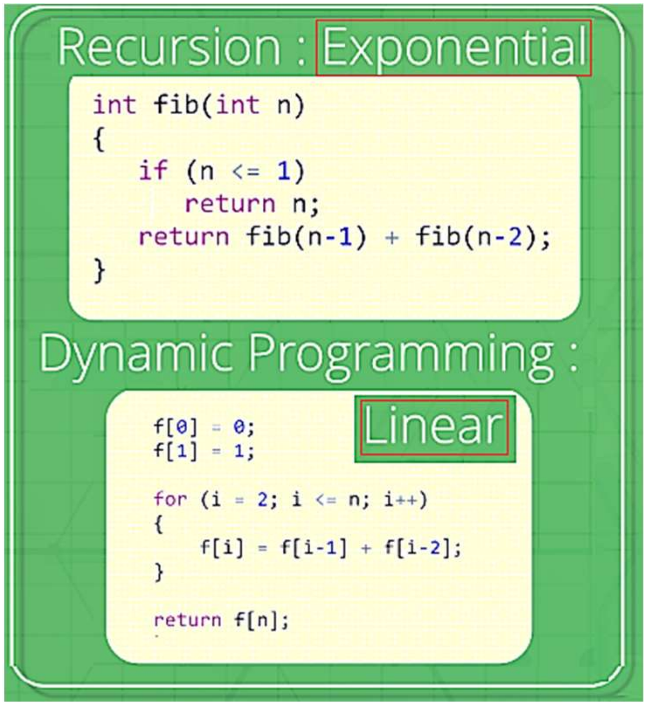
Lecture 17

Dynamic Programming: Comparison of DP, Divide & Conquer, and Elements of Dynamic Programming

Real Life Example of DP

- Fibonacci Numbers
  - Recursive solution of Fibonacci numbers have exponential time complexity, after optimizing it by storing solutions of the subproblems, time complexity reduces to linear.
  - The Fibonacci numbers are the numbers in the following integer sequence.
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..
  - $F_n=F_{n-1} + F_{n-2}$ with seed values $F_0=0$ and $F_1=1$.
  - Examples
    - Input n=2, output=1
    - Input n=9, Output=34



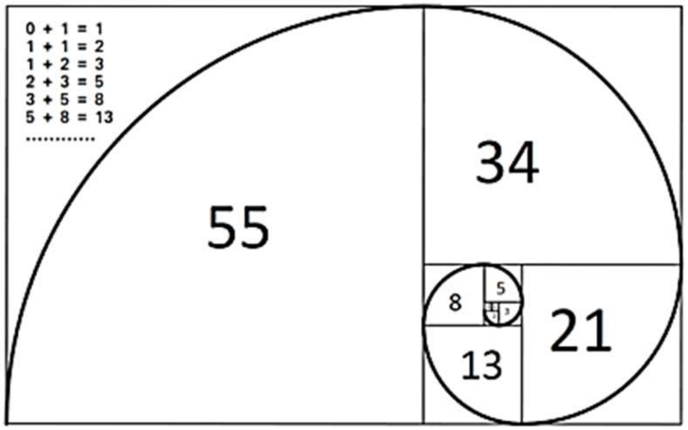
Recursion : Exponential

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

Dynamic Programming :

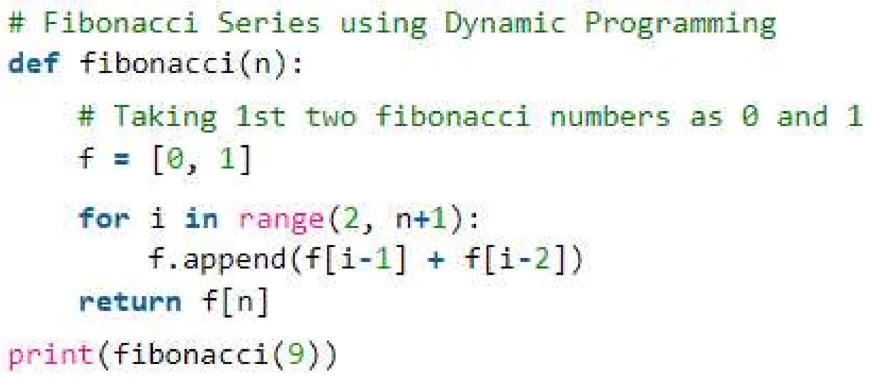
Linear

```
f[0] = 0;
f[1] = 1;

for (i = 2; i <= n; i++)
{
    f[i] = f[i-1] + f[i-2];
}

return f[n];
```



