Those Who Cannot Remember the Past
Are Condemned to Repeat it…
----DP

Dynamic Programming:

- Dynamic programming is a technique for solving problems by breaking them down into smaller subproblems and storing the solutions to these subproblems to avoid redundant work. It is a powerful tool for solving problems that can be divided into smaller subproblems that are similar in structure, and can be used to solve a wide range of problems in computer science and engineering, including optimization problems, graph algorithms, and dynamic control problems.

- Dynamic programming works by building up solutions to the subproblems in a bottom-up manner, starting with the smallest subproblems and working up to the original problem. It stores the solutions to the subproblems in a table or array, so that they can be reused when needed, and uses these solutions to construct the solution to the original problem.

- Dynamic programming is an efficient approach to solving problems, as it avoids the need to recalculate the solutions to subproblems that have already been solved. However, it can be somewhat difficult to implement and requires a good understanding of the problem and the underlying recursive structure of the solution.

Technique :

There are two different ways to store the values so that the values of a sub-problem can be reused:

1. **Memoization:** Memoization is a technique used in dynamic programming to improve the time complexity of recursive algorithms by storing the solutions to subproblems in a cache or table, so that they can be reused later instead of being recomputed. This technique is also known as **"top-down"** dynamic programming because it starts with the original problem and recursively breaks it down into smaller subproblems, but unlike tabulation, it does not start solving subproblems from the smallest to the largest.

   Memoization can be implemented by using a simple data structure such as an array or hash table to store the solutions to subproblems, which are indexed by the inputs to the subproblem. When a subproblem is encountered, the algorithm first checks if its solution is already in the cache before computing it. If the solution is in the cache, the algorithm uses the cached value and skips the computation.

   Memoization is an optimization technique that is useful when the same subproblems are encountered multiple times during the execution of the algorithm.

2. **Tabulation:** Tabulation, also known as **"bottom-up"** dynamic programming, is a method for solving a problem by breaking it down into smaller subproblems and storing the solution to each subproblem in a table, so that each subproblem is only solved once. The method starts by solving the smallest subproblems first, and then using these solutions to solve larger subproblems, until the solution to the original problem is obtained. Tabulation is used to improve the time complexity of recursive algorithms by avoiding the recomputation of previously solved subproblems.

So, When do we use DP ??? How to know which problem can be solved using DP ? – Anyone ?? Anyone !!

Thank you – Buddha ...

We have to check two Key characteristics : (Two key characteristics of DP)

- **Overlapping Subproblems:** A problem exhibits overlapping subproblems when it involves solving the same subproblems multiple times. DP utilizes this property by storing the solutions to these subproblems in a cache or table, so they can be reused later instead of being recomputed.
- **Optimal Substructure:** A problem exhibits optimal substructure when an optimal solution to the problem can be constructed from optimal solutions to its subproblems.
  In other word we can say that - if a problem can be solved by using the solutions of its subproblems then the problem has Optimal substructure property.

Let's Understand DP with two classical Problem-

A. Longest Common Sequence (LCS)
B. Fibonacci Sequence

Longest Common Sequence (LCS):

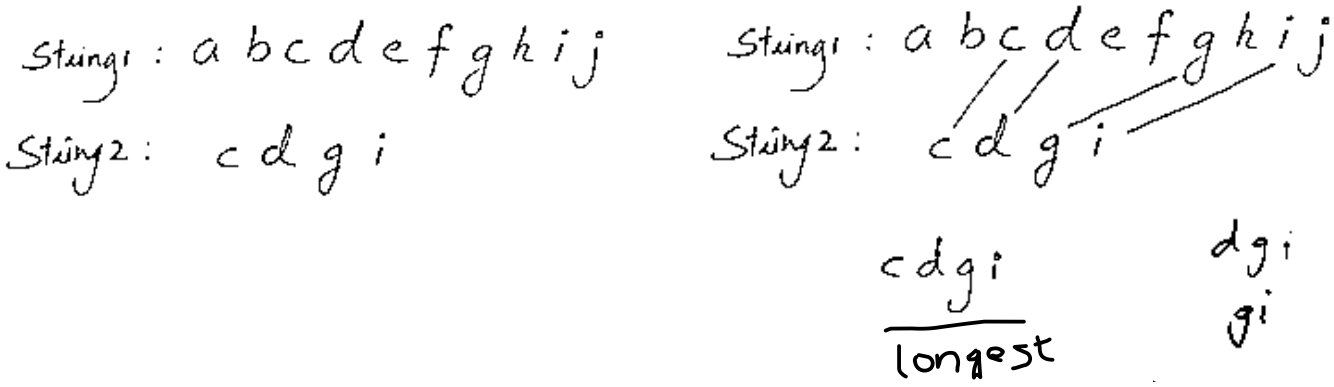
The Longest Common Subsequence (LCS) problem is the problem of finding the longest subsequence that is present in given two sequences in the same order. LCS is not necessarily contiguous in a given sequence, in contrast Longest Common Substring is.

