PW SKILLS

# Lesson Plan

# Introduction to DP

PW SKILLS

# Prerequisites:

- Arrays
- Recursion

# Today's Checklist

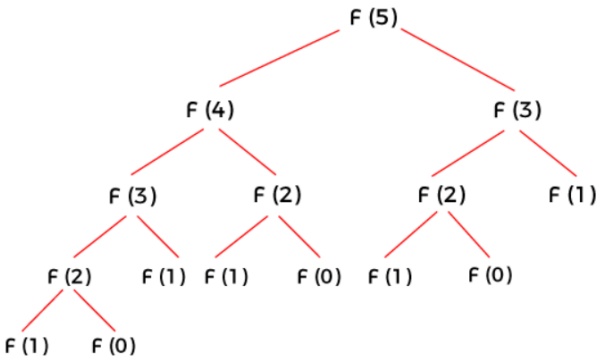
- What is Dynamic Programming?
- What are overlapping subproblems?
- What is an optimal substructure?
- Greedy vs DP
- Where to use dynamic programming?
- What are the various approaches of DP?
- What is memoization?
- Steps of solving a DP problem
- Problems of Memoization
- What is the Limitation of Top-DownProgramming?
- What is Tabulation?
- How is tabulation different from memoization?
- Fibonacci with tabulation
- Problems of tabulation

# What is Dynamic Programming

- Dynamic Programming (DP) is defined as a technique that solves some particular type of problems in Polynomial Time.
- It 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.
- It is mainly an optimization over plain recursion.
- Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming.
- The idea is to simply store the results of subproblems so that we do not have to re-compute them when needed later.
- This simple optimization reduces time complexities from exponential to polynomial.

# What are overlapping subproblems?

- A problem is said to have overlapping subproblems if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems.
- For example: consider the recursive tree to calculate the fibonacci sequence upto n = 5.




The problem of computing the nth Fibonacci number F(n), can be broken down into the subproblems of computing F(n - 1) and F(n - 2), and then adding the two. The subproblem of computing F(n - 1) can itself be broken down into a subproblem that involves computing F(n - 2). Therefore, the computation of F(n - 2) is reused, and the Fibonacci sequence thus exhibits overlapping subproblems.

- Dynamic Programming is mainly used when solutions to the same subproblems are needed again and again.
- Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point in storing the solutions if they are not needed again.

# What is an optimal substructure?

- A problem is said to have Optimal Substructure if the optimal solution of the given problem can be obtained by using the optimal solution to its subproblems instead of trying every possible way to solve the subproblems.
- For example: Consider finding a shortest path for traveling between two cities by car. Such an example is likely to exhibit optimal substructure. That is, if the shortest route from Seattle to Los Angeles passes through Portland and then Sacramento, then the shortest route from Portland to Los Angeles must pass through Sacramento too. That is, the problem of how to get from Portland to Los Angeles is nested inside the problem of how to get from Seattle to Los Angeles.
- Typically, a greedy algorithm is used to solve a problem with optimal substructure, otherwise Dynamic Programming is used.
- Whereas, a non-optimal substructure is like N-Queens problem, where you can't extrapolate solution of N-1 size chess set to get the solution of N size chess cell.

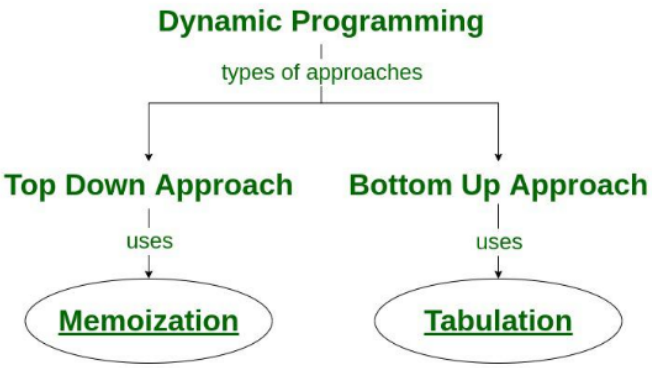
# Greedy Algorithm VS Dynamic Programming Algorithm

