Dynamic Programming

Dynamic programming is a computer programming technique where an algorithmic problem is **first broken down into sub-problems, the results are saved**, and then the sub-problems are optimized to find the overall solution — which usually has to do with finding the maximum and minimum range of the algorithmic query.
Richard Bellman was the one who came up with the idea for dynamic programming in the 1950s. It is a method of mathematical optimization as well as a methodology for computer programming.
Dynamic programming has two concepts:

1. Overlapping subproblems
2. Optimal substructure

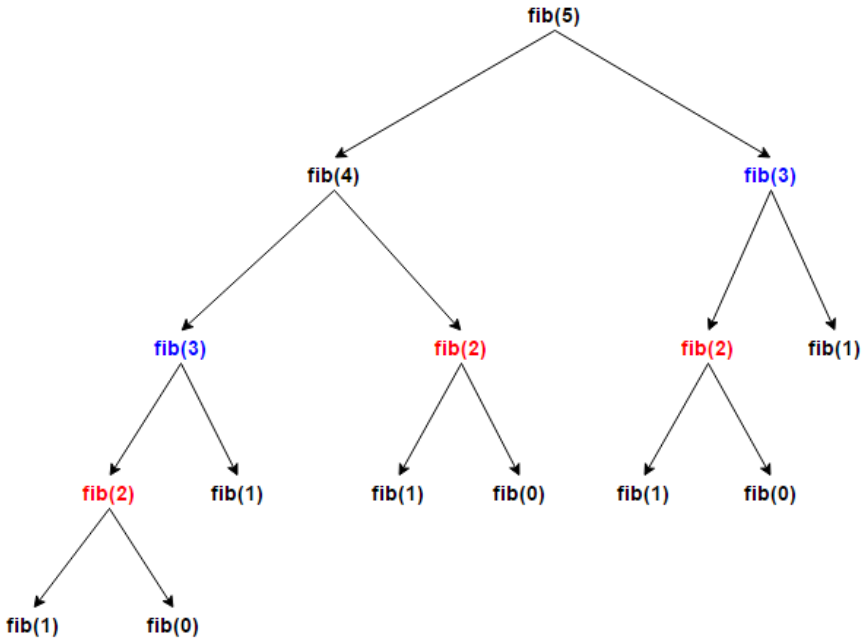
Overlapping Subproblems

A classic example of understanding the overlapping subproblem concept is a program to print the **Fibonacci series**.

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...]

The logic of the Fibonacci series is given by "**fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)**". And executing this function seamlessly can be done using a recursive solution, with a base case of **fibonacci(0)=0** and **fibonacci(1)=1**.

```
int fibonacci(int n){
      if(n==0 || n==1)
         return n;
      return fibonacci(n-1)+fibonacci(n-2);
 }
```



T(n) = T(n-1) + T(n-2) + O(1)
= 2^n

In the above recursion tree, we can see that **fib(2)** and **fib(3)** are repeated more than once. We have observed duplication in a recursion tree of height 5, now imagine we have a recursive call of a huge number, and subsequently, there will be a recursion tree with a big height. And there will be many such duplicate computations, which are known as **overlapping subproblems**.

We have two methods to deal with this (i)tabulation, (ii) memoization

Memoization

Solving the fibonacci problem using the memoization method can be done as shown below

```
int fibonacci(int n){
        if(memo[n]!=-1)
                return memo[n];
        if(n==0 || n==1){
                memo[n]=n;
                return n;
        }
        memo[n] = fibonacci(n-1)+fibonacci(n-2);
        return memo[n];
}
```

In the above code, we are creating a maintaining a lookup table(memo[]) and storing the values of computed results. Since we have memorized all the computed values, we can use them in the future if required, hence avoiding duplicate computations and overlapping subproblems.

All the logic is similar to the recursive solution, but the only difference we made is we are noting them in the memo array before we return the value to the main method. The only constraint to this algorithm is we need an extra space of size O(n), but we are optimizing the previous recursive solution.

Tabulation

This method is a little different from the above solution. Memoization follows the same recursive solution. But in tabulation, we follow an **iterative solution**. It is also called the **bottom-up approach**. Let us look at the implementation of the bottom-up approach.

```
 int fibonacci(int n){
       int table[]=new int[n+1];
       for(int i=0;i<=n;i++){
          if(i==0 || i==1)
             table[i]=i;
          else{
             table[i]=table[i-1]+table[i-2];
          }
       }
       return table[n];
   }
```

As we know that fibonacci(n)=fibonacci(n-1)+fibonacci(n-2), In memoization we started with fibonacci(4) function call, and then we realized that we haven't computed its value, so we've computed its values and then stored it.
We also have a way to compute fibonacci from 0 to nth element iteratively and then return the nth fibonacci element. This is what we've implemented in the above code. This code also has the same space requirement O(n).

