Chapter 4

Dynamic Programming

Dynamic Programming: technique is among the most powerful for designing algorithms for optimization problems. Dynamic programming problems are typically optimization problems (find the minimum or maximum cost solution, subject to various constraints). The technique is related to divide-and-conquer, in the sense that it breaks problems down into smaller problems that it solves recursively. However, because of the somewhat different nature of dynamic programming problems, standard divide-and-conquer solutions are not usually efficient. The basic elements that characterize a dynamic programming algorithm are:

Substructure: Decompose your problem into smaller (and hopefully simpler) subproblems. Express the solution of the original problem in terms of solutions for smaller problems.

Table-structure: Store the answers to the sub-problems in a table. This is done because subproblem solutions are reused many times.

Bottom-up computation: Combine solutions on smaller subproblems to solve larger subproblems. (We also discusses a top-down alternative, called memoization)

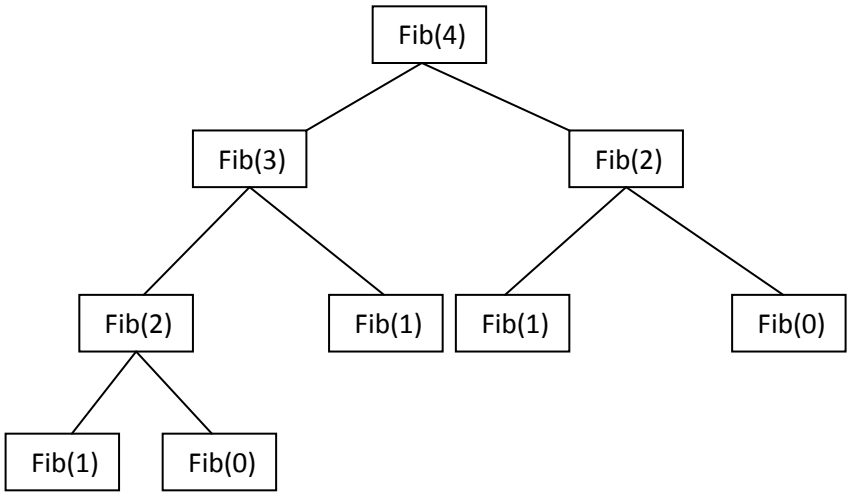
The most important question in designing a DP solution to a problem is how to set up the subproblem structure. This is called the formulation of the problem. Dynamic programming is not applicable to all optimization problems. There are two important elements that a problem must have in order for DP to be applicable.

Optimal substructure: (Sometimes called the principle of optimality.) It states that for the global problem to be solved optimally, each subproblem should be solved optimally. (Not all optimization problems satisfy this. Sometimes it is better to lose a little on one subproblem in order to make a big gain on another.)

Polynomially many subproblems: An important aspect to the efficiency of DP is that the total number of subproblems to be solved should be at most a polynomial number.

Fibonacci numbers

Recursive Fibonacci revisited: In recursive version of an algorithm for finding Fibonacci number we can notice that for each calculation of the Fibonacci number of the larger number we have to calculate the Fibonacci number of the two previous numbers regardless of the computation of the Fibonacci number that has already be done. So there are many redundancies in calculating the Fibonacci number for a particular number. Let's try to calculate the Fibonacci number of 4. The representation shown below shows the repetition in the calculation.



In the above tree we saw that calculations of fib(0) is done two times, fib(1) is done 3 times, fib(2) is done 2 times, and so on. So if we somehow eliminate those repetitions we will save the running time.

Algorithm:

```
DynaFibo(n)
{
        A[0] = 0, A[1]= 1;
        for(i = 2 ; i <=n ; i++)
        A[i] = A[i-2] +A[i-1] ;
        return A[n] ;
}
```

Analysis

Analyzing the above algorithm we found that there are no repetition of calculation of the subproblems already solved and the running time decreased from $O(2^{n/2})$ to O(n). This reduction was possible due to the remembrance of the subproblem that is already solved to solve the problem of higher size.

0/1 Knapsack Problem

Statement: A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of i^{th} item is w_i and it worth v_i. An amount of item can be put into the bag is 0 or 1 i.e. x_i is 0 or 1. Here the objective is to collect the items that maximize the total profit earned.