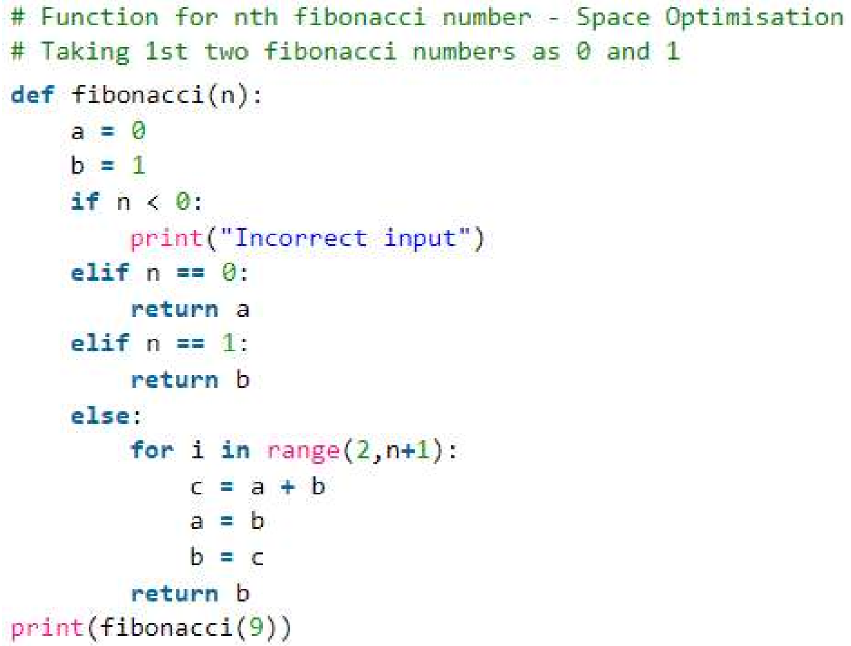
Python Code: Dynamic Programming

- **Time complexity**: O(n) for given n
- **Auxiliary space**: O(n)



```python
# Fibonacci Series using Dynamic Programming
def fibonacci(n):

    # Taking 1st two fibonacci numbers as 0 and 1
    f = [0, 1]

    for i in range(2, n+1):
        f.append(f[i-1] + f[i-2])
    return f[n]

print(fibonacci(9))
```

- **Time Complexity:** O(n)
  Extra Space: O(1)



```python
# Function for nth fibonacci number - Space Optimisation
# Taking 1st two fibonacci numbers as 0 and 1
def fibonacci(n):
    a = 0
    b = 1
    if n < 0:
        print("Incorrect input")
    elif n == 0:
        return a
    elif n == 1:
        return b
    else:
        for i in range(2,n+1):
            c = a + b
            a = b
            b = c
        return b
print(fibonacci(9))
```

https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/

Dynamic Programming

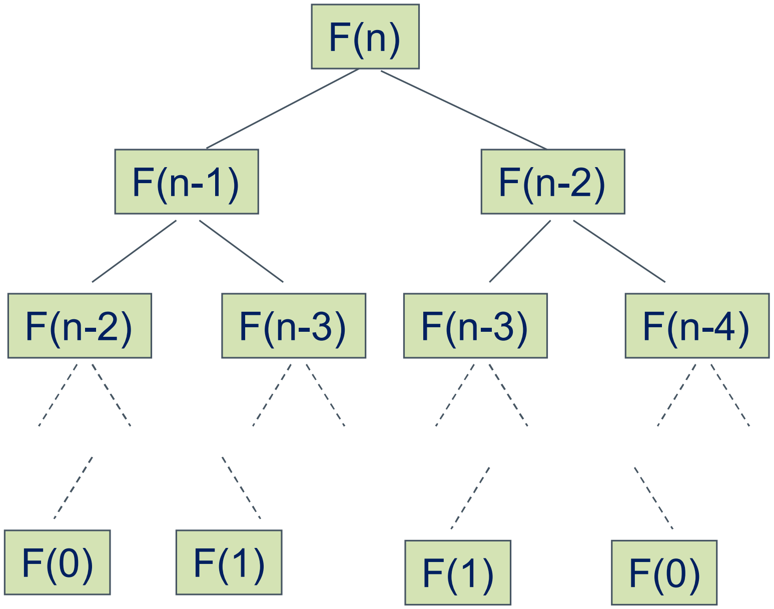
- Breaking down an optimization problem into simpler sub-problems, and storing the solution to each sub-problem so that each sub-problem is only solved once
- Dynamic Programming is an algorithmic paradigm for solving a given complex problem by breaking it down into subproblems and memorizing the outcomes of those subproblems to prevent repeating computations.
- Dynamic programming is applicable in
  - graph theory;
  - game theory;
  - AI and machine learning;
  - Economics and finance problems;
  - Bioinformatics;
  - Calculating the shortest path, which is used in GPS.
- Dynamic Programming can only be applied to the given problem if it follows the properties of dynamic programming.

