Lesson:

Dynammic Programming-1

Pre Requisites:

- Recursion
- Basic JAVA syntax

List of concepts involved :

- Introduction to DP
- DP Vs Greedy approach
- Fibonacci series
- staircase problem
- Coin change problem
- 0-1 knapsack problem

Introduction to Dynamic programming :

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. Dynamic programming guarantees to find the optimal solution of a problem if the solution exists. Dynamic programming is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

Why do greedy algorithms fail ?

Let's understand this through an example :

Given a value of V Rs and an infinite supply of each of the denominations {1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, The task is to find the minimum number of coins and/or notes needed to make the change?

Examples:
Input: V = 70
Output: 2
Explanation: We need a 50 Rs note and a 20 Rs note.

Input: V = 121
Output: 3
Explanation: We need a 100 Rs note, a 20 Rs note, and a 1 Rs coin.

Approach :

- The intuition would be to take coins with greater value first. This can reduce the total number of coins needed.
- Start from the largest possible denomination and keep adding denominations while the remaining value is greater than 0.

Let's change the denomination of coins to {1 , 5 , 6 , 9}. And the sum we want to make is 11. Now as per greedy

algorithm it will return the answer as 3 coins. [9 + 1 + 1]. But the minimum number of coins could be 2 [5 + 6].

This shows greedy algorithms won't always work.

Dynamic programming is the technique that discovers every single possibility and then comes up with a solution which is optimal.

Fibonacci series :

Q1. Given an integer n. Print the first n numbers of the fibonacci series. (N is chosen such that its doesn't overflow integer range)
Input 1 : n = 5
Output 1: 0 1 1 2 3

Input 2 : n = 8
Output 2 : 0 1 1 2 3 5 8 13

Solution :

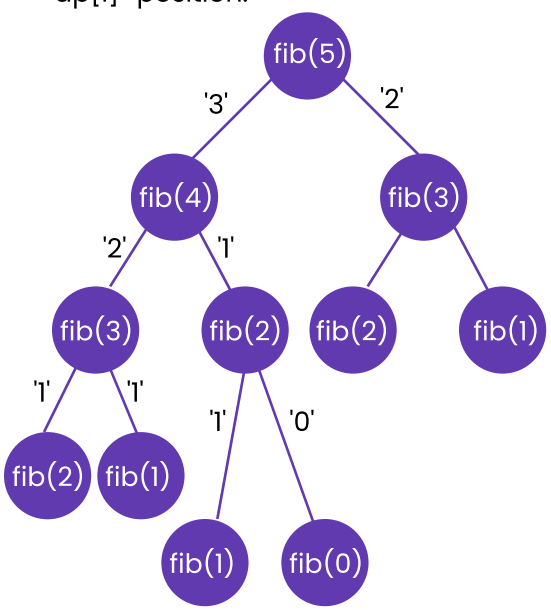
Code : LP_Code1.java

Output :

```
0 1 1 2 3 5 8 13 21 34 55 89
```

Approach :

- We have already seen the recurrence relation for fibonacci series during recursion.
- it was f(i) = f(i-1) + f(i-2);
- For calculating any ith term we first need the sum of two previous terms. If we are not saving the previous result anywhere then it will take 2 steps to count the previous two terms, then 4 other steps to count 2 steps of both the previous terms and so on.
- This process will end up in a exponential time complexity where we are currently calculating the nth term.
- Therefore we used dynamic programming to store the result obtained at each stage so that it can be used later on for further calculations.
- For storing we used an array of "n+1" size and after calculating the result of any ith stage we stored it at "dp[i]" position.



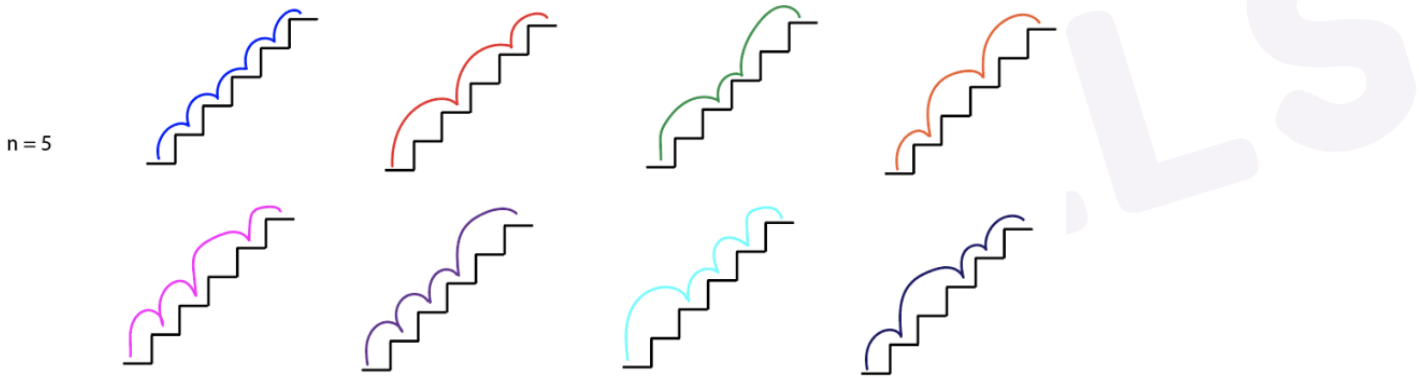
Staircase problem:

Q2. There are n stairs, a person standing at the bottom wants to reach the top. The person can climb either 1 stair or 2 stairs at a time. Count the number of ways the person can reach the top.

Examples:
Input: n = 1
Output: 1
There is only one way to climb 1 stair

Input: n = 2
Output: 2
There are two ways: (1, 1) and (2)

Input: n = 4
Output: 5
(1, 1, 1, 1), (1, 1, 2), (2, 1, 1), (1, 2, 1), (2, 2)



Possibilities for n = 5.