Optimal Substructure

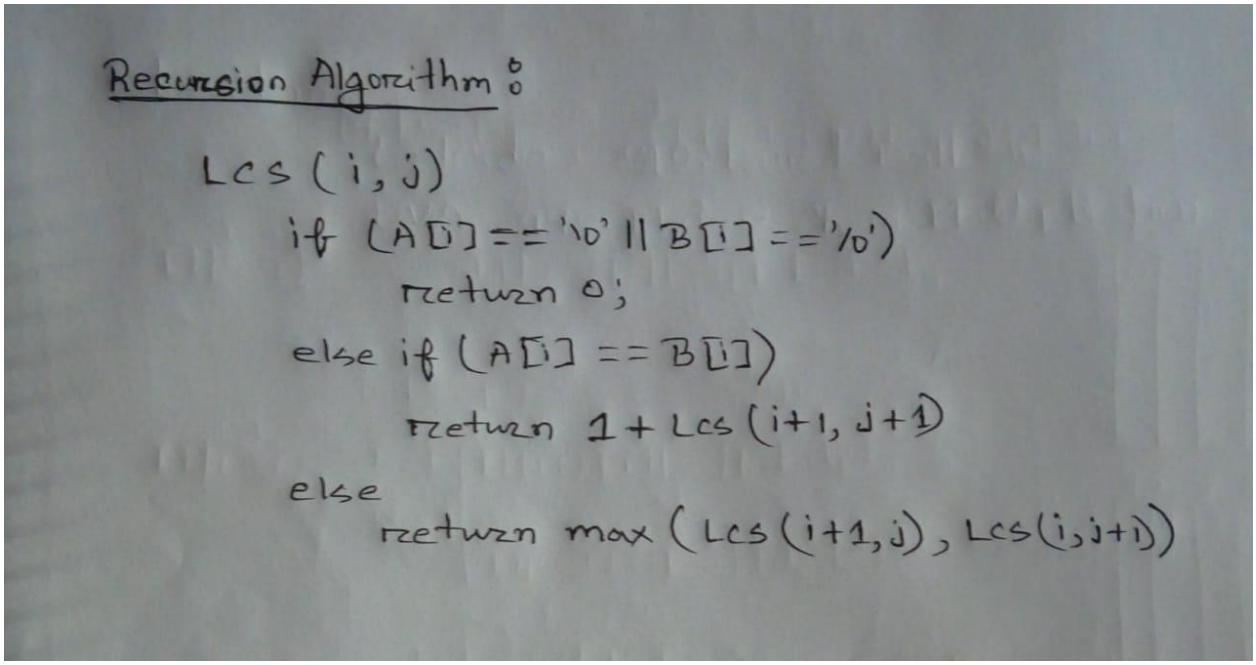
In this concept, we can obtain an optimal solution to a problem only if we have optimally solved its corresponding subproblems.

