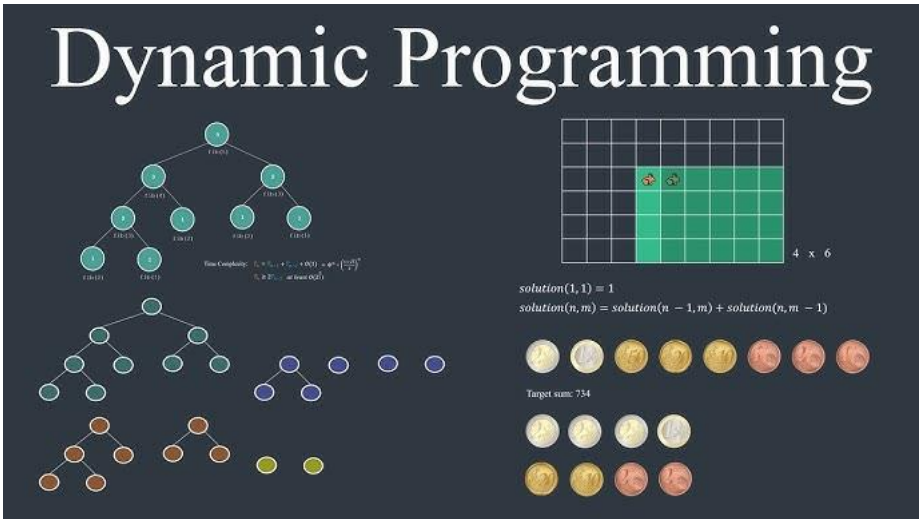
DYNAMIC PROGRAMMING

Dynamic Programming (DP) is defined as a technique that solves some particular type of problems in Polynomial Time. Dynamic Programming solutions are faster than the exponential brute method and can be easily proved their correctness.

Dynamic Programming 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.



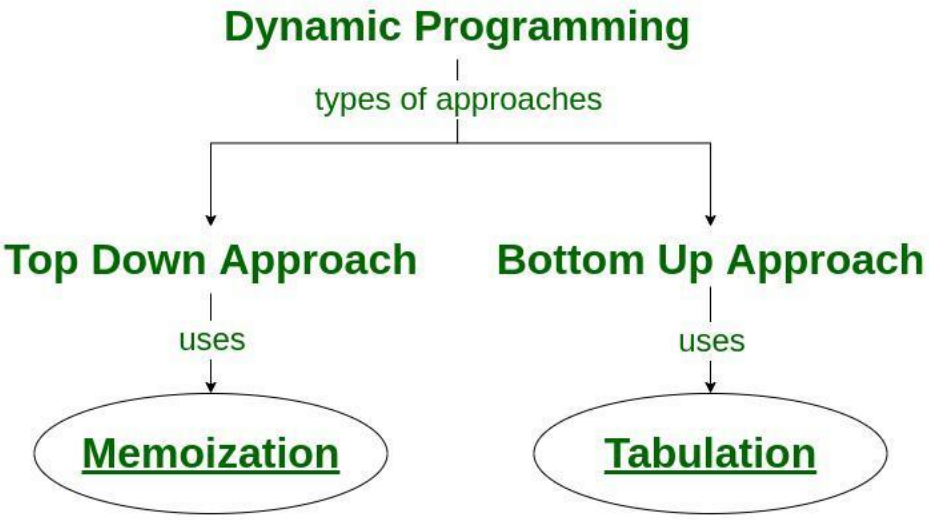
Characteristics of Dynamic Programming Algorithm:

→ Characterize structure of optimal solution, i.e. build a mathematical model of the solution.
→ Recursively define the value of the optimal solution.
→ Using bottom-up approach, compute the value of the optimal solution for each possible subproblems.
→ Construct optimal solution for the original problem using information computed in the previous step.

Techniques to solve Dynamic Programming Problems:

→ 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

→ Bottom-Up (Tabulation):
Analyze the problem and see in what order the subproblems are solved and work your way up from the trivial subproblem to the given problem. This process ensures that the subproblems are solved before the main problem.



How to solve a Dynamic Programming Problem?
To dynamically solve a problem, we need to check two necessary conditions:

→ Overlapping Subproblem When the solutions to the same subproblems are needed repetitively for solving the actual problem. The problem is said to have overlapping subproblems property.

→ Optimal Substructure Property: If the optimal solution of the given problem can be obtained by using optimal solutions of its subproblems then the problem is said to have Optimal Substructure Property.