Dynamic Programming

- What is the drawback of recursive algorithms like Fibonacci sequence?

```
Fib(n)
    if ( n <= 1)
        return n;
    return ( Fib(n-1) + Fib(n-2));
```

$O(2^n)$

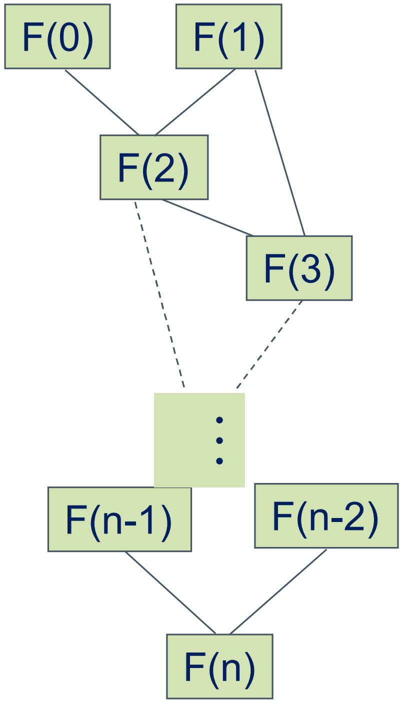


Dynamic Programming

- Solving Fibonacci sequence using dynamic programming.

```
Fib_DP(n)
    f[0] = 0;
    f[1] = 1;
    for i = 2 to n
        f[i] = f[i-1] + f[i-2];
    return f[n];
```

$O(n)$

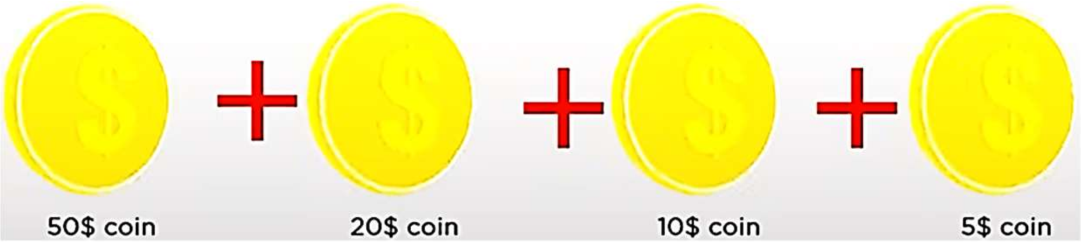


Properties of Dynamic Programming

- Optimal Structure
- Overlapping Subproblems
- Variant: Utilizing Memory

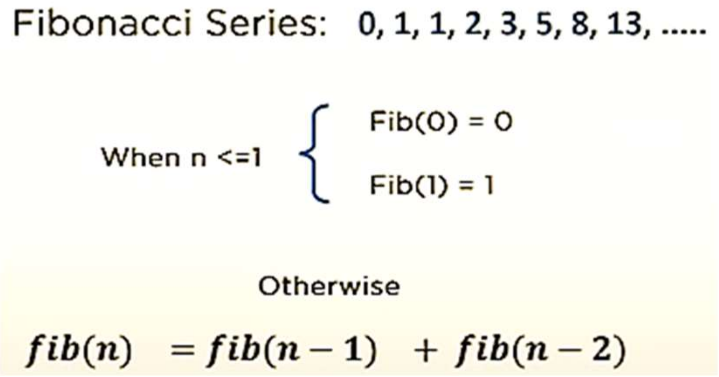
(1) Optimal substructure

- A problem is said to have optimal substructure if we can formulate a recurrence relation for it.
- For Example
  - Consider the coin change problem, in which you have to construct coin combinations with the least potential number of coins.
    - Divide the $85 into 50$ + 20$ + 10$ + 5$ the least possible coins to construct the solution.
    - This conversion of large problems into smaller subproblems by breaking it at each iteration is known as Optimal Substructure



(2) Overlapping subproblem

- A problem is said to have an overlapping subproblem if the subproblems reoccur when implementing a solution to the larger problems.
- Overlapping can be understood by developing recurrence relationships.
- For Example, Fibonacci Series.
  - *fib(n = fib(n-1) + fib(n-2)* Fibonacci series is the set of numbers which appear in nature all the time. Each number in this series is equal to the sum of the previous numbers before it.



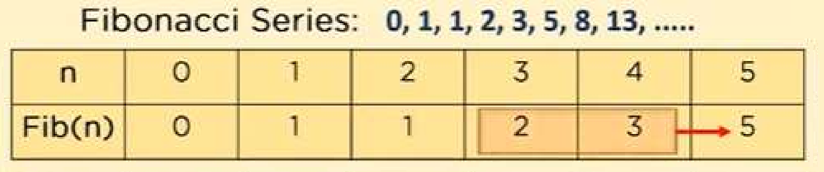
Fibonacci Series: **0, 1, 1, 2, 3, 5, 8, 13, ...**