For example, given the sequences "ABCD" and "ACDF", a longest common subsequence would be "ACD". Another example: "ABCD" and "EACB" would have "AC" as longest common subsequence.

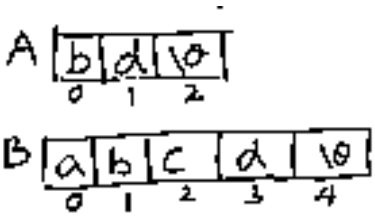
LCS can be solved using various approaches, for example using Dynamic Programming, by creating a 2D array and filling it up in a bottom-up manner. The time complexity of this approach is O(m*n) where m and n are the lengths of the given sequences.

LCS has many use cases, for example in bioinformatics, it is used to identify the similarities between DNA or protein sequences, and in text processing, it can be used for spell-checking and plagiarism detection.

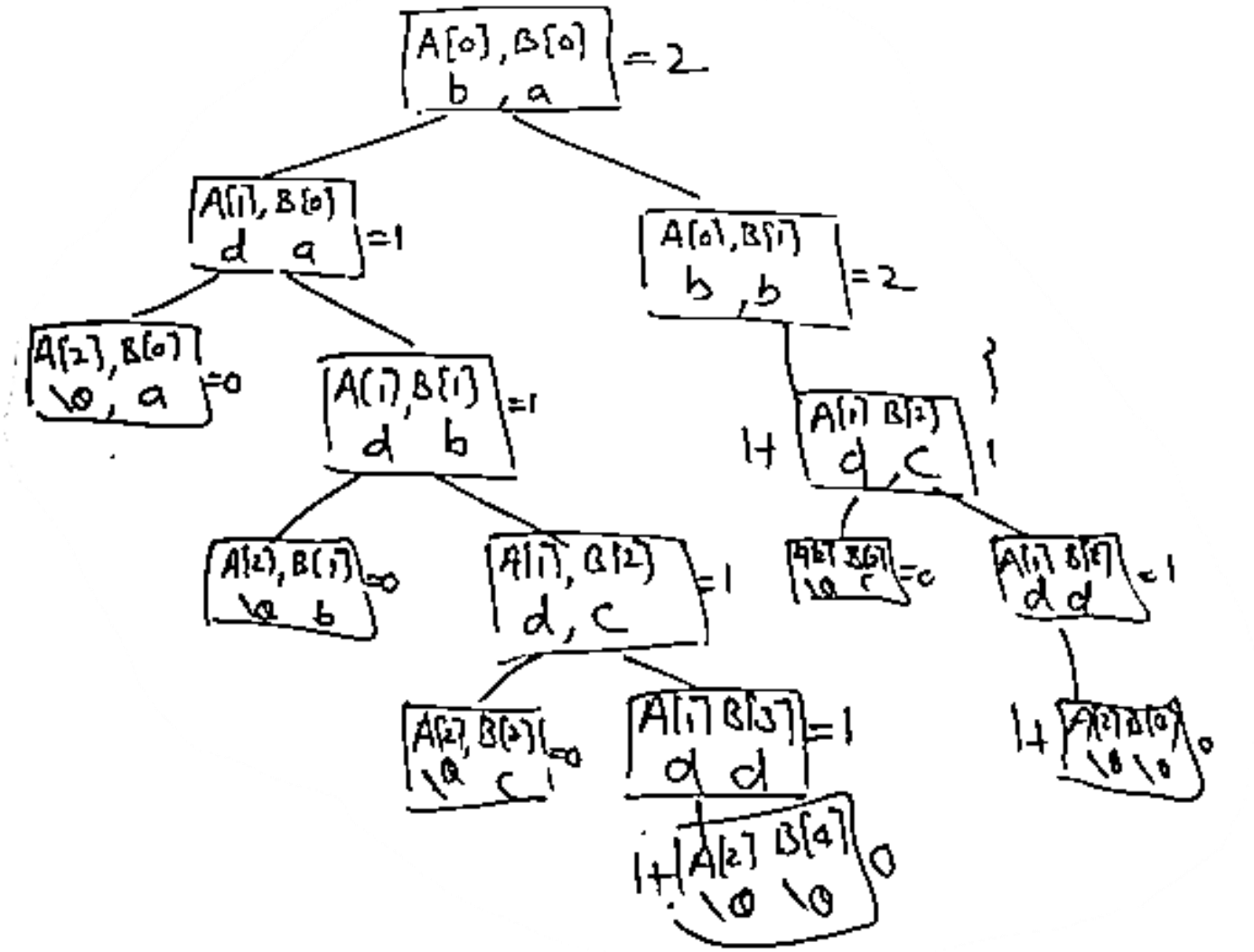
Board work :



