

The slide features a decorative background. On the left, a blue wavy vertical band contains a repeating pattern of the 'Learn OA' logo and the tagline 'To The Next Level...'. In the top right corner, there are stylized pink icons of a paperclip, a ruler, the Greek letter alpha, the Greek letter gamma, a pencil, and a protractor. The main title is centered in a bold, pink, sans-serif font.

# OBJECT ORIENTED ANALYSIS & DESIGN DATA STRUCTURES & ALGORITHMS

## Dynamic Programming

# Overview

Recursion

Dynamic Programming Introduction

Characteristics of Dynamic Programming

- I. Overlapping Sub-Problems
- II. Optimal Substructure Property

Dynamic Programming Methods

- III. Top-down with Memoization
- IV. Bottom-up with Tabulation



# Recursion

- Recursion is the process in which a function calls itself directly or indirectly.
- This technique provides a way to break complicated problems down into simple problems which are easier to solve.
- The corresponding function is called as **recursive function**
- In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.
- Recursion is a prerequisite for **Dynamic Programming**

# A simple example of Recursion

Adding two numbers together is easy to do but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

```
public class Main {  
    public static void main(String[] args) {  
        int result = sum(10);  
        System.out.println(result);  
    }  
    public static int sum(int k) {  
        if (k > 0) {  
            return k + sum(k - 1);  
        } else {  
            return 0;  
        }  
    }  
}
```

// base case

## Another example of Recursion

In this example of calculating factorial for a number, the base case for  $n \leq 1$  is defined and larger value of number can be solved by converting to smaller one till base case is reached.

```
int fact(int n)
{
    if (n <= 1) // base case
        return 1;
    else
        return n*fact(n-1);
}
```



# Stack Overflow problem in Recursion

While solving problems through recursion, the choice of base case is very critical. A wrong base case, which is either not reached or not defined, may cause the Stack Overflow problem. See the following for example:

```
int fact(int n)
{
    // wrong base case (it may cause
    // stack overflow).
    if (n == 100)
        return 1;

    else
        return n*fact(n-1);
}
```

In this case, if fact(10) is called, it will call fact(9), fact(8), fact(7) and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

# Direct Recursion

If a function calls itself, it's known as direct recursion. This results in a one-step recursive call. The function calls itself inside its own function body.

Something like this:

```
void directRecursionFunction()
{
    // some code...

    directRecursionFunction();

    // some code...
}
```

# Example of Direct Recursion

Below is an example of a direct recursive function that computes the square of a number:

```
int square(int x)
{
    // base case
    if (x == 0)
    {
        return x;
    }

    // recursive case
    else
    {
        return square(x-1) + (2*x) - 1;
    }
}

int main() {
    // implementation of square function
    int input=30;
    cout << input<<"^2 = "<<square(input);
    return 0;
}
```



# Indirect Recursion

If the function f1 calls another function f2, and f2 calls f1 then it is indirect recursion (or mutual recursion).

This is a two-step recursive call: the original function calls another function, which in turn calls the original function.

**Note:** For indirect recursion, both the functions need to be declared **before** they are defined.

```
void indirectRecursionFunctionf1();  
void indirectRecursionFunctionf2();  
  
void indirectRecursionFunctionf1()  
{  
    // some code...  
  
    indirectRecursionFunctionf2();  
  
    // some code...  
}  
  
void indirectRecursionFunctionf2()  
{  
    // some code...  
  
    indirectRecursionFunctionf1();  
  
    // some code...  
}
```

# Example of Indirect Recursion

Here is an example of an indirect recursive function that prints the first 20 integers.

```
int n=0;  
// declaring functions  
void foo1(void);  
void foo2(void);
```

```
// defining recursive functions  
void foo1()  
{  
    if (n <= 20)  
    {  
        cout<<n<<" "; // prints n  
        n++;           // increments n by 1  
        foo2();         // calls foo2()  
    }  
    else  
        return;  
}  
  
void foo2()  
{  
    if (n <= 20)  
    {  
        cout<<n<<" "; // prints n  
        n++;           // increments n by 1  
        foo1();         // calls foo1()  
    }  
    else  
        return;  
}
```

# Dynamic Programming

Write down  $1+1+1+1+1+1+1+1 =$  on a sheet of paper

What's that equal to? Eight!

Write down another "1+" on the Left

What about that? Quickly "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 1s!

