DYNAMIC PROGRAMMING

(Part-2)

In this document we will be mainly focussing on two dimensional dynamic programming. I suggest the reader to go through and practise problems from the first document on dynamic programming before reading further. As we already saw that 1-d dp involves only one dp state to make a relation between current and the previous subproblems. As the name suggests in 2-d dp will use two dp states to form the relation. Let us understand this with a simple example.

EXAMPLE- 1: Let's consider we have a matrix of n rows and m columns, where each cell has a cost associated with it $cost[i][j]$. You start from cell $(0,0)$ and your destination is $(n-1, m-1)$. At every step you can go either towards right i.e. (i,j) to $(i,j+1)$ or you can go down i.e. (i,j) to $(i+1,j)$. Every time you reach a cell, you need to pay the corresponding cost associated with it. Of course at any point you cannot leave the matrix. Your task is to find out the minimum cost possible to reach from the source to the destination.

Solution: Before reading the solution I urge you to think about it for a while and try to come up with the solution.
Let's consider we are standing at the cell (i,j) and we need to find the minimum cost needed to reach the destination if we start at cell (i,j). We denote this with $dp[i][j]$. As stated in the problem statement, we can go either towards right or down, which means if we go towards right, the minimum cost needed to reach the destination is nothing but $cost[i][j] + dp[i][j+1]$. Similarly if we go down, then the minimum cost will be $cost[i][j] + dp[i+1][j]$. Thus, the relation established is

$$dp[i][j] = cost[i][j] + min(dp[i+1][j], dp[i][j+1])$$

Of course you need to take care when you are at the boundary cells, where you don't have the choice of going in both the directions. Thus, our final answer will be $dp[0][0]$ which is the minimum cost to reach the destination from the source cell. Remember, while finding $dp[i][j]$ you have to make sure that you have already computed $dp[i+1][j]$ and $dp[i][j+1]$. Thus the order of traversal for computing $dp[i][j]$ matters. The order in which you should compute is left as a task for the reader. This algorithm runs in $O(n * m)$ time.

Hopefully the above example might have given you some insight about 2-d dp. One of the most crucial part of solving any dp problem is to decide the dp states. Once you have correctly decided that, half of your problem is done. Let's consider one more example. This is a classical dp problem called **subset sum problem**.

EXAMPLE- 2: You have an array of length N. You are given an integer k and you need to find if there exists a subset of elements whose sum is k. The elements of the array are non-negative.

Solution: Let's choose our dp states to be the index i of the array where we are considering that our subset will have elements with index $<= i$ and j which denotes the sum for which we need to check. More formally, let's consider a 2-d boolean array $dp[i][j]$ which is true if there exists a subset in the first $i+1$ elements(elements with index $<= i$) and having a sum j. Now, if $arr[i] = j$, then of course $dp[i][j] = true$. Also if we choose not to take $arr[i]$ in our subset then if $dp[i-1][j]$ is true then $dp[i][j]$ is also true. Else the subset (considering $arr[i]$ to be a part of this subset) will also have previous elements (elements with index less than i). Now, since $arr[i]$ will always be a part of that subset we can check if there exists a subset whose indices are $<= i-1$ and sum is $j - arr[i]$, since if there exists such a subset we will add $arr[i]$ to that subset and get the desired sum j. Hence, we get this relation

$if(arr[i] == j)\ dp[i][j] = true;$
$else\ if(dp[i-1][j-arr[i]] == true)\ dp[i][j] = true;$
$else\ if\ (dp[i-1][j] == true)\ dp[i][j] = true;$
$else\ dp[i][j] = false;$

This algorithm runs in $O(N * k)$ time.

Below are some links of some useful articles and some problems for practise.

- Subset sum problem: geeksforgeeks and video tutorial.
- Knapsack problem: geeksforgeeks and video tutorial.
- Matrix chain multiplication: geeksforgeeks and video tutorial.
- Longest common subsequence: geeksforgeeks and video tutorial.

Practise problems:

- Problem-1
- Problem-2
- Problem-3
- Problem-4