Recursive Algorithm:




```
int LCS(i, j)
{
  if (A[i]=='\0' || B[j]=='\0')
      return 0;
  else if (A[i] == B[j])
    return 1 + LCS(i+1, j+1);
  else
    return max(LCS(i+1, j), LCS(i, j+1));
}
```



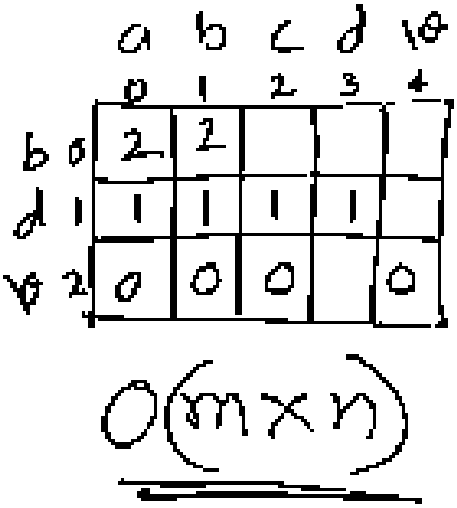
Complexity : $O(2^n)$ (Explained Later)

Let`s Solve it DP (Memoization):

1)Create a table where rows are representing the First string and the columns are representing the second String (could be vise-versa).

2) Store the answers each of recursive calls into the table

3) If we encounter a recursive call whose value is already Stored then no need of calling that recursive call.



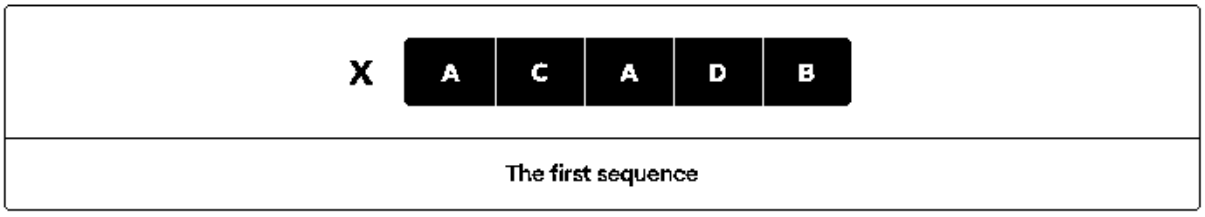
As Memoization reduced the number of calls the time complexity will be reduced . Which is O(m*n) where m= size od String A and n= size of string B.

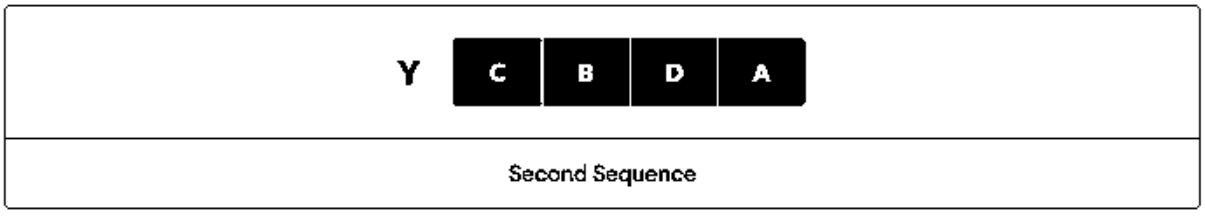
DP – Tabulation Method

The following steps are followed for finding the longest common subsequence.

1. Create a table of dimension `n+1*m+1` where n and m are the lengths of `X` and `Y` respectively. The first row and the first column are filled with zeros. Initialize a table
2. Fill each cell of the table using the following logic.
3. If the character corresponding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
4. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.
5. **Step 2** is repeated until the table is filled.
6. The value in the last row and the last column is the length of the longest common subsequence.
7. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to (↖) symbol form the longest common subsequence.