Dynamic Programming is just a fancy word for remembering stuff to save time later



# Dynamic Programming

Dynamic Programming is a method for solving a complex problem by breaking it down into a collection of simpler sub-problems, solving each of those sub-problems just once, and storing their solutions using a memory-based data structure (array, map, etc).

Each of the sub-problem solutions is indexed in some way, typically based on the values of its input parameters, to facilitate its lookup. So, the next time the same sub-problem occurs, instead of re-computing its solution, one simply looks up the previously computed solution, thereby saving computation time.

This technique of storing solutions to sub-problems instead of re-computing them is called **memoization**.





# Dynamic Programming – Characteristics

## Overlapping Sub-Problems

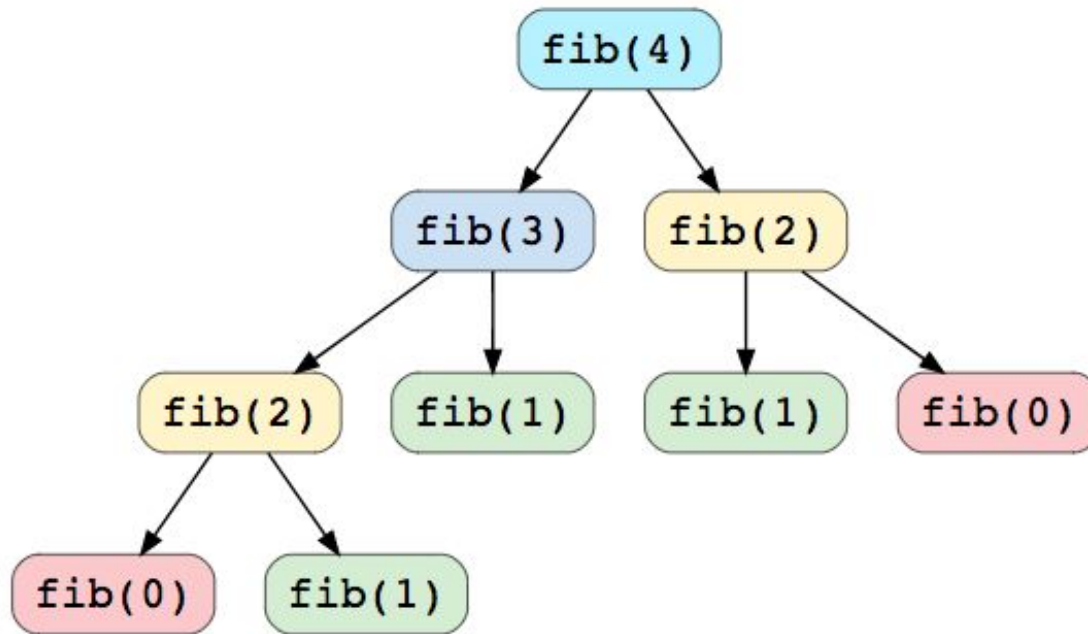
Sub-Problems are smaller versions of the original problem. Any problem has overlapping sub-problems if finding its solution involves solving the same sub-problem multiple times.

Take the example of the Fibonacci numbers; to find the  $\text{fib}(4)$ , we need to break it down into the following sub-problems:

We can clearly see the overlapping sub-problem pattern here, as  $\text{fib}(2)$  has been called twice and  $\text{fib}(1)$  has been called three times.



# Dynamic Programming – Characteristics



We can clearly see the overlapping sub-problem pattern here, as fib(2) has been called twice and fib(1) has been called three times.

# Dynamic Programming – Characteristics

## Optimal Substructure Property

Any problem has optimal substructure property if its overall optimal solution can be constructed from the optimal solutions of its sub-problems. For Fibonacci numbers, as we know,