Solution :

Code : LP_Code2.java

Output :

```
The possible ways are : 8
```

Approach :

- The key idea behind this problem (like other recursive problems) is to recognize the pattern so that you could divide it into sub-problems or find a pattern.
- To write a recursive equation, just imagine that you are standing at the top and there are only 2 ways to reach from stairs: 1) by taking 1 step, or 2) by taking two steps. So the recursive equation should be :
- T(n) = T(n-1) + T(n-2)
- The base condition could be thought as :
- T(1) = 1 ------> 1 way of taking 1 stair.
- T(0) = 1 ------> 1 way of not taking any stairs.
- The above is a fibonacci series that may be solved using DP.

Coin change problem :

Q3. Given an integer array of coins[] of size N representing different types of currency and an integer sum, The task is to find the number of ways to make a sum by using different combinations from coins[]. If the target amount is unreachable, return -1.
Note: Assume that you have an infinite supply of each type of coin.

Examples:
Input: sum = 4, coins[] = {1,2,3},
Output: 4
Explanation: there are four solutions: {1, 1, 1, 1}, {1, 1, 2}, {2, 2}, {1, 3}.

Input: sum = 10, coins[] = {2, 5, 3, 6}
Output: 5
Explanation: There are five solutions:
{2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}.

Solution :

Code : LP_Code3.java

Output :

```
The minimum number of coins are : 3
```

Approach :

- We can consider the possible sums from 0 -> amount, and we update the number of coins required at that sum if we find a more optimal solution.
- As with all DP problems, it's ideal to first consider the optimal subproblems:
- At sum 0, 0 coins are required
- If we do not take coins[i], it means the amount of coins to reach this sum remains the same
- If we take coins[i], it means the amount of coins to reach this sum is the amount that it took to reach sums[j-coins[i]] + 1. Essentially, the amount of coins to reach the last sum without this current coin + 1, since the current coin is an additional coin used.
- **sums[i] = Math.min(sums[j], sums[sums[j] - coins[i]]+1)**

- Thus, we iterate through every denomination of coins that we have, and we check whether it's possible to generate a sum from 0 -> amount with the current coin.
- If it is possible, then we take the minimum amount of coins possible. Thus, this algorithm is guaranteed to produce the minimum coins possible, since the logic of using the minimum coins to reach each sum is applied for each coin value.
- FInally, we should return sums[sums.length-1] since that should store the minimum amount of coins required to reach this sum, which is equal to the target amount. If it is Integer.MAX_VALUE, then it is impossible.

0-1 knapsack problem :

Q4. We are given N items where each item has some weight and profit associated with it. We are also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The target is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Note: The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

Examples:
Input: N = 3, W = 4, profit[] = {1, 2, 3}, weight[] = {4, 5, 1}
Output: 3
Explanation: There are two items which have weight less than or equal to 4. If we select the item with weight 4, the possible profit is 1. And if we select the item with weight 1, the possible profit is 3. So the maximum possible profit is 3. Note that we cannot put both the items with weight 4 and 1 together as the capacity of the bag is 4.

Input: N = 3, W = 3, profit[] = {1, 2, 3}, weight[] = {4, 5, 6}
Output: 0

Solution :

Code : LP_Code4.java

Output :

```
The maximum achievable profit is : 411
```

Approach :

- A simple solution is to consider all subsets of items and calculate the total weight and profit of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the subset with maximum profit.
- To consider all subsets of items, there can be two cases for every item:
  - Case 1: The item is included in the optimal subset.
  - Case 2: The item is not included in the optimal set.
- In a DP[][] table let's consider all the possible weights from '1' to 'W' as the columns and the elements that can be kept as rows.
- The state DP[i][j] will denote the maximum value of 'j-weight' considering all values from '1 to ith'. So if we consider 'wi' (weight in 'ith' row) we can fill it in all columns which have 'weight values > wi'. Now two possibilities can take place:
- Fill 'wi' in the given column.
- Do not fill 'wi' in the given column.
- Now we have to take a maximum of these two possibilities,
- Formally if we do not fill the 'ith' weight in the 'jth' column then the DP[i][j] state will be the same as DP[i-1][j]
- But if we fill the weight, DP[i][j] will be equal to the value of ('wi'+ value of the column weighing 'j-wi') in the previous row.
- So we take the maximum of these two possibilities to fill the current state.

Let, weight[] = {1, 2, 3}, profit[] = {10, 15, 40}, Capacity = 6

If no element is filled, then the possible profit is 0.

item ↓ / weight --->	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1							
2							
3							

For filling the first item in the bag: If we follow the above mentioned procedure, the table will look like the following.

item ↓ / weight --->	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2							
3							

For filling the second item:
When j = 2, then maximum possible profit is max (10, DP[1][2-2] + 15) = max(10, 15) = 15.
When j = 3, then maximum possible profit is max(2 not put, 2 is put into bag) = max(DP[1][3], 15+DP[1][3-2]) = max(10, 25) = 25.

item ↓ / weight --->	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3							

For filling the third item:
When j = 3, the maximum possible profit is max(DP[2][3], 40+DP[2][3-3]) = max(25, 40) = 40.
When j = 4, the maximum possible profit is max(DP[2][4], 40+DP[2][4-3]) = max(25, 50) = 50.
When j = 5, the maximum possible profit is max(DP[2][5], 40+DP[2][5-3]) = max(25, 55) = 55.
When j = 6, the maximum possible profit is max(DP[2][6], 40+DP[2][6-3]) = max(25, 65) = 65.

item ↓ / weight --->	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65