$$\text{When } n <=1 \begin{cases} Fib(0) = 0 \\ Fib(1) = 1 \end{cases}$$

Otherwise

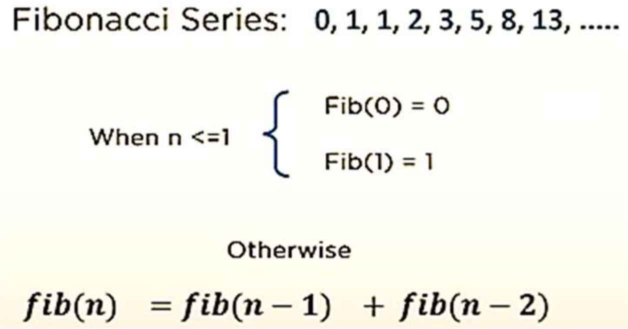
$$fib(n) = fib(n-1) + fib(n-2)$$

(2) Overlapping subproblem: Recursion



Fibonacci Series: **0, 1, 1, 2, 3, 5, 8, 13, ...**

n	0	1	2	3	4	5
Fib(n)	0	1	1	2	3	→ 5



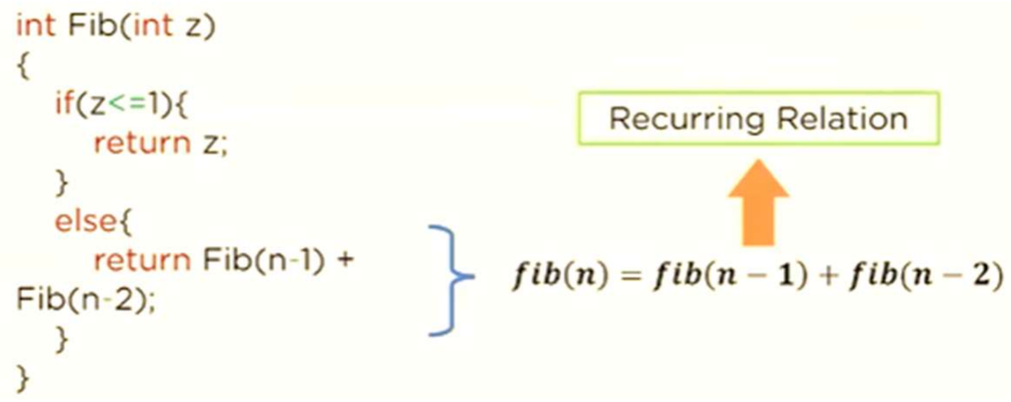
Fibonacci Series: **0, 1, 1, 2, 3, 5, 8, 13, ...**

When n <=1 $\begin{cases} \text{Fib(0) = 0} \\ \text{Fib(1) = 1} \end{cases}$

Otherwise

$$fib(n) \quad = fib(n-1) \quad + fib(n-2)$$

If we can establish a recurrence relation for a problem, we say it has an optimal substructure.