$$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$$

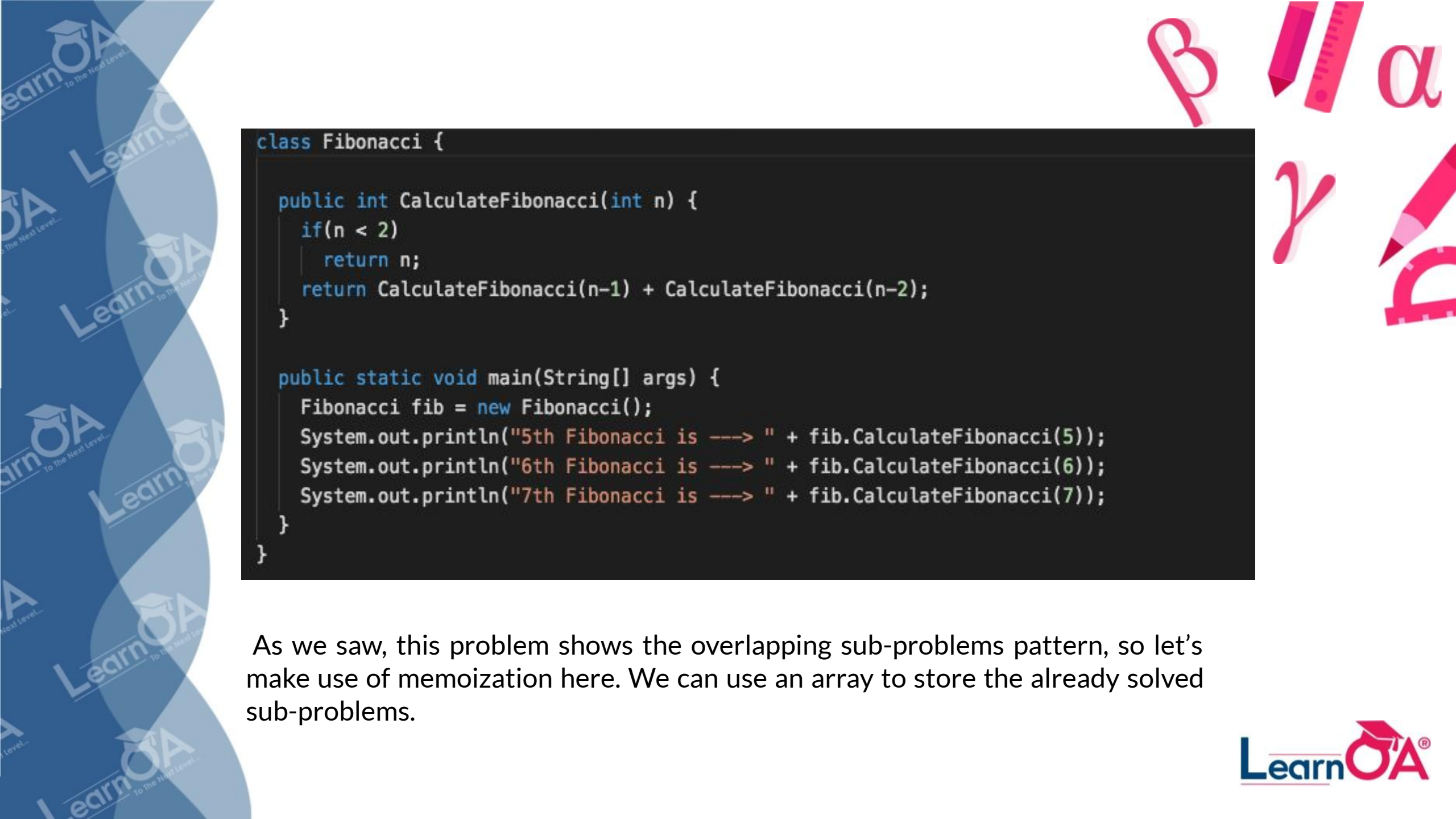
This clearly shows that a problem of size 'n' has been reduced to sub-problems of size 'n-1' and 'n-2'. Therefore, Fibonacci numbers have optimal substructure property.

# Dynamic Programming - Methods

## Top-down with Memoization

In this approach, we try to solve the bigger problem by recursively finding the solution to smaller sub-problems. Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. Instead, we can just return the saved result. This technique of storing the results of already solved sub-problems is called Memoization.