Steps to solve a Dynamic programming problem:

1. Identify if it is a Dynamic programming problem.
2. Decide a state expression with the Least parameters.

3. Formulate state and transition relationships.
4. Do tabulation (or memorization).

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.

Approach:
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)
Adding 1 to all possible combinations of state (n = 6)
Eg: [(1+1+1+1+1+1) + 1]
[(1+1+1+3) + 1]
[(1+1+3+1) + 1]
[(1+3+1+1) + 1]
[(3+1+1+1) + 1]
[(3+3) + 1]
[(1+5) + 1]
[(5+1) + 1]

Code: Using Recursion

```
# Python program to Returns the number of arrangements to form 'n'
def solve(n):
  # Base case
    if(n < 0):
        return 0
    if(n == 0):
        return 1
    return solve(n-1)+solve(n-3)+solve(n-5)

  # This code is contributed by ishankhandelwals.
```

Code: Using Memoization

```
# Initialize to -1
dp = []

# This function returns the number of
# arrangements to form 'n'
def solve(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
    dp[n] = solve(n-1) + solve(n-3) + solve(n-5)
    return dp[n]

  # This code is contributed by ishankhandelwals.
```

Code: Using Iteration

```
def fibo(n):
  ans = [None] * (n + 1)

  # Storing the independent values in the
  # answer array
  ans[0] = 0
  ans[1] = 1

  # Using the bottom-up approach
  for i in range(2,n+1):
    ans[i] = ans[i - 1] + ans[i - 2]

  # Returning the final index
  return ans[n]

# Drivers code
n = 5

# Function Call
print(fibo(n))
```

Output: 8

Leet Code Problems

Problem: **Counting Bits**

Given an integer n, return an array ans of length n + 1 such that for each i (0 <= i <= n), ans[i] is the number of 1's in the binary representation of i.

Input: n = 5
Output: [0,1,1,2,1,2]

Code:

```
class Solution:
    def countBits(self, n: int) -> List[int]:
        ans = [0] *(n+1)
        for i in range(0,n+1):
            ans[i] = ans[i>>1] + (i & 1)
        return ans
```

Output: **Output:** [0,1,1,2,1,2]

Explanation:

n = 5.
ans = [0, 0, 0, 0, 0, 0] (initialization)
→ For i = 0: ans[0] = ans[0 >> 1] + (0 & 1) = 0 + 0 = 0 → ans = [0, 0, 0, 0, 0, 0]
→ For i = 1: ans[1] = ans[1 >> 1] + (1 & 1) = ans[0] + 1 = 0 + 1 = 1 → ans = [0, 1, 0, 0, 0, 0]
→ For i = 2: ans[2] = ans[2 >> 1] + (2 & 1) = ans[1] + 0 = 1 + 0 = 1 → ans = [0, 1, 1, 0, 0, 0]
→For i = 3: ans[3] = ans[3 >> 1] + (3 & 1) = ans[1] + 1 = 1 + 1 = 2 → ans = [0, 1, 1, 2, 0, 0]
→ For i = 4: ans[4] = ans[4 >> 1] + (4 & 1) = ans[2] + 0 = 1 + 0 = 1 → ans = [0, 1, 1, 2, 1, 0]
→For i = 5: ans[5] = ans[5 >> 1] + (5 & 1) = ans[2] + 1 = 1 + 1 = 2 → ans = [0, 1, 1, 2, 1, 2]
-→ **Time Complexity:** O(n)
-→ **Space Complexity:** O(n)

Problem: **Pascal's Triangle**

Given an integer numRows, return the first numRows of Pascal's triangle.
In Pascal's triangle, each number is the sum of the two numbers directly

Input: numRows = 5
Output: [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]

Code:

```
class Solution:
    def generate(self, numRows: int) -> List[List[int]]:
        ans = []
        for row in range(1, numRows+1):
            res= 1
            temp = [res]
            for col in range(1, row):
                res = res * (row-col)
                res = res // col
                temp.append(res)
            ans.append(temp)
        return ans
```

Output: [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]

Explanation:

→ **Row 1:**

- temp = [1]
- Add [1] to ans.

→**Row 2:**

- temp = [1, 1]
- The inner loop computes one value and adds [1, 1] to ans.

→ **Row 3:**

- temp = [1] initially.
- Compute the second value: res = 1 * (3-1) // 1 = 2.
- Add [1, 2, 1] to ans.

→ **Row 4:**

- temp = [1] initially.
- Compute the second value: res = 1 * (4-1) // 1 = 3.
- Compute the third value: res = 3 * (4-2) // 2 = 3.
- Add [1, 3, 3, 1] to ans.