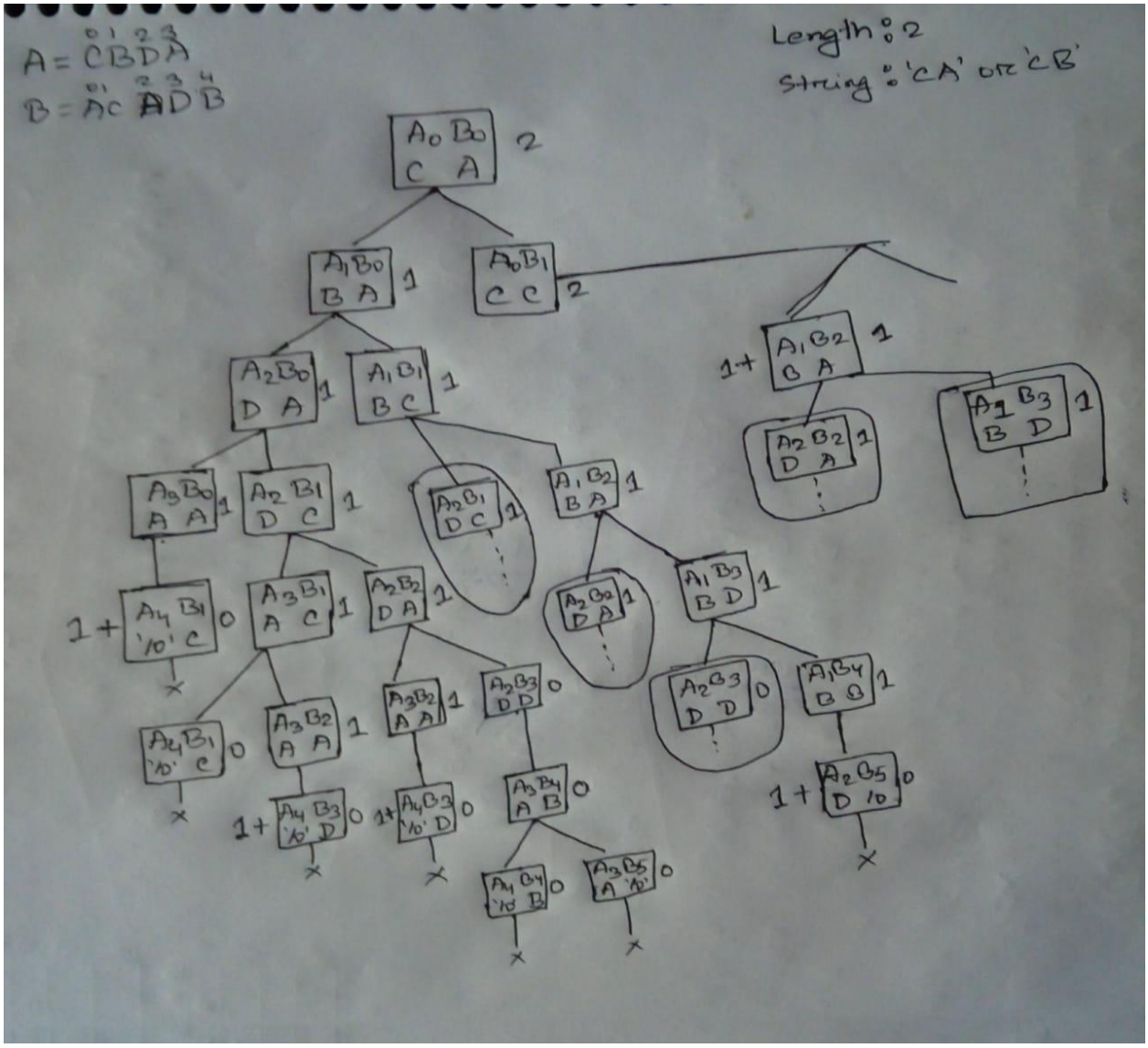
And a classic example of this concept is considering a traversal between nodes in a graph.

And a classic example for this property is the **longest common subsequence** in both strings. A subsequence is different from a substring. In a subsequence, the characters need not be consequent in the original string.

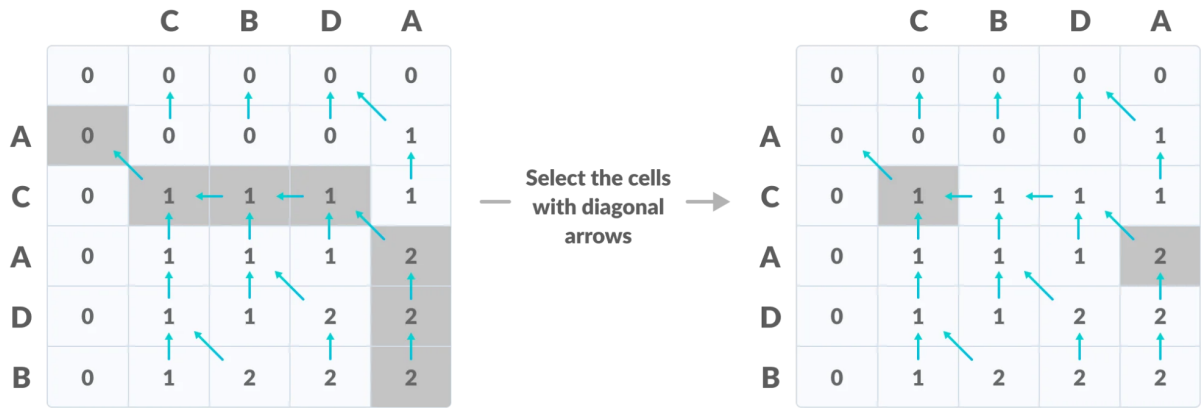
S1 = "CBDA" S2 = "ACADB"
Output = 2

Recursion Solution:




A = CBDA
B = AC ADB
Length: 2
String: 'CA' or 'CB'

Dynamic Problem Solution:




```
int lcs(String s1, String s2, int m, int n ){
   int dp[][] = new int[m+1][n+1];
   for(int i=0; i<=m; i++){
     for(int j=0; j<=n; j++){
       if(i == 0 || j == 0)
           dp[i][j] = 0;
       else if(s1.charAt(i-1) == s2.charAt(j-1))
           dp[i][j] = dp[i-1][j-1]+1;
       else
           dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
     }
   }
  return dp[m][n];
  }
```

Practice problem: https://leetcode.com/problems/longest-common-subsequence/
https://leetcode.com/problems/longest-palindromic-substring/