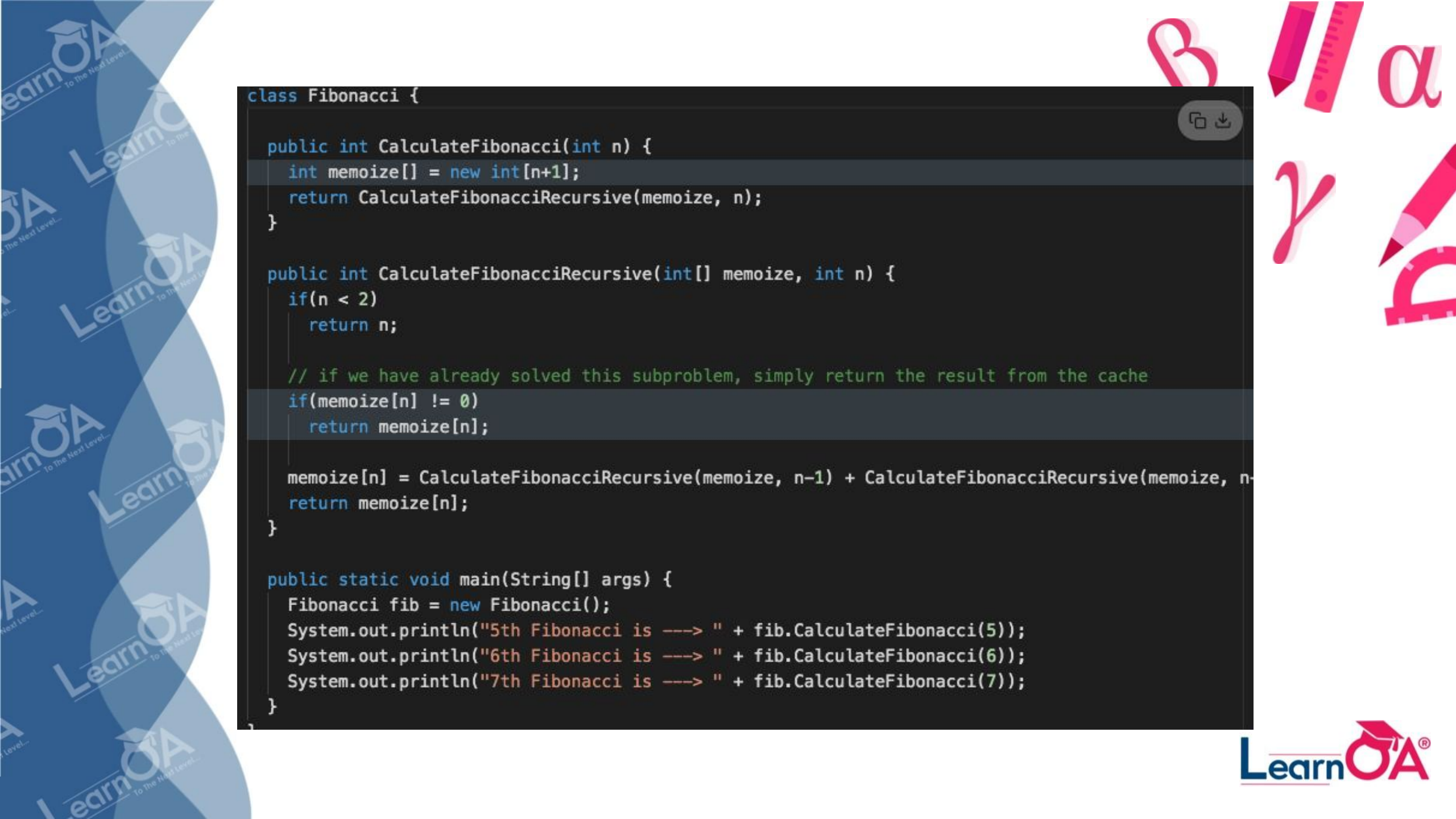
We'll see this technique in our example of Fibonacci numbers. First, let's see the non-DP recursive solution for finding the nth Fibonacci number:



```
class Fibonacci {  
  
    public int CalculateFibonacci(int n) {  
        if(n < 2)  
            return n;  
        return CalculateFibonacci(n-1) + CalculateFibonacci(n-2);  
    }  
  
    public static void main(String[] args) {  
        Fibonacci fib = new Fibonacci();  
        System.out.println("5th Fibonacci is ---> " + fib.CalculateFibonacci(5));  
        System.out.println("6th Fibonacci is ---> " + fib.CalculateFibonacci(6));  
        System.out.println("7th Fibonacci is ---> " + fib.CalculateFibonacci(7));  
    }  
}
```