| GREEDY | DYNAMIC PROGRAMMING |
|---|---|
| A Greedy algorithm builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. | A dynamic programming algorithm builds up the solution to a problem by solving its subproblems recursively. |
| It is useful for solving problems where making locally optimal choices at each step leads to a global optimum. | It is used where the optimal solution can be obtained by combining optimal solutions to subproblems. |
| It does not necessarily consider the future consequences of the current choice. | Dynamic programming stores the solutions to subproblems and reuses them when necessary to avoid solving the same subproblems multiple times. |
| The greedy approach is generally faster and simpler, but may not always provide the optimal solution. | Dynamic programming guarantees the optimal solution but is slower and more complex. |

# Where to use Dynamic Programming?

- 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. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.
- Dynamic programming is useful for solving problems where the optimal solution can be obtained by combining optimal solutions to subproblems.

# What are the various approaches of DP?




- Top-Down(Memoization): Break down the given problem in order to begin solving it. If you see that the problem has already been solved, return the saved answer. If it hasn't been solved, solve it and save it. This is usually easy to think of and very intuitive, This is referred to as Memoization.
- Bottom-Up(Tabulation): In this approach, the problem is solved by solving its subproblems in a bottom-up manner, starting from the smallest subproblem and progressively solving larger subproblems until the main problem is solved.
- The bottom-up approach typically uses iteration or loops to solve the problem and build up the solution step by step.

# What is Memoization?

- Memoization is a specific form of caching that is used in dynamic programming that is used to speed up execution time by eliminating the repetitive computation of results, and by avoiding repeated calls to functions that process the same input.
- It basically stores the previously calculated result of the subproblem and uses the stored result for the same subproblem.
- This removes the extra effort to calculate again and again for the same problem.
- For example, consider the below recursive part of the code to calculate factorial of a number:

```
public int factorial(int n) {
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

If you write the complete code for the above snippet, you will notice that there will be 2 methods in the code: factorial(n) and main().
Now if we have multiple queries to find the factorial, such as finding factorial of 2, 3, 9, and 5, then we will need to call the factorial() method 4 times and each time the factorial method goes down to calculating the factorial of each number from n to 1 to compute the final result.
So for finding factorial of numbers K numbers, the time complexity needed will be O(N*K)where O(N) to find the factorial of a particular number, and O(K) to call the factorial() method K different times.

Let's see how we can optimize the time complexity here:
If we notice in the above problem, while calculation factorial of 9:
We are calculating the factorial of 2, We are also calculating the factorial of 3, and We are calculating the factorial of 5 as well.
Therefore if we store the result of each individual factorial at the first time of calculation, we can easily return the factorial of any required number in just O(1) time. This process is known as Memoization.

### How Memoization works?

If we find the factorial of 9 first and store the results of individual sub-problems, we can easily print the factorial of each input in O(1).
Therefore the time complexity to find factorial numbers using memorization will be O(N): O(N) to find the factorial of the largest input and O(1) to print the factorial of each input.

# Steps of solving a DP problem

1. **Identify if it is a Dynamic programming problem:**
    - Typically, all the problems that require maximizing or minimizing certain quantities or counting problems that say to count the arrangements under certain conditions or certain probability problems
    - Problems that satisfy the overlapping subproblems property and most of the DP problems also satisfy the optimal substructure property.
2. **Deciding the state**
    - What is the state?
    - Since generally, a state is the particular condition that something is in at a specific point of time, similarly, in Dynamic Programming, a state is defined by a number of necessary variables at a particular instant that are required to calculate the optimal result.
    - In other words, state is the minimum set of parameters needed to uniquely define a sub-problem.
    - It is a combination of variables that will keep changing over different instants.
    - Two states are the same, if all their corresponding variables have the same logical value.
    - In order to verify if the state is minimal or not, try to calculate or derive one of the parameters of state with the help of others, if it's possible to do so, then remove that parameter.
3. **Formulating relation among the states**
    - This is the most crucial and tough part of solving a DP problem.
    - Let us understand it through an example:

Given 3 numbers {1, 3, 5}, the task is to tell the total number of ways we can form a number N using the sum of the given three numbers. (allowing repetitions and different arrangements).

Let's try to solve this problem for N = 6.

Let's look at the output first:
The total number of ways to form 6 is: 8

1+1+1+1+1+1
1+1+1+3
1+1+3+1
1+3+1+1
3+1+1+1
3+3
1+5
5+1

As we can only use 1, 3, or 5 to form a given number N, let us assume that we know the result for N = 1, 2, 3, 4, 5, 6.

Let us say we know the result for state (n = 1), state (n = 2), state (n = 3) ..... state (n = 6)
Now, we wish to know the result of the state (n = 7).

Now we can get a sum total of 7 in the following 3 ways:
Adding 1 to all possible combinations of state (n = 6)
Adding 3 to all possible combinations of state (n = 4)
Adding 5 to all possible combinations of state(n = 2)

Now, think carefully that the above three cases are covering all possible ways to form a sum total of 7.
Therefore, we can say that result for state(7) = state (6) + state (4) + state (2)
OR
state(7) = state (7-1) + state (7-3) + state (7-5)
In general,
state(n) = state(n-1) + state(n-3) + state(n-5)

This is how we pick a state and then formulate relationships amongst them.

Let's take a look at the recursive function for the above example:

```
public static int solve(int n) {
       // base case
       if (n < 0)
           return 0;
       if (n == 0)
           return 1;

       return solve(n - 1) + solve(n - 3) + solve(n - 5);
   }
```

**4. Adding memoization for the state**

- The simplest portion of a solution based on dynamic programming is this.
- Simply storing the state solution will allow us to access it from memory the next time that state is needed.
- Let's add memoization to the above example

```
static final int MAX_VALUE = 10000; // Choose a suitable value for
the dynamic array
    static int[] dp = new int[MAX_VALUE];

    public static int solve(int n) {
        // base case
        if (n < 0)
            return 0;
        if (n == 0)
            return 1;

        // Checking if already calculated
        if (dp[n] != -1)
            return dp[n];

        // Storing the result and returning
        return dp[n] = solve(n - 1) + solve(n - 3) + solve(n - 5);
    }`
```

# Problems on Memoization

**Q1.** The Fibonacci numbers, commonly denoted F(n) form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

F(0) = 0, F(1) = 1
F(n) = F(n - 1) + F(n - 2), for n > 1.

Given n, calculate F(n). 0 <= n < 30.

**Input1:** n = 2
**Output1:** 1

**Explanation:** F(2) = F(1) + F(0) = 1 + 0 = 1.

**Input2:** n = 3
**Output2:** 2

**Explanation:** F(3) = F(2) + F(1) = 1 + 1 = 2.

**Explanation:**

- Recursively, the approach is easy. We just need to return fib(n-1) + fib(n-2) with the base case as if n == 0 || n == 1, return n.
- Now we optimize this solution by adding memoization to it.
- First we create a static array of size 30 as n can be maximum up to 30. This array will be static so that we don't have to pass it as a parameter to the recursive function.
- Initialize the array with -1 for all indices so that, we can spot if the answer for the current index has been computed already or not.
- Now instead of returning the answer in the recursive function, we store it in the array.
- First check if current index is computed yet or not.
- If not, for the base case, if n <= 1 meaning if n = 0 or 1, store the same value for the same index in the array.
- We call the recursive function as is for n-1 and n-2, just store it in the array at index N.
- If value for current index is already computed, return value of current index in the array.

**CODE:**

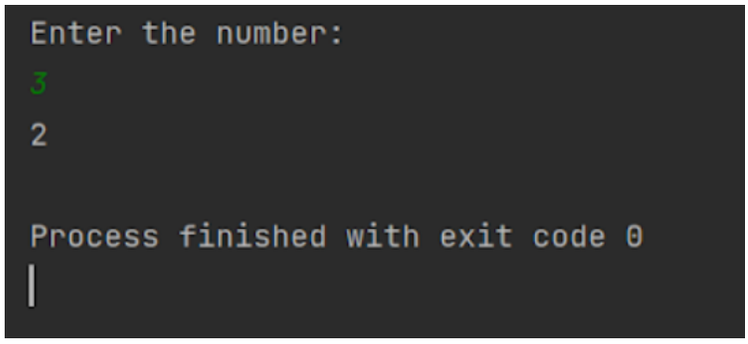
```
import java.util.Arrays;
import java.util.Scanner;