Let us take two sequences:



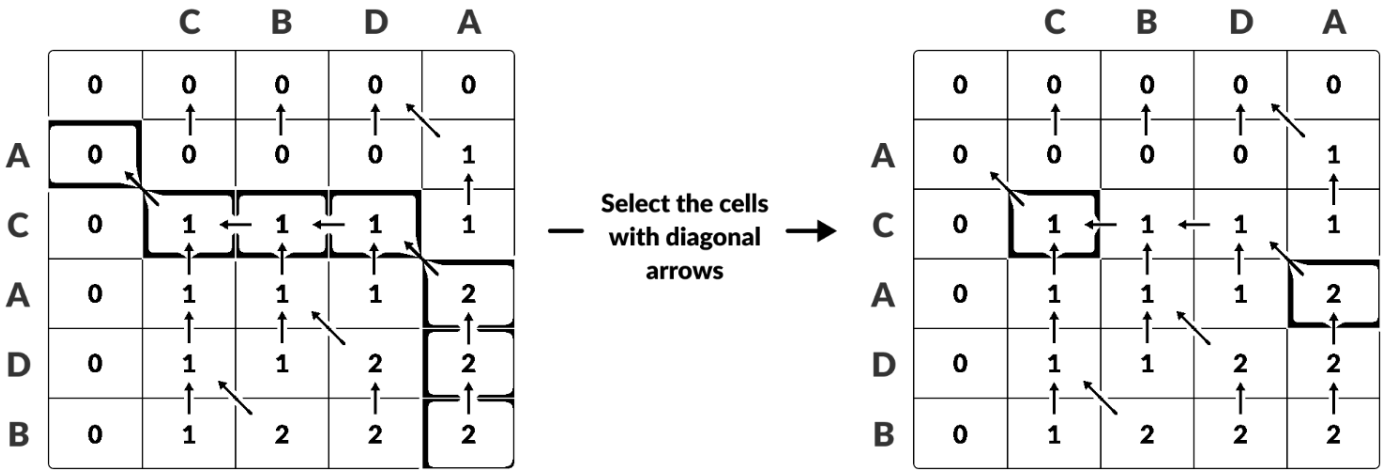


	C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

	C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0				
A	0				
D	0				
B	0				

	C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2
D	0	1	1	2	2
B	0	1	2	2	2

	C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2
D	0	1	1	2	2
B	0	1	2	2	2



Thus, the longest common subsequence is **CA**

Here , the time complexity is also O(m*n)

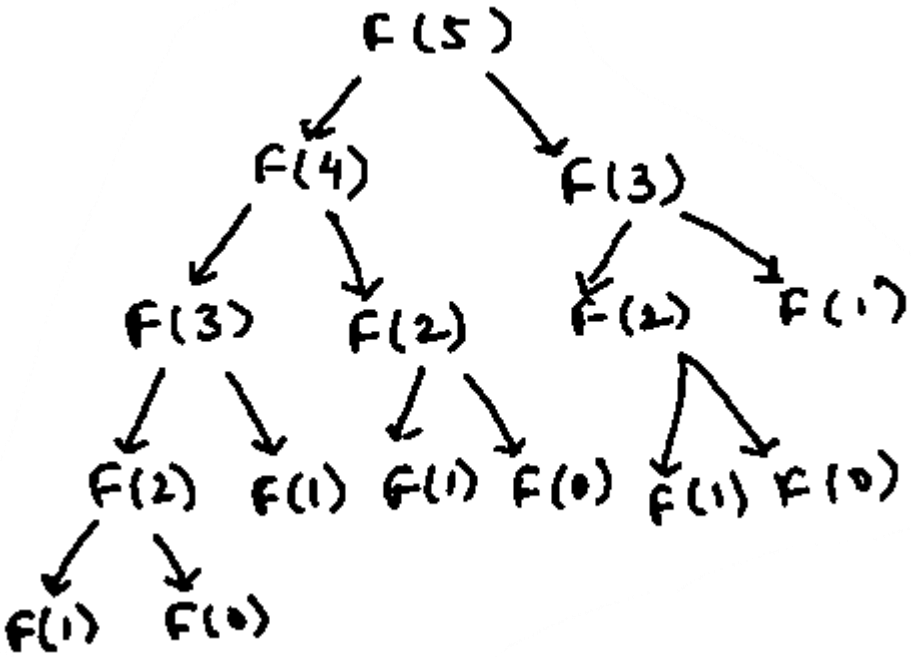
Fibonacci Series :

- The Fibonacci series in as infinite sequence of integers 0 ,1,1,2,3,5,8,……
- The first two integers 0 and 1 are initial values .
- The next subsequent numbers are obtained by adding previous two numbers
- In general, nth Fibonacci no a[n]= a[n-1]+a[n-2]

Recursive Algo.