As we saw, this problem shows the overlapping sub-problems pattern, so let's make use of memoization here. We can use an array to store the already solved sub-problems.





```
class Fibonacci {  
  
    public int CalculateFibonacci(int n) {  
        int memoize[] = new int[n+1];  
        return CalculateFibonacciRecursive(memoize, n);  
    }  
  
    public int CalculateFibonacciRecursive(int[] memoize, int n) {  
        if(n < 2)  
            return n;  
  
        // if we have already solved this subproblem, simply return the result from the cache  
        if(memoize[n] != 0)  
            return memoize[n];  
  
        memoize[n] = CalculateFibonacciRecursive(memoize, n-1) + CalculateFibonacciRecursive(memoize, n-2);  
        return memoize[n];  
    }  
  
    public static void main(String[] args) {  
        Fibonacci fib = new Fibonacci();  
        System.out.println("5th Fibonacci is ----> " + fib.CalculateFibonacci(5));  
        System.out.println("6th Fibonacci is ----> " + fib.CalculateFibonacci(6));  
        System.out.println("7th Fibonacci is ----> " + fib.CalculateFibonacci(7));  
    }  
}
```



# Dynamic Programming - Methods



## Bottom-up with Tabulation

Tabulation is the opposite of the top-down approach and avoids recursion. In this approach, we solve the problem “bottom-up” (i.e. by solving all the related sub-problems first). This is typically done by filling up an n-dimensional table. Based on the results in the table, the solution to the top/original problem is then computed.

Tabulation is the opposite of Memoization, as in Memoization we solve the problem and maintain a map of already solved sub-problems. In other words, in memoization, we do it top-down in the sense that we solve the top problem first (which typically recurses down to solve the sub-problems).

Let's apply Tabulation to our example of Fibonacci numbers. Since we know that every Fibonacci number is the sum of the two preceding numbers, we can use this fact to populate our table.

```
class Fibonacci {

    public int CalculateFibonacci(int n) {
        int dp[] = new int[n+1];

        //base cases
        dp[0] = 0;
        dp[1] = 1;

        for(int i=2; i<=n; i++)
            dp[i] = dp[i-1] + dp[i-2];

        return dp[n];
    }

    public static void main(String[] args) {
        Fibonacci fib = new Fibonacci();
        System.out.println("5th Fibonacci is ----> " + fib.CalculateFibonacci(5));
        System.out.println("6th Fibonacci is ----> " + fib.CalculateFibonacci(6));
        System.out.println("7th Fibonacci is ----> " + fib.CalculateFibonacci(7));
    }
}
```

# KnapSack Problem

Given the weights and profits of 'N' items, we are asked to put these items in a knapsack which has a capacity 'C'.

The goal is to get the maximum profit from the items in the knapsack. Each item can only be selected once, as we don't have multiple quantities of any item. Let's take the example of Merry, who wants to carry some fruits in the knapsack to get maximum profit.

Here are the weights and profits of the fruits:

Items: { Apple, Orange, Banana, Melon }

Weights: { 2, 3, 1, 4 }

Profits: { 4, 5, 3, 7 }

Knapsack capacity: 5



# KnapSack Problem

Let's try to put different combinations of fruits in the knapsack, such that their total weight is not more than 5:

Apple + Orange (total weight 5) => 9 profit

Apple + Banana (total weight 3) => 7 profit

Orange + Banana (total weight 4) => 8 profit

Banana + Melon (total weight 5) => 10 profit

This shows that Banana + Melon is the best combination, as it gives us the maximum profit and the total weight does not exceed the capacity.

