DP
Lesson 1

1. **Overlapping Subproblems**
2. **Optimal Substructure**
3. **Memoization Method**
4. **Tabulation Method**

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.
Following are the two main properties of a problem that suggests that the given problem can be solved using Dynamic programming

Overlapping Subproblems

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of the same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems 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) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn't have common subproblems. If we take an example of the following recursive program for Fibonacci Numbers, there are many subproblems that are solved again and again.

```
int fib(int n)
{
        if (n <= 1)
                return n;
        return fib(n - 1) + fib(n - 2);
}
```

```
                         fib(5)
                     /             \
               fib(4)                fib(3)
             /        \              /     \
         fib(3)      fib(2)         fib(2)    fib(1)
        /     \        /    \       /    \
  fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
  /    \
fib(1) fib(0)
```

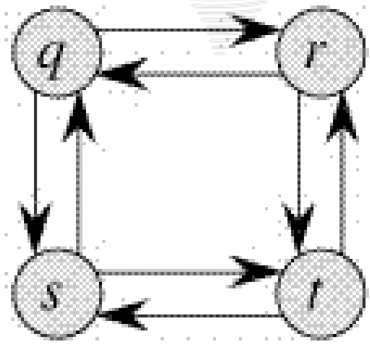
Optimal Substructure

A given problem has an Optimal Substructure Property if an optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

For example, the Shortest Path problem has the following optimal substructure property:

If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. The standard All Pair Shortest Path algorithm like Floyd–Warshall and Single Source Shortest path algorithm for negative weight edges like Bellman–Ford are typical examples of Dynamic Programming.

On the other hand, the Longest Path problem doesn't have the Optimal Substructure property. Here by Longest Path we mean the longest simple path (path without cycle) between two nodes. There are two longest paths from q to t: q→r→t and q→s→t. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path q→r→t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q→s→t→r and the longest path from r to t is r→q→s→t.



Memoization Method
Top Down Dynamic Programming

The memoized program for a problem is similar to the recursive version with a small modification that looks into a lookup table before computing solutions. Whenever we need the solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

Here, we start our journey from the top most destination state and compute its answer by taking into count the values of states that can reach the destination state, till we reach the bottom most base state.

In this case the memory layout is linear, that's why it may seem that the memory is being filled in a sequential manner like the tabulation method, but you may consider any other top down DP having a 2D memory layout, here the memory is not filled in a sequential manner.

```
#include <bits/stdc++.h>
using namespace std;
#define NIL -1
#define MAX 100

int lookup[MAX];

/* Function to initialize NIL values in lookup table */
void _initialize()
{
        int i;
        for (i = 0; i < MAX; i++)
                lookup[i] = NIL;
}

/* function for nth Fibonacci number */
int fib(int n)
```

```
{
    if (lookup[n] == NIL) {
        if (n <= 1)
            lookup[n] = n;
        else
            lookup[n] = fib(n - 1) + fib(n - 2);
    }

    return lookup[n];
}

// Driver code
int main()
{
    int n = 40;
    _initialize();
    cout << "Fibonacci number is " << fib(n);
    return 0;
}
```

Tabulation Method
Bottom Up Dynamic Programming

The tabulated program for a given problem builds a table in bottom-up fashion and returns the last entry from the table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3), and so on. So literally, we are building the solutions of subproblems bottom-up.

As the name itself suggests starting from the bottom and accumulating answers to the top.

Now, Why do we call it a tabulation method?

To know this, let's first write some code to calculate the factorial of a number using the bottom up approach. Once, again as our general procedure to solve a DP we first define a state. In this case, we define a state as dp[x], where dp[x] is to find the factorial of x.

Now, it is quite obvious that dp[x+1] = dp[x] * (x+1)

```
// Tabulated version to find factorial x.
int dp[MAXN];

// base case
int dp[0] = 1;
for (int i = 1; i< =n; i++)
{
   dp[i] = dp[i-1] * i;
}
```

The above code clearly follows the bottom-up approach as it starts its transition from the bottom-most base case dp[0] and reaches its destination state dp[n]. Here, we may notice that the dp table is being populated sequentially and we are directly accessing the calculated states from the table itself and hence, we call it a tabulation method.

```
#include <stdio.h>
int fib(int n)
{
        int f[n + 1];
        int i;
        f[0] = 0;
        f[1] = 1;
        for (i = 2; i <= n; i++)
                f[i] = f[i - 1] + f[i - 2];

        return f[n];
}

int main()
{
        int n = 9;
        printf("Fibonacci number is %d ", fib(n));
        return 0;
}
```

	Tabulation	Memoization
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous states from the table	Slow due to lot of recursive calls and return statements
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand.

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

- Typically, all the problems that require maximizing or minimizing certain quantities or counting problems that say to count the arrangements under certain conditions 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.

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 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 the 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 the 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'
int solve(int n)
{
// base case
if (n < 0)
        return 0;
if (n == 0)
        return 1;

return 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 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

```
int dp[MAXN];
int solve(int n)
{if (n < 0)
        return 0;
if (n == 0)
        return 1;
 if (dp[n]!=-1)
        return dp[n];
 return dp[n] = solve(n-1) + solve(n-3) + solve(n-5);
}
```

Print first n Fibonacci Numbers