Dynamic programming Strategies

Dynamic **P**rogramming (DP) is a technique that solves some particular type of problems in Polynomial Time. Dynamic Programming solutions are faster than exponential brute method and can be easily proved for their correctness. Before we study how to think Dynamically for a problem, we need to learn:

1. Overlapping Subproblems.
2. Optimal Substructure Property.

Both above points 1 and 2 we have already discus in introduction part of **Dynamic programming.**

Steps to solve a DP

1) Identify if it is a DP problem
2) Decide a state expression with
   least parameters
3) Formulate state relationship
4) Do tabulation (or add memoization).

Step 1 : How to classify a problem as a Dynamic Programming Problem?

- Typically, all the problems that require to maximize or minimize certain quantity or counting problems that say to count the arrangements under certain condition or certain probability problems can be solved by using Dynamic Programming.
- All dynamic programming problems satisfy the overlapping subproblems property and most of the classic dynamic problems also satisfy the optimal substructure property. Once, we observe these properties in a given problem, be sure that it can be solved using DP.

Step 2 : Deciding the state

DP problems are all about state and their transition. This is the most basic step which must be done very carefully because the state transition depends on the choice of state definition you make. So, let's see what do we mean by the term "state".

State A state can be defined as the set of parameters that can uniquely identify a certain position or standing in the given problem. This set of parameters should be as small as possible to reduce state space.

For example: In our famous Knapsack problem, we define our state by two parameters **index** and **weight** i.e DP[index][weight]. Here DP[index][weight] tells us the maximum profit it can make by taking items from range 0 to index having the capacity of sack to

be weight. Therefore, here the parameters **index** and **weight** together can uniquely identify a subproblem for the knapsack problem.

So, our first step will be deciding a state for the problem after identifying that the problem is a DP problem.

As we know DP is all about using calculated results to formulate the final result.

So, our next step will be to find a relation between previous states to reach the current state.

Step 3 : Formulating a relation among the states

This part is the hardest part of for solving a DP problem and requires a lot of intuition, observation and practice. Let's understand it by considering a sample problem

Given 3 numbers {1, 3, 5}, we need to tell
the total number of ways we can form a number 'N'
using the sum of the given three numbers.
(allowing repetitions and different arrangements).

Total number of ways to form 6 is: 8
1+1+1+1+1+1
1+1+1+3
1+1+3+1
1+3+1+1
3+1+1+1

3+3

1+5

5+1

Let’s think dynamically about this problem. So, first of all, we decide a state for the given problem. We will take a parameter n to decide state as it can uniquely identify any subproblem. So, our state dp will look like state(n). Here, state(n) means the total number of arrangements to form n by using {1, 3, 5} as elements.

Now, we need to compute state(n).

How to do it?

So here the intuition comes into action. As we can only use 1, 3 or 5 to form a given number. Let us assume that we know the result for n = 1,2,3,4,5,6 ; being termilogistic let us say we know the result for the

state (n = 1), state (n = 2), state (n = 3) …… state (n = 6)

Now, we wish to know the result of the state (n = 7). See, we can only add 1, 3 and 5. Now we can get a sum total of 7 by the following 3 ways:

1) Adding 1 to all possible combinations of state (n = 6)

Eg : [(1+1+1+1+1+1) + 1]

[(1+1+1+3) + 1]

[(1+1+3+1) + 1]

[(1+3+1+1) + 1]

[(3+1+1+1) + 1]

[(3+3) + 1]

[(1+5) + 1]

[(5+1) + 1]

2) Adding 3 to all possible combinations of state (n = 4);

Eg : [(1+1+1+1) + 3]

[(1+3) + 3]

[(3+1) + 3]

3) Adding 5 to all possible combinations of state(n = 2)

Eg : [(1+1) + 5]

Now, think carefully and satisfy yourself that the above three cases are covering all possible ways to form a sum total of 7;

Therefore, we can say that result for

state(7) = state (6) + state (4) + state (2)

or

state(7) = state (7-1) + state (7-3) + state (7-5)

In general,

state(n) = state(n-1) + state(n-3) + state(n-5)

So, our code will look like:

```
/ Returns the number of arrangements to
/ form 'n'
nt solve(int n)

/ base case
f (n < 0)
   return 0;
f (n == 0)
   return 1;

eturn solve(n-1) + solve(n-3) + solve(n-5);
```

The above code seems exponential as it is calculating the same state again and again. So, we just need to add a memoization.

Step 4 : Adding memoization or tabulation for the state

This is the easiest part of a dynamic programming solution. We just need to store the state answer so that next time that state is required, we can directly use it from our memory

Adding memoization to the above code

```
// initialize to -1
int dp[MAXN];

// this function returns the number of
// arrangements to form 'n'
int solve(int n)
{
// base case
if (n < 0)
    return 0;
if (n == 0)
    return 1;

// checking if already calculated
if (dp[n]!=-1)
    return dp[n];

// storing the result and returning
return dp[n] = solve(n-1) + solve(n-3) + solve(n-5);
}

```