We can formally state this problem as, maximize $\sum_{i=1}^{n} x_i v_i$ Using the constraints

$\sum_{i=1}^{n} x_i w_i \leq W$

The algorithm takes as input maximum weight W, the number of items n, two arrays v[] for values of items and w[] for weight of items. Let us assume that the table c[i,w] is the value of solution for items 1 to i and maximum weight w. Then we can define recurrence relation for 0/1 knapsack problem as

$$C[i,w] = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0 \\ C[i-1,w] & \text{if } wi > w \\ \text{Max}\{vi + C[i-1,w-wi], c[i-1,w] & \text{if } i>0 \text{ and } w>wi \end{cases}$$

```
DynaKnapsack(W,n,v,w)
{
        for(w=0; w<=W; w++)
                C[0,w] = 0;
        for(i=1; i<=n; i++)
                C[i,0] = 0;
        for(i=1; i<=n; i++)
        {
                for(w=1; w<=W;w++xw1z)
                {
                        if(w[i]<w)
```

```
                    {
                        if v[i] +C[i-1,w-w[i]] > C[i-1,w]
                        {
                            C[i,w] = v[i] +C[i-1,w-w[i]];
                        }
                        else
                        {
                            C[i,w] = C[i-1,w];
                        }
                    }
                    else
                    {
                        C[i,w] = C[i-1,w];
                    }
                }
            }
        }
```

Analysis

For run time analysis examining the above algorithm the overall run time of the algorithm is O(nW).

Example

Let the problem instance be with 7 items where v[] = {2,3,3,4,4,5,7}and w[] = {3,5,7,4,3,9,2}and W = 9.

w / i	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2	2
2	0	0	0	2	2	3	3	3	5	5
3	0	0	0	2	2	3	3	3	5	5
4	0	0	0	2	4	4	4	6	6	7
5	0	0	0	4	4	4	6	8	8	8
6	0	0	0	4	4	4	6	8	8	8
7	0	0	7	7	7	11	11	11	13	15

.

Profit= C[7][9]=15

Matrix Chain Multiplication

Chain Matrix Multiplication Problem: Given a sequence of matrices $A_1;A_2; : : : ; A_n$ and dimensions $p_0; p_1; : : : ; p_n$, where A_i is of dimension p_{i-1} x p_i, determine the order of multiplication that minimizes the number of operations.

Important Note: This algorithm does not perform the multiplications, it just determines the best order in which to perform the multiplications.

Although any legal parenthesization will lead to a valid result, not all involve the same number of operations. Consider the case of 3 matrices: A1 be 5 x 4, A2 be 4 x 6 and A3 be 6 x 2.

multCost[((A1A2)A3)] = (5 . 4 . 6) + (5 . 6 . 2) = 180

multCost[(A1(A2A3))] = (4 . 6 . 2) + (5 . 4 . 2) = 88

Even for this small example, considerable savings can be achieved by reordering the evaluation sequence.

Let $A_{i....j}$ denote the result of multiplying matrices i through j. It is easy to see that $A_{i...j}$ is a p_{i-1} x p_j matrix. So for some k total cost is sum of cost of computing $A_{i...k}$, cost of computing $A_{k+1...j}$, and cost of multiplying $A_{i...k}$ and $A_{k+1...j}$.

Recursive definition of optimal solution: let m[j,j] denotes minimum number of scalar multiplications needed to compute $A_{i...j}$.

$$C[i,w] = \begin{cases} 0 & \text{if } i=j \\ \min_{i \le k < j}\{m[I,k] + m[k+1,j] + p_{i-1}p_k p_j & \text{if } i<j \end{cases}$$

```
Matrix-Chain-Multiplication(p)
{
        n =length[p]
        for( i= 1 i<=n i++)
        {
                m[i, i]= 0
        }
    for(l=2; l<= n; l++)
    {
        for( i= 1; i<=n-l+1; i++)
        {
                j = i + l − 1
            m[i, j] = ∞
            for(k= i; k<= j-1; k++)
            {
                        c= m[i, k] + m[k + 1, j] + p[i−1] * p[k] * p[j]
                        if c < m[i, j]
                        {
                                m[i, j] = c
                                s[i, j] = k
                        }
            }
        }
    }
return m and s
}
```

Analysis

The above algorithm can be easily analyzed for running time as $O(n^3)$, due to three nested loops.

