorithm**

A common problem in graphs is to determine the shortest path between any two vertices.   Perhaps the graph represents a communications network, or the cities an airline flies to, or cities served by the postal service.  Finding the most efficient way to move data, people, or packages saves time and money for the company and makes the customers happy.

**Floyd's shortest path algorithm calculates the cost of the shortest path from any vertex to any other vertex.**  This algorithm assumes a directed graph that is also weighted.  So, the cost of a path is the sum of the cost of the edges in the path.

**A brute force approach** to this would be to simply examine all the possible paths and pick the minimum.  This is a factorial time algorithm, **O(n!).**  By using dynamic programming, Floyd's algorithm does better.

The problem of finding the shortest path is called **an optimization problem**.  We want to optimize our cost from getting from one vertex to another vertex.

Conceptually, what Floyd's algorithm does, is to start by constructing the shortest path of 1 edge from each vertex to every other vertex.  Then it considers the shortest path of length 2 to be the shortest path of length 1 plus the minimum next edge.   This uses the principle of optimality (the shortest path of length 2, going through A-B-C, must contain the shortest path of length 1 from A to B).  So, we start at the bottom (paths of length 1), and works our way up to the complete solution.

Here's the algorithm: In this algorithm, W is the adjacency matrix for the graph, and D is an array of the same size as the adjacency matrix.

**void floyd (int n, int W [][], int D[][])  
{  
    D = W;**

**for (k = 0; k < n; k++)  
        for (i = 0; i < n; i++)  
            for (j = 0; j < n; j++)  
                D[i][j] = min (D[i][j], D[i][k] + D[k][j]);  
}**

**What is the big-Oh of Floyd's shortest path algorithm?**

Floyd's algorithm works great if all you need is the cost of the shortest path.   Usually, though, we also want to know what the shortest path actually is.  **We can modify Floyd's algorithm to keep track of the shortest paths, also using dynamic programming**.  To do this we add another array, named P.  In this array, we keep track of the last vertex on the shortest path from one vertex to another. For example, if the shortest path from v1 to v4 was v1-v3-v2-v4, then P [1][4] would contain 2, indicating that on the shortest path from v1 to v4, the last vertex before you get to v4 was v2.

A modified version of Floyd's algorithm is as follows:

**void floyd2 (int n, int W [][], int D[][], int P[][])  
{  
    D = W;**

**for (i = 0; i < n; i++)  
        for (j = 0; j < n; j++)  
            P[i][j] = 0;**

**for (k = 0; k < n; k++)  
        for (i = 0; i < n; i++)  
            for (j = 0; j < n; j++)  
                if (D[i][k] + D[k][j] < D[i][j])  
                {  
                    P[i][j] = k + 1;  
                    D[i][j] = D[i][k] + D[k][j];  
                }  
}**

The only difference in this algorithm is that it also modifies the array P.  Once you have array P filled out, how do we figure out what the shortest path was from one vertex to another, say from i to j?

An element in P contains the next to last vertex in the shortest path, say k.  k is the end of a shortest path from i to k.  So, we can define the shortest path recursively.

If we want to find the shortest path between i and j, we can define that as the path from i to k, plus the vertex j.  This is a simple recursive algorithm to print out the shortest path between nodes i and j:

void path (int i, int j, int P[][])  
{  
    if (P[i][j] != 0)  
    {  
        path (i, P[i][j], P);  
        display "v" + j;  
    }  
    else  
    {  
        display "v" + i;  
        display "v" + j;  
    }  
}

This version of the path algorithm is **not the same as the one presented in your book**.

Floyd's algorithm is an excellent example of using **arrays to store intermediate results**, so, **they do not need to be calculated multiple times**.  It is also an excellent example of using the principle of optimality to discard suboptimal solutions.  **Don't forget, however, the principle of optimality only makes sense for some problems.**

## **The Traveling Salesperson Problem**

This problem is perhaps one of the most famous in computer science.  It is used as an example of a certain class of problems that are difficult to solve.