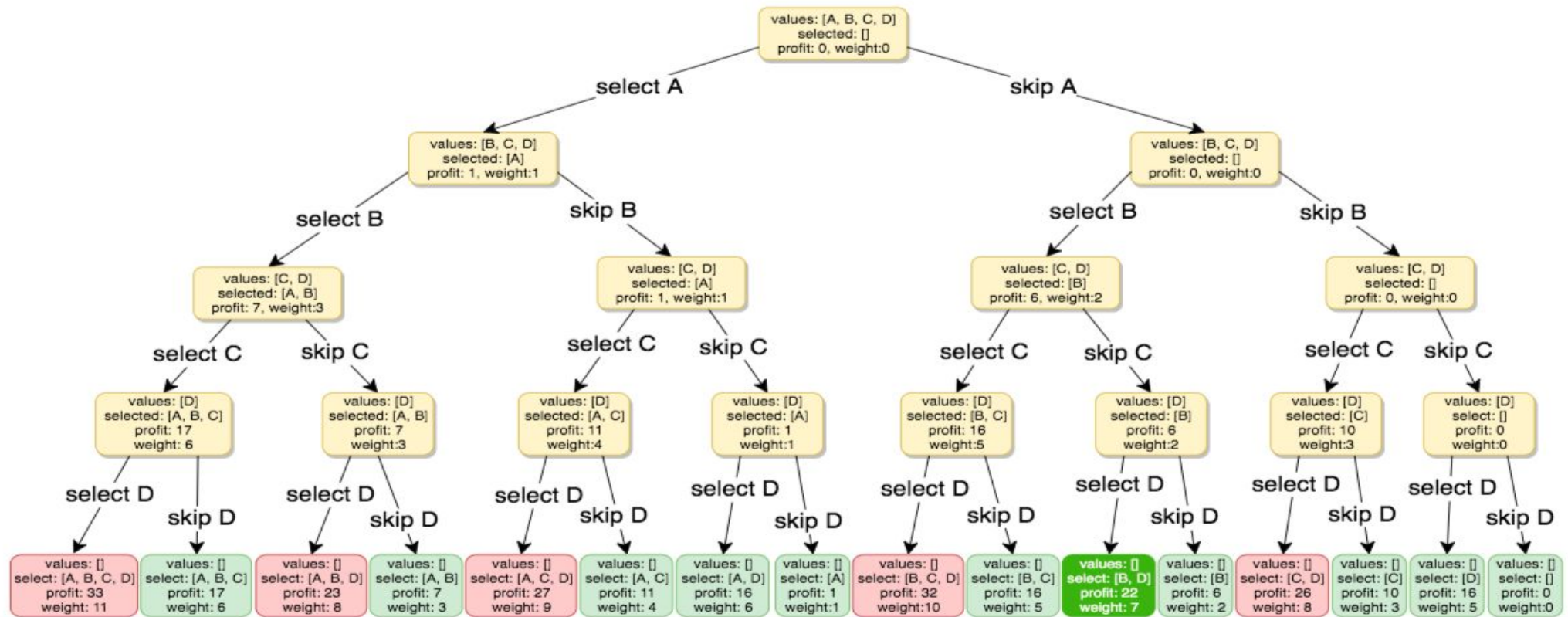


# KnapSack Brute Force



items	A	B	C	D
profit	1	6	10	16
weight	1	2	3	5

Capacity: 7





# KnapSack Problem

Implement Dynamic Programming to Solve the Same



# Rod Cutting Problem

Given a rod of length  $n$  and list of prices of rod of length  $i$  where  $1 \leq i \leq n$ , find the optimal way to cut rod into smaller rods in order to maximize profit



# Rod Cutting Problem

For example, consider below rod lengths and values

**Input:**

`length[] = [1, 2, 3, 4, 5, 6, 7, 8]`

`price [] = [1, 5, 8, 9, 10, 17, 17, 20]`

Rod length: 4

**Best:** Cut the rod into two pieces of length 2 each to gain revenue of  $5 + 5 = 10$

Cut	Profit
4	9
1, 3	$(1 + 8) = 9$
2, 2	$(5 + 5) = 10$
3, 1	$(8 + 1) = 9$
1, 1, 2	$(1 + 1 + 5) = 7$
1, 2, 1	$(1 + 5 + 1) = 7$
2, 1, 1	$(5 + 1 + 1) = 7$
1, 1, 1, 1	$(1 + 1 + 1 + 1) = 4$



# Rod Cutting Problem

Implement Dynamic Programming to Solve the Same



# Equal Subset Sum Partition

Given a set of positive numbers, find if we can partition it into two subsets such that the sum of elements in both the subsets is equal.

Example 1:

Input: {1, 2, 3, 4}

Output: True

Explanation: The given set can be partitioned into two subsets with equal sum: {1, 4} & {2, 3}



## More Examples

### Example 2:

Input: {1, 1, 3, 4, 7}

Output: True

Explanation: The given set can be partitioned into two subsets with equal sum: {1, 3, 4} & {1, 7}

### Example 3:

Input: {2, 3, 4, 6}

Output: False

Explanation: The given set cannot be partitioned into two subsets with equal sum.