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

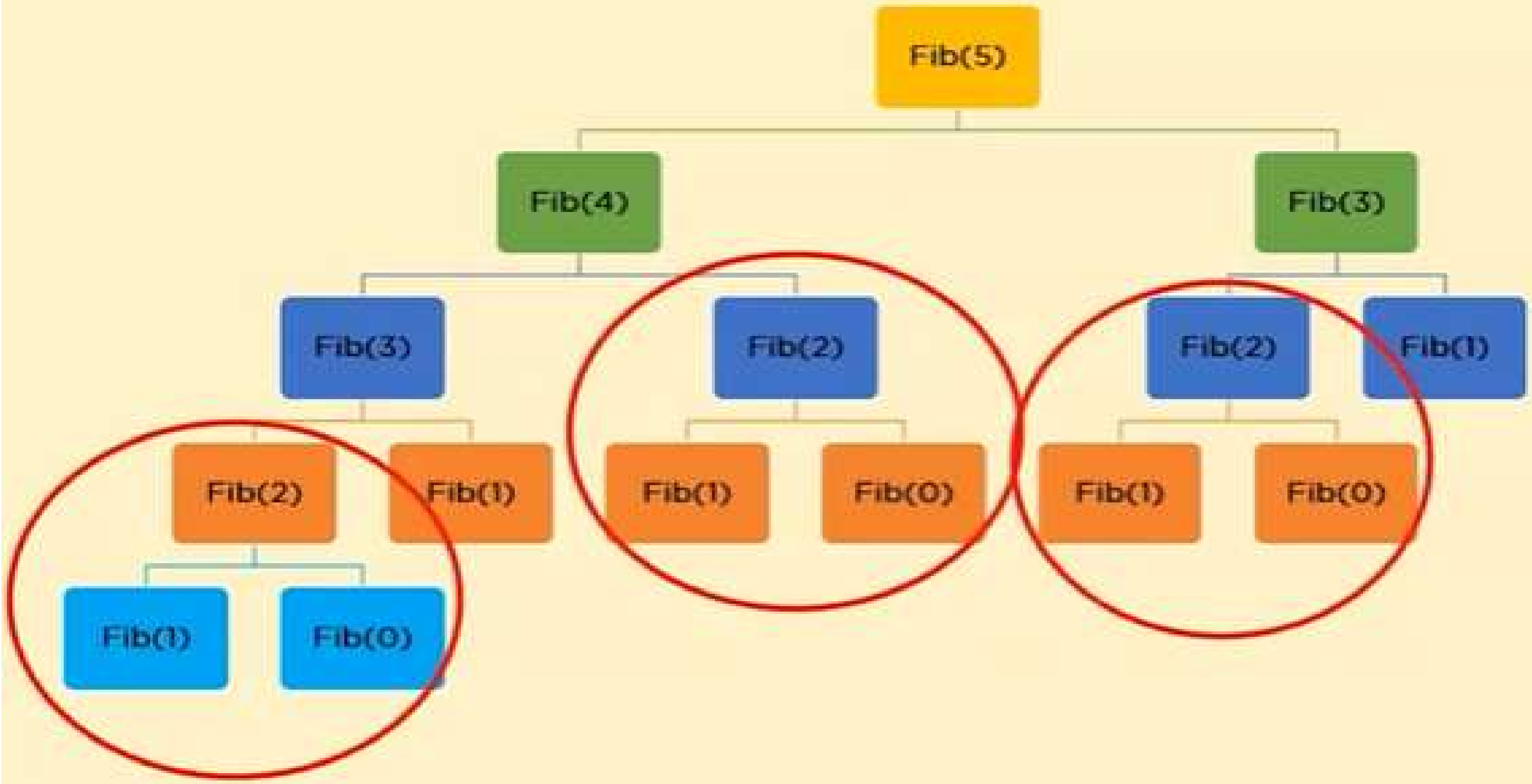
Recurring Relation

$$fib(n) = fib(n-1) + fib(n-2)$$

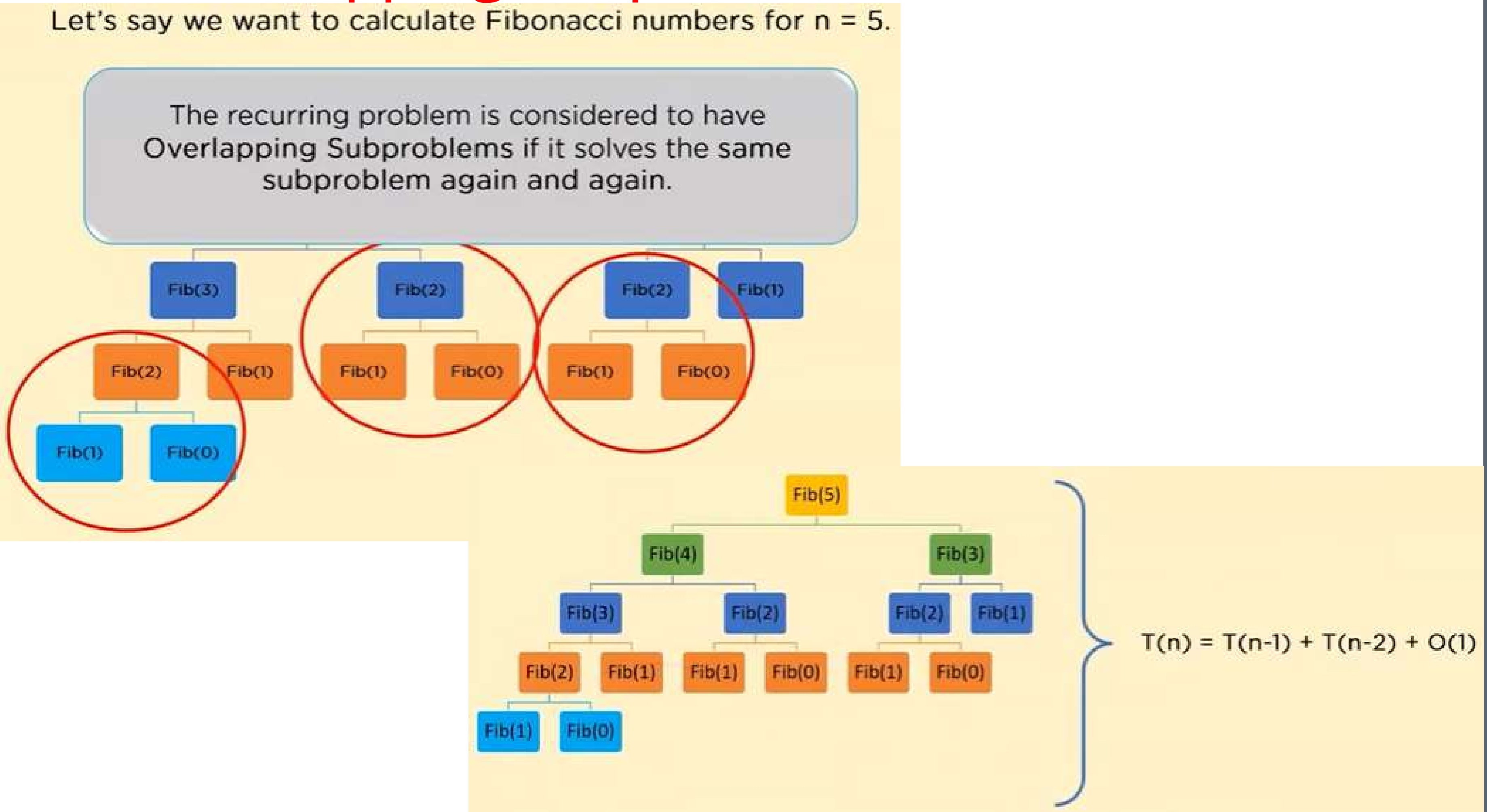
What about the
OVERLAPPING STRUCTURE?