→ **Row 5:**

- temp = [1] initially.
- Compute the second value: res = 1 * (5-1) // 1 = 4.
- Compute the third value: res = 4 * (5-2) // 2 = 6.
- Compute the fourth value: res = 6 * (5-3) // 3 = 4.
- Add [1, 4, 6, 4, 1]

-→ **Time Complexity:** $O(n^2)$
-→ **Space Complexity:** $O(n^2)$

Problem: **N-th Tribonacci Number**

The Tribonacci sequence Tn is defined as follows:
T0 = 0, T1 = 1, T2 = 1, and Tn+3 = Tn + Tn+1 + Tn+2 for n >= 0.
Given n, return the value of Tn.

Input: n = 4
Output: 4

Code:

```
class Solution:
    def tribonacci(self, n: int) -> int:
        def t(n):
            if n==0: return 0
            if n<=2: return 1
            return t(n-1)+t(n-2) + t(n-3)
        return t(n)
```

Output: 4

Explanation:

→ **n = 4**
→ **Call:** tribonacci(4)
  - Calls t(4)

→ **Call:** t(4)
  - Calls t(3), t(2), and t(1)

→ **Call:** t(3)
  - Calls t(2), t(1), and t(0)

→ **Call:** t(2)
  - Returns 1 (base case)

→ **Call:** t(1)
- Returns 1 (base case)

→ **Call:** t(0)
- Returns 0 (base case)

Thus, t(3) returns 1 + 1 + 0 = 2

→**Call:** t(2)
- Returns 1 (base case)

→ **Call:** t(1)
- Returns 1 (base case)

Thus, t(4) returns 2 + 1 + 1 = 4

-→ **Time Complexity:** $O(3^n)$
-→ **Space Complexity:** O(n)

Problem: **Minimum Cost to Make All Characters Equal**

You are given a 0-indexed binary string s of length n on which you can apply two types of operations:

Choose an index i and invert all characters from index 0 to index i (both inclusive), with a cost of i + 1
Choose an index i and invert all characters from index i to index n - 1 (both inclusive), with a cost of n - i
Return the minimum cost to make all characters of the string equal.

Invert a character means if its value is '0' it becomes '1' and vice-versa.

Input: s = "0011"
Output: 2

Code:

```
class Solution:
    def minimumCost(self, s: str) -> int:
        cost = 0
        for i in range(1, len(s)):
            if s[i-1] != s[i]:
                cost += min(i, len(s) - i)
        return cost
```

Output: 2

Explanation:

→ s = "0011"
→ Iteration 1 (i = 1):
- Comparing: s[0] = '0' and s[1] = '0'.
- Condition: Since s[0] == s[1], no action is taken. The cost remains 0.

→ Iteration 2 (i = 2):
- Comparing: s[1] = '0' and s[2] = '1'.
- Condition: Since s[1] != s[2], this indicates a change from '0' to '1'.
- Cost Calculation: The cost to make this transition consistent is the minimum of i and len(s) - i. Here, i = 2and len(s) - i = 2.
- Update Cost: cost += min(2, 2) = 2. So, cost is updated to 2.

→ Iteration 3 (i = 3):
- Comparing: s[2] = '1' and s[3] = '1'.
- Condition: Since s[2] == s[3], no action is taken. The cost remains 2.

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

Problem: **Construct the Longest New String**

You are given three integers x, y, and z.
You have x strings equal to "AA", y strings equal to "BB", and z strings equal to "AB". You want to choose some (possibly all or none) of these strings and concatenate them in some order to form a new string. This new string must not contain "AAA" or "BBB" as a substring.
Return *the maximum possible length of the new string*.
A **substring** is a contiguous **non-empty** sequence of characters within a string.

Input: x = 2, y = 5, z = 1
Output: 12

Code:
```
class Solution:
    def longestString(self, x: int, y: int, z: int) -> int:
        if x == y:
            return (4 * x + 2 * z)
```

```
        mini = min(x, y)
        return (2 * mini + 2 * (mini + 1) + 2 * z)
```

Output: 12

Explanation:

x = 2, y = 5, z = 1
→ Check if x == y:
- In this case, x = 2 and y = 5, so x != y.
- Therefore, the function moves to the else block.

→ Calculate mini:
- Here, mini = min(2, 5) = 2.

2 * mini + 2 * (mini + 1) + 2 * z
2 * 2 + 2 * (2 + 1) + 2 * 1 = 4 + 6 + 2 = 12

-→ **Time Complexity:** O(1)
-→ **Space Complexity:** O(1)

Problem: Minimum Operations to Make Binary Array Elements Equal to One II

You are given a binary array nums. You can do the following operation on the array any number of times (possibly zero):

Choose any index i from the array and flip all the elements from index i to the end of the array.
Flipping an element means changing its value from 0 to 1, and from 1 to 0.

