NOTES

Dynamic programming

Dynamic programming works on idea that if you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again. If the given problem can be broken up in to smaller sub-problems and these smaller subproblems are in turn divided into still-smaller ones, and in this process, if you observe some over-lapping subproblems, then it is a big hint for DP. Also, the optimal solutions to the subproblems contribute to the optimal solution of the given problem

Recursion uses the top-down approach to solve the problem i.e. It begin with core (main) problem then breaks it into subproblems and solve these subproblems similarily. In this approach same subproblem can occur multiple times and consume more CPU cycle, hence increase the time complexity. Whereas in Dynamic programming same subproblem will not be solved multiple times but the prior result will be used to optimise the solution. Dynamic algorithms use Memorization.

Dynamic Programming in simple language is-

Writes down "1+1+1+1+1+1+1+1 =" on a sheet of paper.
"What's that equal to?"
Counting "Eight!"
Writes down another "1+" on the left.
"What about that?"
"Nine!" " How'd you know it was nine so fast?"
"You just added one more!"
"So you didn't need to recount because you remembered there were eight! Dynamic Programming is just a fancy way to say remembering stuff to save time later!"

Let's try to understand this by taking an example of **Fibonacci numbers.**

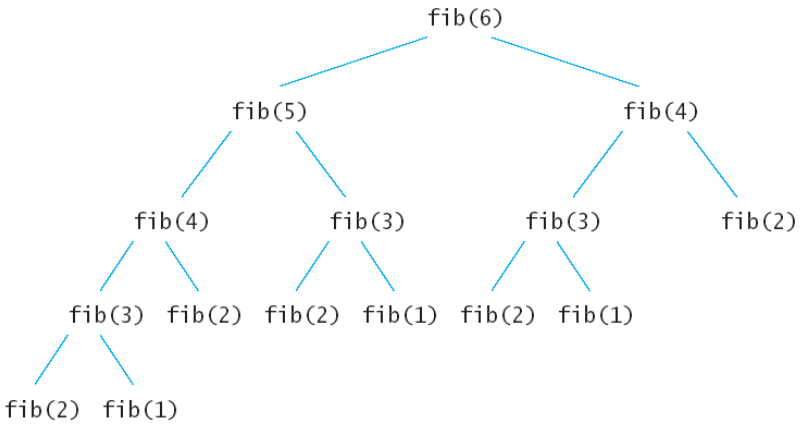
Fibonacci (n) = 1; if n = 0
Fibonacci (n) = 1; if n = 1
Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

So, the first few numbers in this series will be: 1, 1, 2, 3, 5, 8, 13, 21... and so on!

A code for it using pure recursion:

```
def fib (n) :
        if n < 2:
                return 1;
        return fib(n-1) + fib(n-2);
print(fib(5))
```

In the recursive code, a lot of values are being recalculated multiple times. We could do better with calculating each unique quantity only once. Take a look at the image to understand that how certain values were being recalculated in the recursive way:



Using Dynamic Programming approach with memorization:

```
fibresult = [0]*2
def fib (n):
        fibresult[0] = 1;
        fibresult[1] = 1;
        for i in range(2,n):
                fibresult.append(fibresult[i-1] + fibresult[i-2])
        return fibresult

print(fib(5))
```

The intuition behind dynamic programming is that we **trade space for time**, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

There are following two different ways to store the values so that the values of a sub-problem can be reused. Here, will discuss two patterns of solving DP problem:

1. **Tabulation: Bottom Up**
   Let's describe a state for our DP problem to be dp[x] with dp[0] as base state and dp[n] as our destination state. So, we need to find the value of destination state i.e. dp[n]. If we start our transition from our base state i.e dp[0] and follow our state transition relation to reach our destination state dp[n], we call it Bottom Up approach as it is quite clear that we started our transition from the bottom base state and reached the top most desired state.

   Let us take example of factorial

   ```
   fact = [0]*51;

   def factorial(n):
      for i in range(1,n+1):
         fact[i] = fact[i-1] * i
      return fact[n]

   fact[0] = 1
   print(5,"! is ", factorial(5), sep = '')
   print(10,"! is ", factorial(10), sep = '')
   ```

2. **Memorization: Top Down**
   In this approach, we start our journey from the top most destination state and compute its answer by taking in count the values of states that can reach the destination state, till we reach the bottom most base state. If we need to find the value for some state say dp[n] and instead of starting from the base state that i.e. dp[0] we ask our answer from the states that can reach the destination state dp[n] following the state transition relation, then it is the top-down fashion of DP.