(2) Overlapping subproblem: Recursion

Let's say we want to calculate Fibonacci numbers for n = 5.



(2) Overlapping subproblem: Recursion

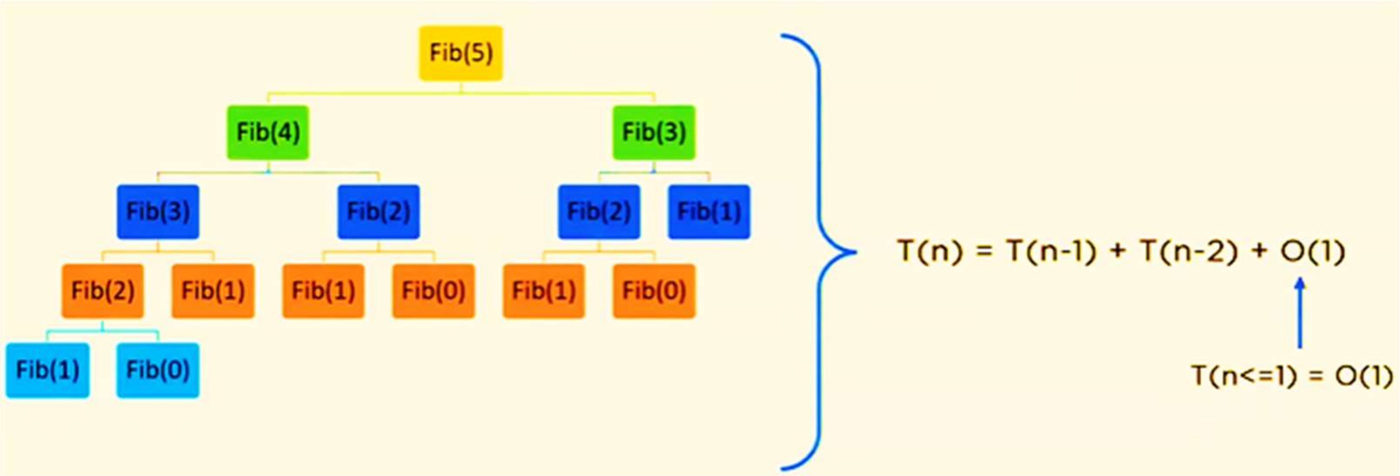


Let's say we want to calculate Fibonacci numbers for n = 5.

The recurring problem is considered to have Overlapping Subproblems if it solves the same subproblem again and again.