The space complexity is $O(n^2)$.

Example:

Consider matrices A1, A2, A3 And A4 of order 3x4, 4x5, 5x2 and 2x3.

M Table(Cost of multiplication)

i \ j	1	2	3	4
1	0	60	64	82
2		0	40	64
3			0	30
4				0

S Table (points of parenthesis)

i \ j	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

Constructing optimal solution

(A1A2A3A4) => ((A1A2A3)(A4)) => (((A1)(A2A3))(A4))

Longest Common Subsequence Problem

Given two sequences X = (x_1, x_2, ……,x_m) and Z = (z_1; z_2; …… ; z_k), we say that Z is a subsequence of X if there is a strictly increasing sequence of k indices (i_1, i_2, ... i_k) ($1 \leq i_1 < i_2 <$ …….. < i_k) such that Z = (X_{i1},X_{i2} ..... X_{ik}).

For example, let X = (ABRACADABRA) and let Z = (AADAA), then Z is a subsequence of X. Given two strings X and Y , the longest common subsequence of X and Y is a longest sequence Z that is a subsequence of both X and Y . For example, let X = (ABRACADABRA) and let Y = (YABBADABBAD). Then the longest common subsequence is Z = (ABADABA)

The Longest Common Subsequence Problem (LCS) is the following. Given two sequences X = (x1; : : : ; x_m) and Y = (y, ……., y_n) determine a longest common subsequence.

DP Formulation for LCS:

Given a sequence $X = <x_1, x_2, \ldots, x_m>$, $X_i = <x_1, x_2, \ldots, x_i>$ is called i^{th} prefix of X, here we have X_0 as empty sequence. Now in case of sequences X_i and Y_j:

If $x_i = y_j$ (i.e. last Character match) , we claim that the LCS must also contail charater x_i or y_j .

If $x_i \neq y_j$ (i.e Last Character do not match) , In this case x_i and y_j cannot both be in the LCS (since they would have to be the last character of the LCS). Thus either x_i is not part of the LCS, or y_j is not part of the LCS (and possibly both are not part of the LCS). Let L[I,j] represents the length of LCS of sequences X_i and Y_j.

$$L[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ L[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{L[i-1,j], L[i,j-1]\} & \text{if } i>0 \text{ and } w>wi \end{cases}$$

```
LCS(X,Y)
{
        m = length[X];
        n = length[Y];
        for(i=1;i<=m;i++)
                c[i,0] = 0;
        for(j=0;j<=n;j++)
                c[0,j] = 0;
        for(i = 1;i<=m;i++)
        for(j=1;j<=n;j++)
        {
                if(X[i]==Y[j])
                {
                        c[i][j] = c[i-1][j-1]+1; b[i][j] = "upleft";
                }
                else if(c[i-1][j]>= c[i][j-1])
```

```
        {
            c[i][j] = c[i-1][j]; b[i][j] = "up";
        }
        else
        {
        c[i][j] = c[i][j-1]; b[i][j] = "left";
        }
    }
    return b and c;
}
```

Analysis:

The above algorithm can be easily analyzed for running time as O(mn), due to two nested loops.

The space complexity is O(mn).

Example:

Consider the character Sequences X=abbabba Y=aaabba

Y / X	Φ	a	A	a	b	b	a
Φ	0	0	0	0	0	0	0
a	0	1 upleft	1upleft	1upleft	1 left	1 left	1 left
b	0	1 up	1 up	1 up	2 upleft	2 upleft	2 left
b	0	1 up	1 up	1 up	2 upleft	3 upleft	3 left
a	0	1 upleft	2 upleft	2 upleft	2 up	3 up	4 upleft
b	0	1 up	2 up	2 up	3 upleft	3 upleft	4 up
b	0	1 up	2 up	2 up	3 upleft	4 upleft	4 up
a	0	1 upleft	2 upleft	3 upleft	3 up	4 up	5 upleft

LCS = aabba