public class Main {
    static int[] arr = new int[30];

    public static void main(String[] args) {
        Arrays.fill(arr, -1); // Initializing arr array with -1
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number: ");
        int N = scanner.nextInt();
        System.out.println(Fibonacci(N));
        scanner.close();
    }

    public static int Fibonacci(int N) {
        // If the subproblem is not computed yet, recursively
compute and store the result
        if (arr[N] == -1) {
            if (N <= 1)
                arr[N] = N;
            else
                arr[N] = Fibonacci(N - 1) + Fibonacci(N - 2);
        }
        // Otherwise, just return the result
        return arr[N];
    }
}
```

**OUTPUT:**




**Q2.** You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.
Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

**Input1:** nums = [1,2,3,1]
**Output1:** 4

**Explanation:** Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total amount you can rob = 1 + 3 = 4.

**Input2:** nums = [2,7,9,3,1]
**Output2:** 12

**Explanation:** Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
Total amount you can rob = 2 + 9 + 1 = 12.

**Explanation:**

- There are two choices in front of each house: rob or not rob.
- Consider standing in front of a house. If you rob this house, then you definitely can't rob the adjacent houses, you can only start the next rob from the house after next.
- If you don't rob this house, then you can walk to the next house and continue making choices.
- When you walk past the last house, you don't have to rob. The money you could rob is obviously 0 (base case).
- In these two choices, you need to choose a larger result each time. You end up with the most money you can rob.
- To add memoization, we create an array of size N+1 with all indices as -1.
- In the recursive function, we pass the input array and an index variable as parameters.
- If index goes out of bounds, we return 0.
- If answer for current index has been computed, return that.
- Else, update answer for current index as max of i-1 or i-2 + houses[i].

**CODE:**

```
import java.util.Scanner;

public class Main {
    static int[] arr;

    public static int rob(int[] houses, int i) {
        if (i < 0) {
            return 0;
        }
        if (arr[i] ⩾ 0) {
            return arr[i];
        }
        int result = Math.max(rob(houses, i - 2) + houses[i],
rob(houses, i - 1));
        arr[i] = result;
        return result;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of houses: ");
        int N = scanner.nextInt();

        int[] houses = new int[N];
        System.out.print("Enter the money of each house: ");
        for (int i = 0; i < N; i++) {
            houses[i] = scanner.nextInt();
        }

        arr = new int[N + 1];
        for (int i = 0; i ⩽ N; i++) {
            arr[i] = -1;
        }

        System.out.println(rob(houses, N - 1));
    }
}
```

**OUTPUT:**

```
Enter the number of houses:
4
Enter the money of each house:
1 2 3 1
4

Process finished with exit code 0
```

**OUTPUT:**

```
Enter the number:
3
2

Process finished with exit code 0
```

**Q3.** A message containing letters from A-Z can be encoded into numbers using the following mapping:
'A' -> "1"
'B' -> "2"
...
'Z' -> "26"

To decode an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, "11106" can be mapped into:
"AAJF" with the grouping (1 1 10 6)
"KJF" with the grouping (11 10 6)
Note that the grouping (1 11 06) is invalid because "06" cannot be mapped into 'F' since "6" is different from "06".
Given a string s containing only digits, return the number of ways to decode it.

**Input1:** s = "12"
**Output1:** 2

**Explanation:** "12" could be decoded as "AB" (1 2) or "L" (12).

**Input2:** s = "226"
**Output2:** 3

**Explanation:** "226" could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).

**Explanation:**

- Intuition behind the recursive solution: We take out 1 letter and 2 letters at a time from the string. We then check if the letters taken out are valid (can be decoded), then we recurse the function without those letter(s), otherwise assign a 0 to that component if its invalid.
- In the code below, our base case is defined as idx == s.length() which is the same as no characters left in the string anymore. We then check if we have a stored value of this substring and just return that. Otherwise, we first take out one letter, check the letter's validity and then call the function again without this letter that we took out. Once that step is completed, we then move on to doing the same thing by taking out 2 letters this time.
- Before working for 2 letters, we need to check if we have anymore letters remaining and also, if the first letter we took out was 1 or 2. This is because the numbers in our range are till 26 only.

**CODE:**