Now, Implement Dynamic Programming to Solve the Same



## Find Optimal Cost to Construct Binary Search Tree

Find optimal cost to construct binary search tree where each key can repeat several times. We are given frequency of each key in same order as corresponding keys in inorder traversal of a binary search tree. In order to construct a binary search tree, for each given key, we have to find out if key already exists

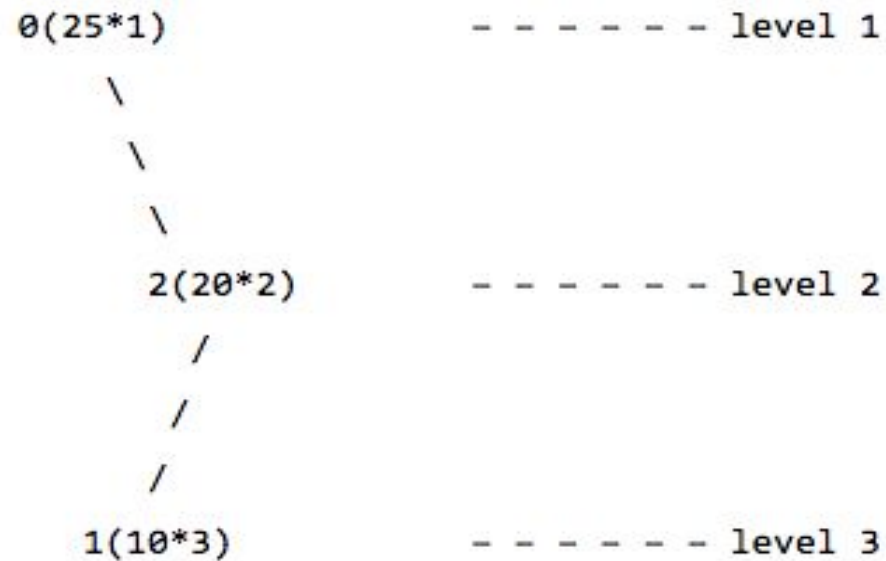
For example, consider below frequency array

`freq[] = { 25, 10, 20 }`

As frequency follows inorder order (ascending keys), let's consider index of `freq[]` as corresponding keys i.e

- key 0 occurs 25 times
- key 1 occurs 10 times
- key 2 occurs 20 times

Output: The optimal cost of constructing BST is 95.  
Below is the optimum BST.



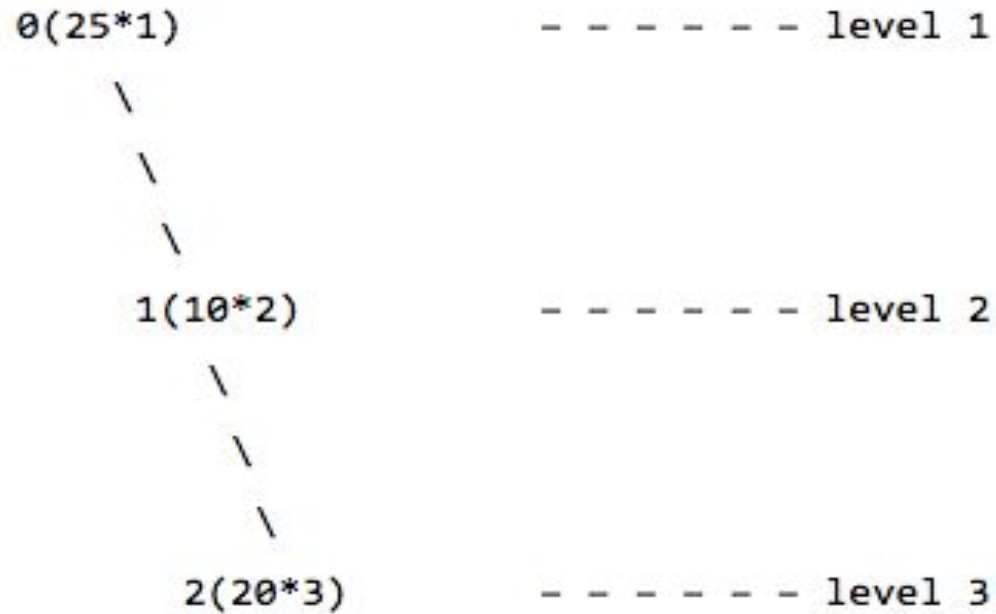
25 searches of key 0 will cost 1 each,  
20 searches of key 2 will cost 2 each,  
10 searches of key 1 will cost 3 each.

