Dynamic Programming

Dynamic programming is a paradigm much like divide and conquer, the problem is split into smaller subproblems which are solved and combined. In a divide and conquer paradigm we create unique subproblems and solve them, however in dynamic programming we solve subproblems which are shared between different subproblems.

Dynamic programming is usually applied to an optimization problem, e.g., maximizing something or minimizing something. These problems have many different solutions, but one of them is an optimal solution.

The steps to creating a dynamic programming algorithm are:

1. Define the structure of an optimal solution
2. Generate the value of the optimal solution recursively
3. Create an optimal solution in a bottom up manner

This is usually hard to understand until we see examples, so I will go over 2 examples

1. Money-board problem

Given an *NxN* board find the path starting from any square at the start, to a target square in row n which maximizes the amount of money that can be obtained. The person can only move forward, diagonally or straight.

Let's assume we have a 4x4 board (Figure 1):

4				
3		X	X	X
2			O	
1				
	1	2	3	4

- Numbers represent index of each row and column.
- O represents our current square.
- X represents the squares we can travel to.
- Cash [*r*, *c*] returns the cash amount on square (*r*, *c*)
- T represents our target square

So how do we solve this? We could use brute force by generating all possible paths and find the path which has the most money but, there are 3^N possibilities which is too slow in cases when *N* gets large. So we will use dynamic programming.

Step 1 - Define the optimal structure for the problem.

As shown in figure 1 that we can move to 3 squares above our current, which means that we can also say that the maximum amount for any square is the maximum amount of the 3 squares below it plus the amount in the current square.

Therefore we can say that the best path to (r, c) is one of

- The best way through (r – 1, c - 1) -- The bottom left square
- The best way through (r – 1, c) -- The bottom square
- The best way through (r – 1, c + 1) -- The bottom right square

If r = 1 then we know that we are at the start and there is no best way to that square. This gives us enough information to create a recursive solution.

Step 2 - Solve the problem recursively

Let C (r, c) be a function which returns the maximum amount of cash that a person can pick up by going to the specified row and column
Knowing the optimal structure, we go to 2nd step of creating a dynamic programming algorithm, creating a recursive function to solve the problem.

$$C(r, c) = \begin{cases} -\infty & \text{If } c < 1 \text{ or } c > n \\ \text{Cash}[r, c] & \text{If } r = 1 \\ \max(C(r-1, c-1), C(r-1, c), C(r-1, c+1)) + \text{Cash}[r, c] & \text{Otherwise} \end{cases}$$

Pseudo Code for the recursive algorithm.

```
function C (r, c)
   if c < 1 or c > N then
      return -∞
   else if r = 1 then
      return Cash [r, c]
   else
      max (C (r - 1, c - 1), C (r - 1, c), C (r - 1, c + 1)) + Cash [r, c]
end
```

This algorithm works however it has a problem; many of the same subproblems are solved multiple times. Fortunately we can solve this problem by calculating upwards starting from row 1.

Step 3 – Generating a solution bottom up

Instead of C being a function, we will use it as a 2D array to store the maximum amount of money that can be picked up by going to (r, c).
In our recursive algorithm we see that for all for values of $r \geq 2$, we need the value of r – 1. This allows us to start at $r = 2$ and go up the money board.

Pseudo Code for the bottom up solution

```
function FindMaximumAmount
   for i : 1 to N
      C [1, i] = Cash[1, i]
      C [i, 0] = -∞
      C [i, N + 1] = -∞
   end
   for r : 2 to N
      for c : 1 to N
         C [r, c] = max (C [r - 1, c – 1], C [r - 1, c], C [r - 1, c + 1]) + Cash[r, c]
      end
   end
end
```

Example 2 – Coin Change Problem

Given N amount of denominations d [N], determine the minimum amount of change required to make change for and amount A. We could use brute force and generate all possibilities however the running time for the algorithm would be exponential. Since this problem has an optimal structure and shared subproblems we will use dynamic programming.

Step 1 - Define the optimal structure for the problem.

In a solution to make change for A cents, we can say that for a value C and $C \leq A$, we can split the optimal solution into C cents and $A - C$ cents. The solution to C cents must be optimal because if there was a way to generate change using fewer coins to make C cents we would replace that method with the better to generate C cents. This is the same for $A - C$ cents.

Step 2 - Solve the problem recursively

Let MC (A) be a function which returns the minimum number of coins needed to make change for A cents. As we said in the optimal structure there exists a value C which splits the amount of coins optimally between 0..C, C+1 .. A. Let C be a value of d[k]. If d[k] is the optimal value then we can say that MC (A) = 1 + MC (A – d[k]).

The problem is that we do not know for which value of k gives the optimal value so we will check all possibilities for k. Also note that if we need to make change for 0 cents we need 0 cents so we can define the following recursive function.

$$MC(A) = \begin{cases} 0 & \text{If } A = 0 \\ \text{Min for all values of k } (1 + MC(A - d[k])) & \text{If } A > 0 \end{cases}$$

Pseudo Code for the recursive algorithm.

```
function MC (A)
   if A = 0 then
      return 0
   else
      minimum = ∞
      for k : 1 to N
         if (d[k] ≤ A) then
            coins = 1 + MC (A – d[k])
            if (coins < minimum) then
               minimum = coins
            end
         end
      end
      return minimum
   end
end
```

Step 3 – Generating a solution bottom up

If you tried the recursive algorithm you will realize that it takes a long time to solve this problem. This is because many subproblems are solved more then once. Fortunate if we start from A = 0 and go up, we are able to compute the solution in O (A*k) time. If we calculate the optimal number of coins needed for each integer amount that that comes inside the range of 1 to *A*, we are able to create a table which has the optimal solution for each amount between 1 to *A*. Let MC be a 1-D array which stores the minimum coin needed at each amount

Pseudo Code for the bottom up solution

```
function CalculateMinimumCoins
   MC [0] = 0
   for i : 1 to A
       minimum = ∞
       for k : 1 to N
          if (d[k] ≤ i) then
             if (1 + MC [i – d[k]] < minimum) then
             minimum = 1 + MC [i – d[k]]
          end
       end
       MC [i] = minimum
    end
end
```

In conclusion dynamic programming is a very effective tool in solving problems that have an optimal structure and have shared subproblems. Dynamic programming can be difficult to master, but with more examples it becomes easier to see how to apply the dynamic programming rules to a problem.

Refrences

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, 2001. *Introduction to Algorithms*, 2nd ed. MIT Press & McGraw-Hill.
- "Dynamic Programming." Wikipedia. 3 Jan 2008 <http://en.wikipedia.org/wiki/Dynamic_Programming>.