```
import java.util.ArrayList;
import java.util.Scanner;

public class Main {
    static ArrayList<Integer> arr;

    public static int decode(String s, int idx) {
        if (idx == s.length())
            return 1;
        if (s.charAt(idx) == '0')
            return 0;
        if (arr.get(idx) != -1)
            return arr.get(idx);

        int res = decode(s, idx + 1);
        if (idx < s.length() - 1 && (s.charAt(idx) == '1' ||
(s.charAt(idx) == '2' && s.charAt(idx + 1) < '7')))
            res += decode(s, idx + 2);

        arr.set(idx, res);
        return res;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the string: ");
        String s = scanner.nextLine();
        int n = s.length();

        arr = new ArrayList<>(n + 1);
        for (int i = 0; i <= n; i++) {
            arr.add(-1);
        }

        System.out.println(decode(s, 0));
    }
}
```

**OUTPUT:**

```
Enter the string:
12
2


Process finished with exit code 0
```

# What is the Limitation of Top-DownProgramming?

- Since the process starts with the big picture, it can be difficult to make changes later on in the process.
- Breaking down the bigger problem into smaller subproblems can take a lot of time, especially if the design is complex.
- It requires massive amounts of expensive recursion.

# What is Tabulation?

- Tabulation is a bottom-up approach where we store the results of the subproblems in a table and use these results to solve larger subproblems until we solve the entire problem.
- It is used when we can define the problem as a sequence of subproblems and the subproblems do not overlap.
- Tabulation is typically implemented using iteration and is well-suited for problems that have a large set of inputs.

# How is tabulation different from memoization?

- Tabulation has Iterative implementation as compared to memoization which follows a recursive implementation.
- Tabulation is used when the subproblems do not overlap whereas memoization is used for overlapping subproblems.
- Tabulation is well-suited for problems with a large set of inputs whereas memoization is well-suited for problems with a relatively small set of inputs.

# Example: Fibonacci sequence using Tabulation

**Ques:** Given an integer n, find the Fibonacci sequence upto n.
Fibonacci sequence till n = 5: 0 1 1 2 3

**Answer:** 3

**Explanation:**

- In the bottom-up dynamic programming approach, we'll reorganize the order in which we solve the subproblems.
- So, we start with the smallest problem and reach till the largest problem which is the given problem.
- For example: We'll compute n = 0, then n = 1 and so on till n = 5.
- Let's checkout the code using a table to store the previously computed results.

**CODE:**

```
import java.util.Scanner;

public class Main {
    public static int fib(int n) {
        int[] arr = new int[n];
        if (n == 0)
            return 0;
        if (n == 1)
            return 1;
        arr[0] = 0;
        arr[1] = 1;
        for (int i = 2; i < n; i++) {
            arr[i] = arr[i - 1] + arr[i - 2];
        }
        return arr[n - 1];
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number: ");
        int n = scanner.nextInt();

        System.out.println(fib(n));
    }
}
```

**OUTPUT:**

```
Enter the number:
5
3


Process finished with exit code 0
```

- Now, if we see, the above approach will only allow us to compute the solution to each problem only once, and we'll only need to save two intermediate results at a time.
- For example, when we're trying to find the answer for n = 2, we only need to have the solutions to n=1 and n=0 available. Similarly, for n = 3, we only need to have the solutions to n = 2 and n = 1.
- To implement this space optimization, we make use of only two variables, and these variables store the required 2 values at each point of iteration until we reach n.
- This will allow us to use less memory space in our code.
- This is called space-state reduction.

**CODE:**