So,

Optimal Cost is:  $25*1 + 20*2 + 10*3 = 95$



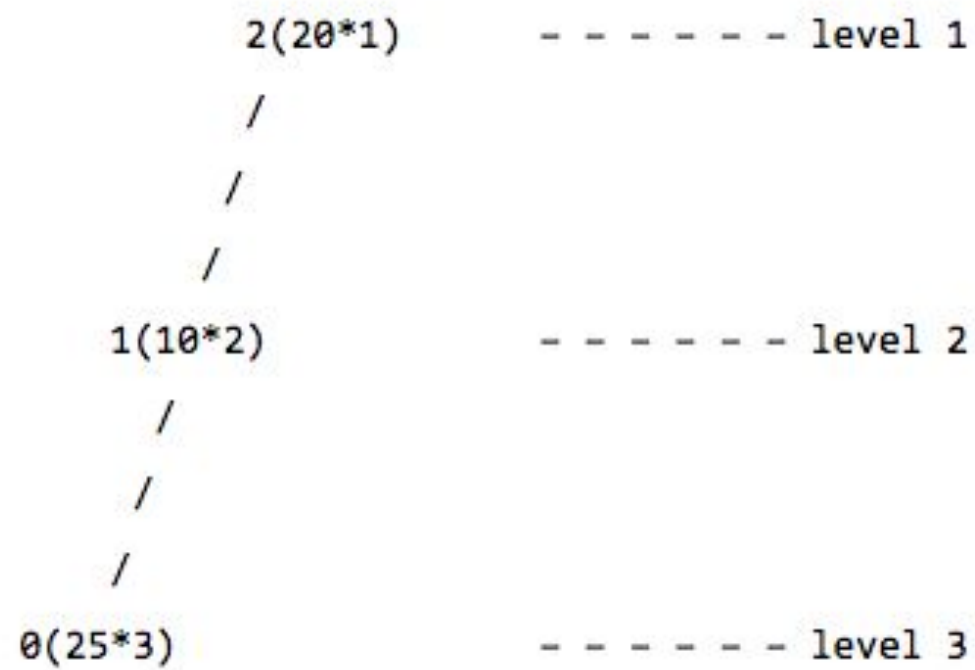
Other possible BSTs are -



Cost is:  $25 + 10*2 + 20*3 = 105$

which is more than the optimal cost 95





Cost is:  $20 + 10*2 + 25*3 = 115$

which is more than the optimal cost 95

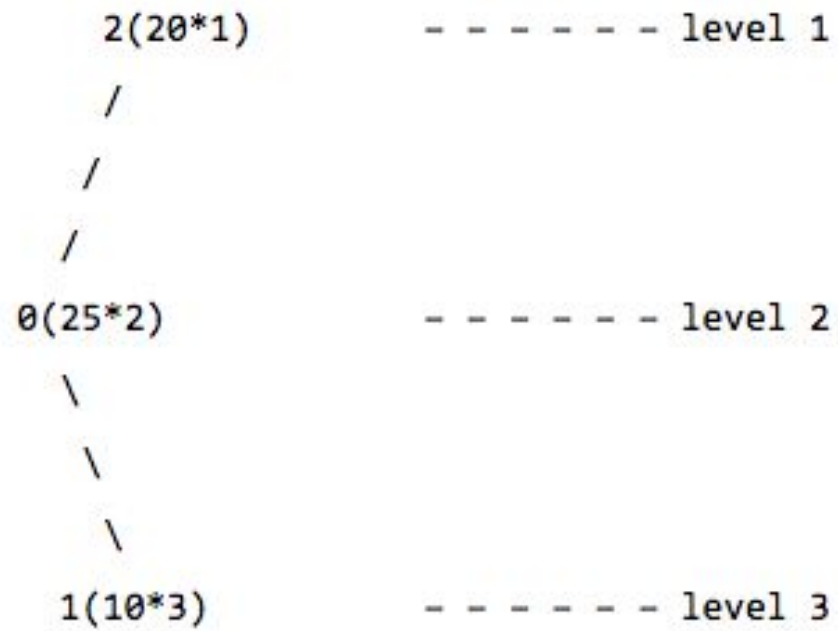






Cost is:  $10 + 25*2 + 20*2 = 100$

which is more than the optimal cost 95



Cost is:  $20 + 25*2 + 10*3 = 100$

which is more than the optimal cost 95

Now, Implement Dynamic Programming to Solve the Same

Thank You!