Return the minimum number of operations required to make all elements in nums equal to 1

Input: nums = [0,1,1,0,1]
Output: 4

Code:

```
class Solution:
    def minOperations(self, nums: List[int]) -> int:
        flag = True
```

```
res = 0
for i in range(len(nums)):
    if (nums[i] != 0) != flag:
        flag = not flag
        res += 1
return res
```

Output: 4

Explanation:

nums = [0, 1, 1, 0, 1], flag = True, res = 0

→ **Iteration 1 (i=0)**: nums[0] = 0, flag = True
- nums[0] != 0 evaluates to False, so False != True is True.
- Since this is True, the condition (nums[i] != 0) != flag is satisfied, so we flip the flag to False and increment res by 1.
- res = 1

→ **Iteration 2 (i=1)**: nums[1] = 1, flag = False
- nums[1] != 0 evaluates to True, so True != False is True.
- The condition is again satisfied, so we flip the flag to True and increment res by 1.
- res = 2

→ **Iteration 3 (i=2)**: nums[2] = 1, flag = True
- nums[2] != 0 evaluates to True, so True != True is False.
- The condition is not satisfied, so nothing changes, and res remains 2.

→ **Iteration 4 (i=3)**: nums[3] = 0, flag = True
- nums[3] != 0 evaluates to False, so False != True is True.
- The condition is satisfied, so we flip the flag to False and increment res by 1.
- res = 3

→ **Iteration 5 (i=4)**: nums[4] = 1, flag = False
- nums[4] != 0 evaluates to True, so True != False is True.
- The condition is satisfied, so we flip the flag to True and increment res by 1.
- res = 4

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

Challenges faced during implementation of Dynamic Programming (DP):

→ Identifying Subproblems and Overlapping Subproblems

- **Challenge:** The core idea of DP is to break down a complex problem into simpler subproblems that overlap. Identifying these subproblems and ensuring they overlap can be difficult. For example, understanding that the solution to the larger problem can be built using the solutions to smaller subproblems is not always intuitive.
- **Example:** In the Fibonacci sequence problem, recognizing that Fib(n) can be broken down into Fib(n-1) and Fib(n-2) is key to using DP.

→ Choosing Between Top-Down (Memoization) and Bottom-Up (Tabulation) Approaches

- **Challenge:** Deciding whether to use a top-down approach with memoization (storing results of subproblems to avoid redundant calculations) or a bottom-up approach with tabulation (solving all subproblems and building up to the solution) can be tricky. Each approach has its own advantages and trade-offs in terms of readability, space complexity, and performance.
- **Example:** For problems like the knapsack problem, a bottom-up approach might be more intuitive, while for recursive problems like Fibonacci, a top-down approach might be easier to implement.

→ Handling State Representation

- **Challenge:** Correctly defining the state of a DP solution is crucial. The state typically represents the subproblem's solution, and choosing the right parameters to define the state can be difficult. Misrepresenting the state can lead to incorrect solutions or inefficient algorithms.
- **Example:** In a longest common subsequence problem, the state might be represented by the indices of the two strings being compared. Incorrectly defining the state (e.g., only using one index) could lead to incorrect results.

→ **Managing Time and Space Complexity**

- **Challenge:** While DP often reduces the time complexity of brute-force solutions, it can sometimes lead to high space complexity, especially when using memoization. Striking a balance between time and space efficiency is a common challenge.
- **Example:** In problems like matrix chain multiplication, where the DP table can be quite large, optimizing space usage while maintaining time efficiency can be difficult.

→ **Debugging and Verifying Correctness**

- **Challenge:** DP solutions can be complex and involve many states and transitions. Debugging these solutions to ensure they work for all edge cases and inputs is challenging. Ensuring that the recursive relations or transitions between states are correct is crucial.
- **Example:** In problems like the coin change problem, ensuring that all possible combinations are accounted for without redundancy or omission requires careful implementation and testing.

→ **Formulating the Recurrence Relation**

- **Challenge:** One of the most difficult parts of implementing DP is formulating the correct recurrence relation that expresses the solution of a problem in terms of its subproblems. This often requires deep insight into the problem's structure and can be non-trivial.
- **Example:** In problems like the edit distance problem, defining how the solution of a smaller subproblem relates to the larger problem (e.g., insertion, deletion, substitution) can be difficult, and a wrong relation can lead to incorrect results.