```
import java.util.Scanner;

public class Main {
    public static int fib(int n) {
        int a = 0;
        int b = 1;
        if (n == 0)
            return a;
        if (n == 1)
            return b;
        for (int i = 2; i < n; i++) {
            int c = a + b;
            a = b;
            b = c;
        }
        return b;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number: ");
        int n = scanner.nextInt();

        System.out.println(fib(n));
    }
}
```

**OUTPUT:**

```
Enter the number:
5
3


Process finished with exit code 0
```

# PROBLEMS on Tabulation

**Q1.** 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[]. Assume that you have an infinite supply of each type of coin.

**Input1:** sum = 4, coins[] = {1,2,3},
**Output1:** 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

**Approach 1:**

```
import java.util.Scanner;

public class Main {
    public static int countCoinChange(int sum, int[] coins) {
        int n = coins.length;

        int[][] dp = new int[n + 1][sum + 1];

        for (int i = 0; i ⩽ n; i++) {
            dp[i][0] = 1;
        }

        for (int i = 1; i ⩽ n; i++) {
            for (int j = 1; j ⩽ sum; j++) {
                int include = (j ⩾ coins[i - 1]) ? dp[i][j -
coins[i - 1]] : 0;
                int exclude = dp[i - 1][j];
                dp[i][j] = include + exclude;
            }
        }

        return dp[n][sum];
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the target sum: ");
        int sum = scanner.nextInt();

        System.out.print("Enter the number of coins: ");
        int coinCount = scanner.nextInt();

        int[] coins = new int[coinCount];

        System.out.print("Enter the coins: ");
        for (int i = 0; i < coinCount; i++) {
            coins[i] = scanner.nextInt();
        }

        int ways = countCoinChange(sum, coins);
        System.out.println("Number of ways to make " + sum + "
using coins: " + ways);
    }
}
```

Now lets see better approach

**Explanation:**

- The Idea to Solve this Problem is by using the Bottom Up(Tabulation). By using the linear array for space optimization.
- Initialize with a linear array table with values equal to 0.
- With sum = 0, there is a way.
- Update the level wise number of ways of coin till the ith coin.
- Solve till j <= sum.

**Code link:**

```
import java.util.Arrays;

public class Main {
    public static int count(int[] coins, int n, int sum) {
        int[] table = new int[sum + 1];
        Arrays.fill(table, 0);

        table[0] = 1;

        for (int i = 0; i < n; i++) {
            for (int j = coins[i]; j ⩽ sum; j++) {
                table[j] += table[j - coins[i]];
            }
        }
        return table[sum];
    }

    public static void main(String[] args) {
        int[] coins = {1, 2, 3};
        int n = coins.length;
        int sum = 5;
        System.out.println(count(coins, n, sum));
    }
}
```

**Leetcode link**

**Q2.** 0/1 KNAPSACK
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.

**Input1:**
N = 3, W = 4, profit[] = {1, 2, 3}, weight[] = {4, 5, 1}
**Output1:** 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.

**Input2:** N = 3, W = 3, profit[] = {1, 2, 3}, weight[] = {4, 5, 6}
**Output2:** 0

**Explanation:**

- 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.
- The maximum value obtained from 'N' items is the max of the following two values.
- Maximum value obtained by N-1 items and W weight (excluding nth item):
- Value of nth item plus maximum value obtained by N-1 items and (W – weight of the Nth item) [including Nth item].
- If the weight of the 'Nth' item is greater than 'W', then the Nth item cannot be included and Case 1 is the only possibility.
- Since subproblems are evaluated again, this problem has Overlapping Sub-problems property and optimal substructure property.
- Re-computation of the same subproblems can be avoided by constructing a temporary array K[][] in a bottom-up manner.
- In a DP[][] table let's consider all the possible weights from '1' to 'W' as the columns and the element 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.

**Code:**