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

Recursion Tree:



Time Complexity : $O(2^n)$

T(n) = T(n-1)+T(n-2) + C

For T(0) =T(1)= 1

Let, T(n-1) ≈ T(n-2)

T(n) = 2T(n-2)+C

=2 {2T(n-2-2)+C}+C

=2 {2T(n-4)+C }+C

=4T(n-4)+3C // k=2

=4{2T(n-4-2)+C}+3C

$=4\{2T(n-6)+C\}+3C$

$=8T(n-6)+7C$ //k=3

$=16T(n-8)+15C$ //k=4

In General ,

$T(n) = 2^kT(n-2k)+(2^k-1)C$

Now, if $T(n-2k)= T(0)$; $n-2k=0 \Rightarrow k=n/2$

$T(n) = 2^{n/2} T(0) + (2^{n/2} – 1)C$

$= (1+C)\ 2^{n/2} -C$

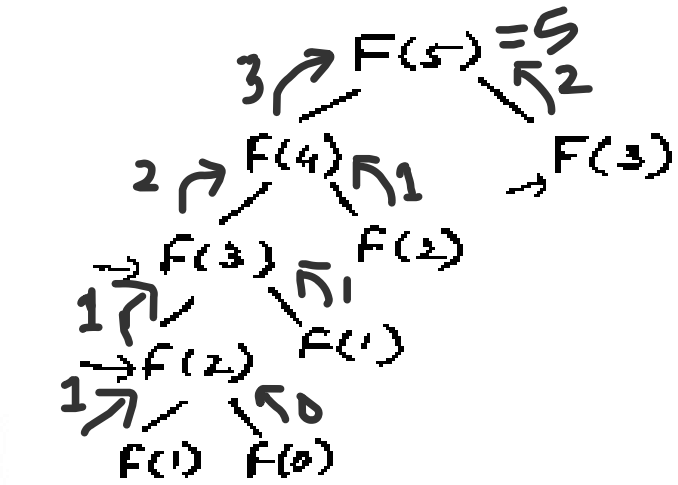
$T(n) \propto 2^{n/2}$ (lower Bound)

If we take $T(n-2) \approx T(n-1)$

We will get $T(n) \propto 2^n$ (upper Bound)

<u>Solve it using DP (Memoization):</u>

```
Fib(n)
{
  if n<=1
    return n
  if Fn is in memory
    return Fn
  else
    Fn ← Fib(n-1) + Fib(n-2)
    Save Fn in memory
    return Fn
}
```



Here , F(0), F(1), F(2), F(3), F(4), F(5) calls are called only once, Hence for F(5) we need 5+1 calls , In general for F(n) we will need n+1 calls . Thus the time complexity is O(n).

Solve it using DP (Tabulation):

```
int f(int n)
{
  if(n<=1)
     return n;
  f(0)=0;
  f(1)=1;
  for(int i=2;i<=n;i++)
  {
     f(i)=f(i-2)+f(i-1);
  }
   return f(n);
}
```

f(5)

	0	1	2	3	4	5
f	0	1	1	2	3	5

f(2) = f(0)+f(1) = 1
f(3) = f(1)+f(2) = 2
f(4) = f(2)+f(3) = 3
f(5) = f(3)+f(4) = 5

Unlike Memoization, here the lowest values generate the top values , Hence it's a bottom up approach .

Time complexity : O(n)..

The Time and Space Complexity of LCS (using recursion)

There are two choices for each character in a string: it can either be included in the subsequence or not. This means there are 2^m possible subsequences of X and 2^n possible subsequences of Y. So in this recursive approach, we are comparing each subsequence of X with each each subsequence of Y. The overall **time complexity** = O(2^m * 2^n)= O(2(m +n)), which is inefficient for large values of m and n.

The space complexity of this solution depends on the size of call stack, which is equal to the height of recursion tree. In this case, input parameters (m and n) are at most decreasing by 1 on each recursive call, so the height of recursion tree is equal to max(m, n). So space complexity = O(max(m, n)).

The Time and Space Complexity of LCS (using memoization/Top-Down)

In the worst case, we will be solving each subproblem only once and there are (m + l)*(n + 1) different sub-problems. So **time complexity** = O(m*n). We are using a 2D table of size (m + l)*(n + 1) to store the solutions of the subproblems. So **space complexity** = O(m*n).

The Time and Space Complexity of LCS (using Tebulation/Bottom-Up)

Time complexity = Time complexity of initializing the table + Time complexity of filling the table into a bottom-up manner = O(m + n) + O(mn) = O(mn). **Space complexity** = O(mn) for storing the table size (m+1)*(n+1)

,