The problem is that of a traveling salesperson who needs to visit every city on their route.  They do not want to waste time, so they need the shortest path that visits each city exactly once, ending back at the salesperson's home city.  In graph terms, we're looking for the shortest path cycle that starts and ends at the salesperson's home city and visits each other city exactly once.  This type of cycle is often called a tour, or a Hamiltonian cycle, or a Hamiltonian circuit. **This problem is always performed on weighted, directed graphs.**

**The brute force approach** would consider every possible path, which would result in **O(n!) time** (for 20 cities, we have 19 possible starting vertices for a path.  If the graph is complete, we have 18 possible second vertices for a path, and so on).  **That is worse than exponential time, and will take much too long for any reasonable size problem.**  However, since this is a shortest path type of problem, the principle of optimality holds and we can use **dynamic programming** to solve the problem.

The dynamic programming approach to the problem ends up being **O(n22n).**   This is better than factorial, **but still worse than exponential**.  But is is useful?  Let's see what the times might be for calculating a tour over 20 cities.  Let's further assume that for the brute force approach, it takes 1 microsecond to calculate each tour.  We assume that it takes 1 microsecond for a program step. In the brute force approach, for 20 cities, there are 19! tours, so the brute force algorithm would take 19! microseconds.  **If you do the math, you end up with 3,857 years**.

In the dynamic programming approach, there are roughly **n22n** program steps.  If n is 20, and each program step takes 1 microsecond, the total time is **419 seconds.**

The times here are not accurate (and your book has another set of times for 20 cities, based on slightly more accurate assumptions), because we made assumptions that the brute force approach is much faster than it really is.  **Even with that assumption, the brute force approach is clearly unsuitable for any but the smallest problems**.  The dynamic programming approach can work for larger numbers of cities, but not very large.   For example, given our assumptions, for 60 cities, the dynamic programming approach would take over 100 million years to run.

A common approach to the traveling salesperson problem is to **redefine the solution**.   Instead of looking for the shortest tour, **we look for a reasonably short tour**.   By relaxing our definition of the solution, the problem can become easier because **we need not find the absolute shortest path, just one that fits some criteria** we define.

Let's look at the dynamic programming approach to the traveling salesperson problem.  We will not look at the algorithm itself, but we'll look at parts of the solution.

Recall that, to use dynamic programming, the principle of optimality must hold.   That is, the optimal solution must consist of optimal solutions to sub-pieces.   If we consider an optimal tour, we can consider it to be the shortext path from vertex 1 to vertex k, plus the shortest path from vertex k to vertex 1 that passes through all the other vertices.

Like all dynamic programming algorithms, we are going to use an array to keep track of the solutions to sub-problems.  In this case, the array will keep track of the shortest paths from one vertex to another vertex that pass through a set of intermediate vertices.  For example, consider the graph where the vertices are the corners of a square:

V(G) = {v1, v2, v3, v4}  
E(G) = {<v1,v2,2>, <v1,v3,9>, <v2,v1,1>, <v2,v3,6>, <v2,v4,4>,<v3,v2,7>, <v3,v4,8>, <v4,v1,6>, <v4,v2,3>}

Our dynamic programming array will keep track of shortest paths.  For example, D[v2][{v4}] should contain the cost of the shortest path from v2 to v1 that passes only through v4.  That path would be v2->v4->v1, and would cost 4 + 6 = 10.   D[v2][{v3,v4}] would contain the cost of the shortest path from v2 to v1 that passes through both v3 and v4.  That cost would be 20.

The traveling salesperson algorithms, like all dynamic programming algorithms, starts with the smallest sub-problems.  In this case, the smallest sub-problem is to calculate the shortest paths from each node back to v1, without going through any other nodes.   For example:

D[v2][{}] = 1  
D[v3][{}] = --  
D[v4][{}] = 6

We can now use those results to help figure the next set of results.  For example:

D[v2][{v3}] = cost (v2, v3) + D[v3][{}] = 6 + -- = --  
D[v2][{v4}] = cost (v2, v4) + D[v4][{}] = 4 + 6 = 10

We continue building this array, using larger and larger sub-problems, until we can fill in the final elements, such as D[v2][{v3, v4}].  That element will be the minimum of: cost (v2, v3) + D[v3][{v4}] and cost (v2, v4) + D[v4][{v3}].  The cost of the shortest tour can then be determined as another minimum, this time taking into account the cost of getting from v1 to the next vertex in the tour (either v2 or v3 in our example) and the cost of getting from the next vertex back to v1 going through all the other vertices.

We keep an array of results so that we need only calculate the shortest path for any given set of nodes exactly once, rather than recalculating it over and over again.