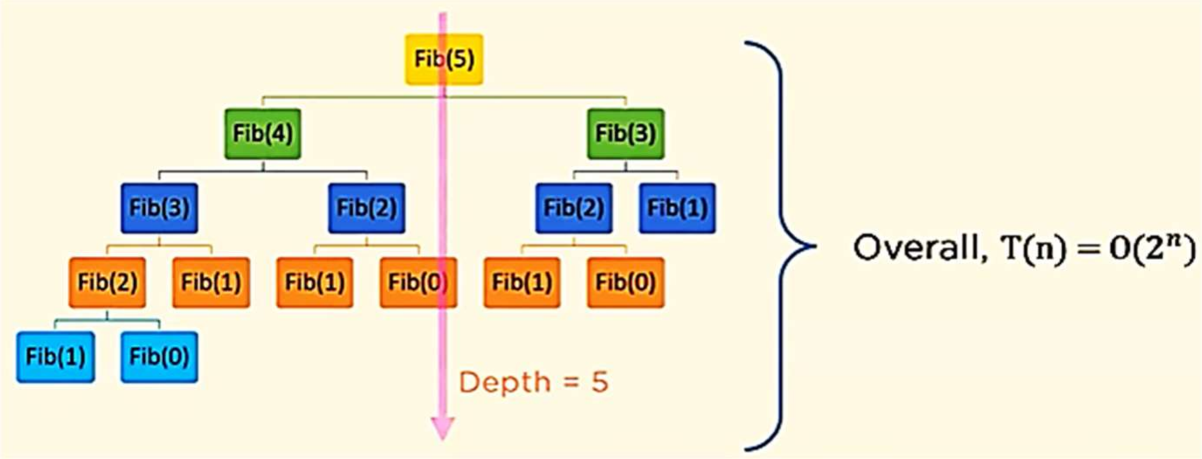
$T(n) = T(n-1) + T(n-2) + O(1)$

(2) Overlapping subproblem: Recursion



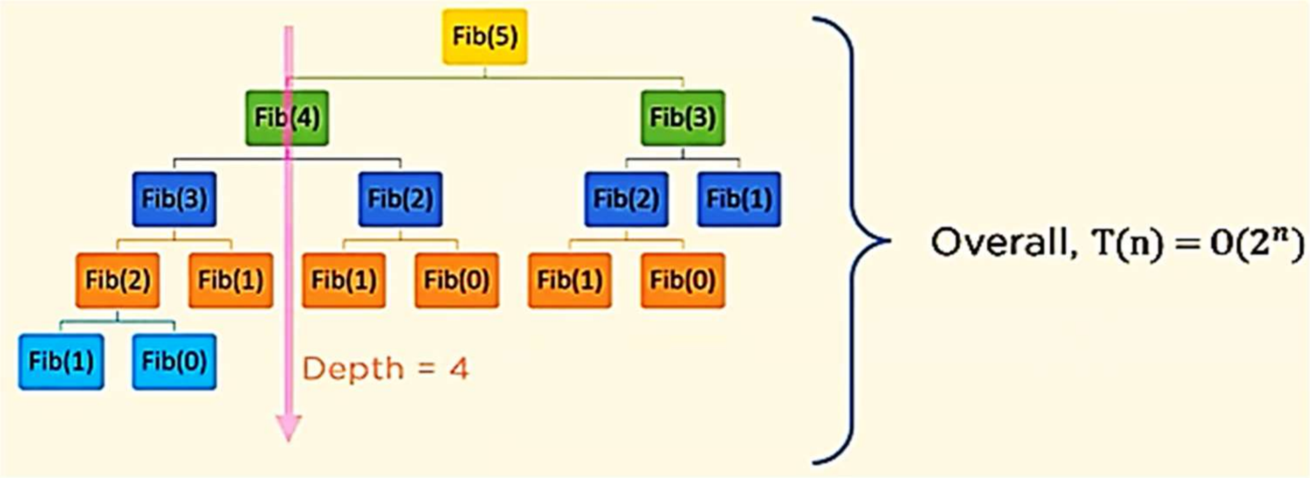
T(n) = T(n-1) + T(n-2) + O(1)

T(n<=1) = O(1)

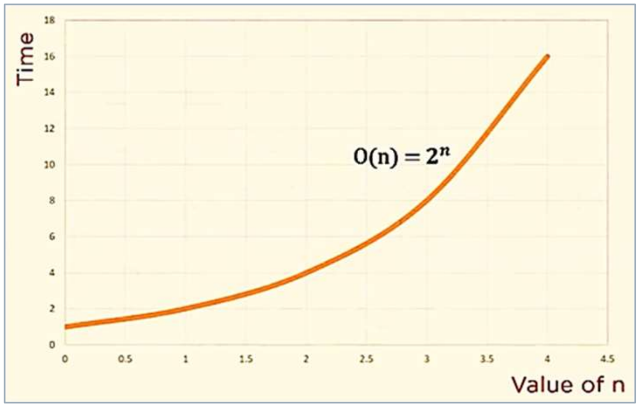


Overall, $T(n) = O(2^n)$

(2) Overlapping subproblem: Recursion

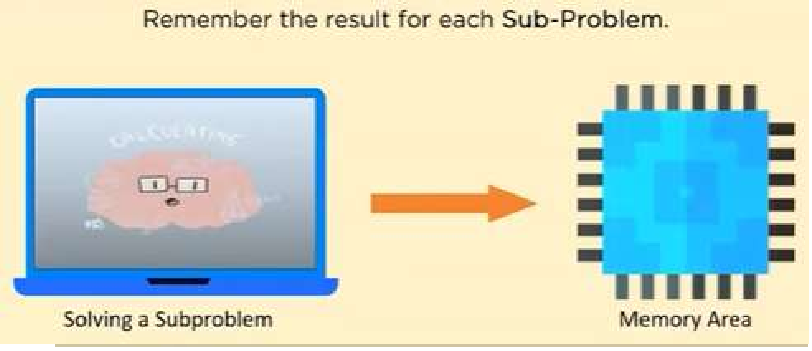


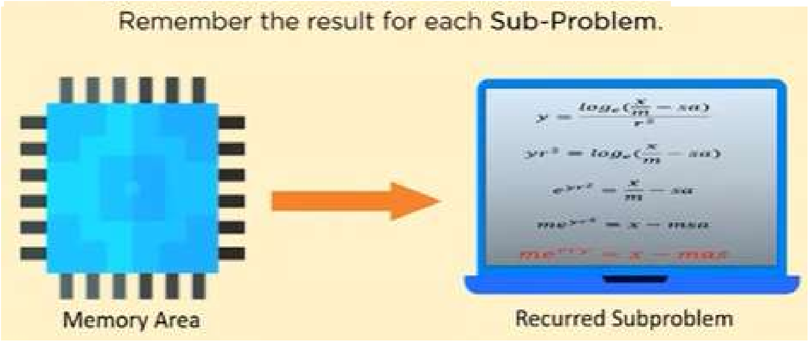
Overall, $T(n) = O(2^n)$



(3) Utilizing Memory

- Solution: Utilizing Memory
  - Dynamic programming stores the results of subproblems in memory and recalls whenever the recurrence of calculated subproblems occurs.
  - Two Methods of Storing results
    - Memorization: results stored in memory whenever we solve a particular subproblem for the first time.
    - Tabulation: precompute the solutions in a linear fashion and store it in a tabular format