   ```
   fact = [-1]* 50

   def factorial(x):
      if x==0:
         return 1;
      if fact[x]!=-1:
         return fact[x];
      fact[x] = x * factorial(x-1)
      return fact[x]


   print(5,"! is ", factorial(5), sep = '')
   print(10,"! is ", factorial(10), sep = '')
   ```

Comparison of both approaches

Tabulation	Memorization
Fast, because direct access of previous state is available	Slow, because we have recursion and return statements.
Better approach if all sub problem has to solved at least once	if some sub problems has to be solved instead of all then it offers advantage than tabulation
All entries in the table are filled one by one	All entries in the table may not fill because entries are filled on demand basis
Code may be complicated when lot of conditions are there	Code is comparatively less complicated
State transition relation is difficult to identify	State transition relation is easy to identify

Dynamic Programming is a powerful technique that allows one to solve different types of problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time. Before we study how to think dynamically for a problem, we need to learn:

1. **Overlapping Sub-problems:** In dynamic programming, computed solutions to sub-problems are stored in a table so that these don't have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) sub-problems because there is no point storing the solutions if they are not needed again.

2. **Optimal Substructure:** A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its sub-problems. In other words **Principle of Optimality** is based on the idea that in order to solve a dynamic optimization problem from some starting period t to some ending period T, one implicitly has to solve sub-problems starting from later dates s, where t<s<T. This is an example of optimal substructure.

How to solve a Dynamic Problem

Although different authors suggest different ways to solve a dynamic problem but more or less they all are saying some common things like find the solution, Analyze the solution means does it have optimal substructure and overlapping sub-problem, after doing that optimize the solution. Basically it is all about your practice on divide and conquer and optimization. The more practice you do the easier it will be.

Problem: Consider a game where a player can score 3 or 5 or 10 points at a time. Given a total score n, find the number of ways to reach the given score. The order of scoring does not matter.

Sample Input: n = 20
output: 4
Explanation: These are the following 4 ways to reach 20:(10, 10) (5, 5, 10) (5, 5, 5, 5) (3, 3, 3, 3, 3, 5)

Sample Input: n = 13
output: 2
Explanation: These are the following 2 ways to reach 13:(10, 3) (5, 5, 3)

Solution: let's assume that the array S contains the scores given and n be the total given score. For example, S = {3, 5, 10} and n can be 20, which means that we need to find the number of ways to reach the score 20 where a player can score either score 3, 5 or 10.

Optimal Sub-structure:
To count the total number of solutions, we can divide all set solutions into two sets.
1) Solutions that do not contain the i^{th} index score (or S[i]).
2) Solutions that contain at least one i^{th} index score.

Let count(S[], m, n) be the function to count the number of solutions where: m is the index of the last score that we are examining in the given array S, and n is the total given score. It can be written as the sum of count(S[], m-1, n) and count(S[], m, n-S[m]), which is nothing but the sum of solutions that do not contain the m^{th} score count(S[], m-1, n) and solutions that contain at least one m^{th} score count(S[], m, n-S[m]). Therefore, the problem has optimal substructure property as the problem can be solved using solutions to subproblems.

Overlapping Subproblems
We can fin solution of the problem recursion and we need to apply the solution again and again for subset also. Look at the code here

```
# Returns number of ways to reach score n.
def count(n):

    # table[i] will store count of solutions for value i.
    # Initialize all table values as 0.
    table = [0 for i in range(n+1)]

    # Base case (If given value is 0)
    table[0] = 1

    # One by one consider given 3 moves and update the
    # table[] values after the index greater than or equal
    # to the value of the picked move.
    for i in range(3, n+1):
        table[i] += table[i-3]
    for i in range(5, n+1):
        table[i] += table[i-5]
    for i in range(10, n+1):
        table[i] += table[i-10]

    return table[n]

# Driver Program
n = 20
print('Count for', n, 'is', count(n))

n = 13
print('Count for', n, 'is', count(n))
```

Output

```
Count for 20 is 4
Count for 13 is 2
```