```
import java.util.Scanner;

public class Main {
    public static int knapSack(int W, int[] wt, int[] val, int n) {
        int[][] K = new int[n + 1][W + 1];
        for (int i = 0; i ⩽ n; i++) {
            for (int w = 0; w ⩽ W; w++) {
                if (i == 0 || w == 0)
                    K[i][w] = 0;
                else if (wt[i - 1] ⩽ w)
                    K[i][w] = Math.max(val[i - 1] + K[i - 1][w -
wt[i - 1]], K[i - 1][w]);
                else
                    K[i][w] = K[i - 1][w];
            }
        }
        return K[n][W];
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of items: ");
        int n = scanner.nextInt();
        System.out.print("Enter the capacity of the knapsack: ");
        int w = scanner.nextInt();

        int[] val = new int[n];
        int[] wt = new int[n];

        System.out.print("Enter the profit associated with each
item: ");
        for (int i = 0; i < n; i++) {
            val[i] = scanner.nextInt();
        }
        System.out.print("Enter weight associated with each item:
");
        for (int i = 0; i < n; i++) {
            wt[i] = scanner.nextInt();
        }

        System.out.println("Maximum profit possible is: " +
knapSack(w, wt, val, n));
    }
}
```

**OUTPUT:**

```
Enter the number of items :
3
Enter the capacity of the knapsack :
4
Enter the profit associated with each item :
1 2 3
Enter weight associated with each item:
4 5 1
Maximum profit possible is:
3

Process finished with exit code 0
```

**Q3.** There are N stones, numbered 1,2,...,N. The height of ith stone is hi.
There is a frog who is initially on Stone 1. He will repeat an action some number of times to reach Stone N. The action is that if the frog is currently on Stone i, it jumps to one of the following: Stone i+1,i+2,...,i+K. Here, a cost of |hi - hj| is incurred, where j is the stone to land on.
Find the minimum possible total cost incurred before the frog reaches Stone N.

**Input1:**
n = 5
k = 3
10 30 40 50 20
**Output1:**
30

**Input2:**
3 1
10 20 10
**Output2:**
20

**Explanation:**

- We create a dp array of the same length as the input array.
- Let's define dp[i] as the minimum cost to reach Stone i from Stone 1. Our goal is to find dp[N], which represents the minimum possible total cost incurred before the frog reaches Stone N.
- We know that the minimum cost to reach Stone 1 from Stone 1 is 0, so we set dp[1] = 0.
- To calculate dp[i], we need to consider all possible jumps the frog can make from the previous stones (i-1, i-2, ..., i-K). We choose the jump that incurs the minimum cost. Mathematically, we can express this as: dp[i] = min(dp[i-1] + |hi - hi-1|, dp[i-2] + |hi - hi-2|, ..., dp[i-K] + |hi - hi-K|)
- We run 2 nested loops, the outer loop runs from 1 to n and the second loop runs from 1 to k.
- We initialize a pointer minn for every iteration.
- We check if the outer index variable is greater than or equal to inner index variable and if so, we update minn as minimum of its current value and height difference of the two indices.
- Update dp of current index as minn.
- Return dp of n-1 in the end.

**Code:**

```
import java.util.Scanner;
import java.util.ArrayList;

public class Main {
    public static int cost(ArrayList<Integer> arr, int n, int k) {
        int[] dp = new int[n];
        for (int ind = 1; ind < n; ind++) {
            int minn = Integer.MAX_VALUE;
            for (int i = 1; i ⩽ k; i++) {
                if (ind ⩾ i) {
                    minn = Math.min(minn, dp[ind - i] +
Math.abs(arr.get(ind) - arr.get(ind - i)));
                }
            }
            dp[ind] = minn;
        }
        return dp[n - 1];
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of days: ");
        int n = scanner.nextInt();
        System.out.print("Enter the number of maximum jumps allowed
at a time: ");
        int k = scanner.nextInt();

        ArrayList<Integer> arr = new ArrayList<>();
        System.out.print("Enter the elements of the array: ");
        for (int i = 0; i < n; i++) {
            arr.add(scanner.nextInt());
        }

        System.out.println("Minimum cost is: " + cost(arr, n, k));
    }
}
```

**OUTPUT:**

```
Enter the number of days :
5
Enter the number of maximum jumps allowed at a time:
3
Enter the elements of the array:
10 30 40 50 20
30

Process finished with exit code 0
```

**Upcoming lectures:**

- Problems on Dynamic Programming

PW SKILLS

# THANK YOU !