Ways to handle Overlapping Subproblems

MEMORIZATION

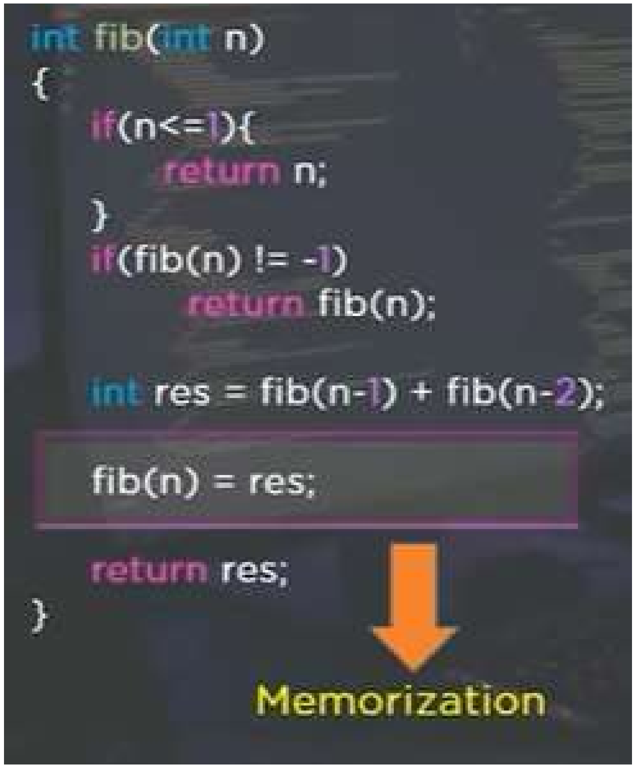
- Known as a Top-down approach
- A lookup table is maintained and checked before computation of any state.
- Recursion is involved

TABULATION

- Known as Bottom-Up Approach
- In this method, solution is built from the base or bottom-most state
- The process is Iterative

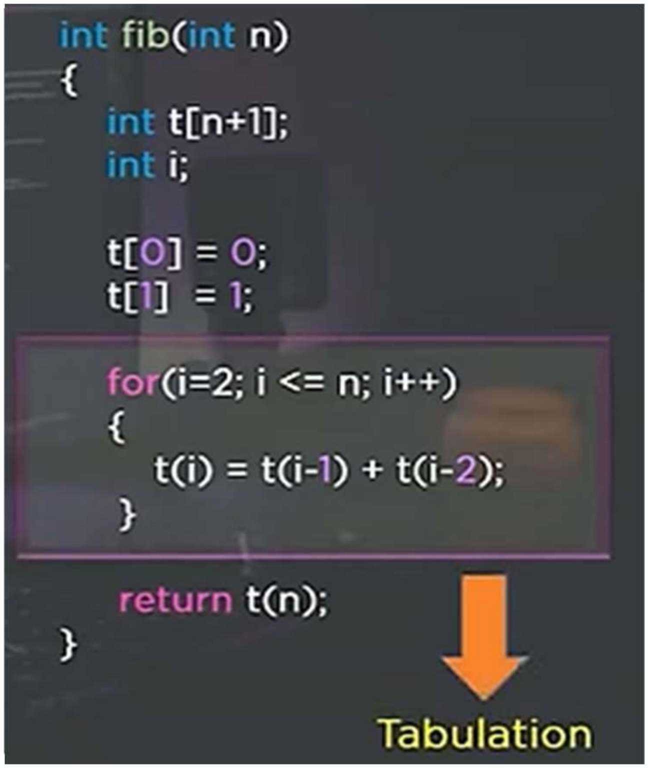
Ways to handle Overlapping Subproblems

MEMORIZATION



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

    int res = fib(n-1) + fib(n-2);

    fib(n) = res;

    return res;
}
```

TABULATION



```
int fib(int n)
{
    int t[n+1];
    int i;

    t[0] = 0;
    t[1]  = 1;

    for(i=2; i <= n; i++)
    {
       t(i) = t(i-1) + t(i-2);
     }

     return t(n);
}
```

When to use Dynamic Programming?

- When we need an exhaustive solution, we can use Dynamic Programming to address minimization and maximization problems.
- Permutation Problems: finding the number of the problems can be solved using DP.

Comparison of Dynamic Programming & Divide and Conquer

- Divide and Conquer
  - Store the problems by dividing problems into subproblems
  - Subproblems are not dependent on each other
    - Examples:
      - MergeSort,
      - QuickSort,
      - Binary Search
  - Does not store solutions of subproblems
  - Top-down algorithm

- Dynamic Programming
  - Store the problems by dividing problems into subproblems
  - Subproblems are dependent on each other
    - Example:
      - 0,1 Knapsack,
      - Matric Chain Multiplication
  - Store solution of subproblems
  - Bottom-up Algorithm

Examples of DP algorithms

- Computing a binomial coefficient
- Longest common subsequence
- Warshall's algorithm for transitive closure
- Floyd's algorithm for all-pairs shortest paths
- Constructing an optimal binary search tree
- Some instances of difficult discrete optimization problems:
  - - Traveling salesman
  - - Knapsack

Thank You!!!

Have a good day