Dynamic Programming

→ Recursion + Storage : Approach 1 in DP
↓
Memoization

→ Method to solve complex problems when problems exhibit :

1) overlapping subproblems : So store

2) optimal substructure : optimal solution to a problem can be constructed from optimal solutions of its - sub-problem

Fibonacci Series : optimal substructure exist
Tower of Hanoi : No optimal substructure

2 Approaches :

(1) Memoization (2) Tabulation

Major Patterns :

1) Knapsack Based 2) Fibonacci based

3) Longest Common Subsequence (LCS)

4) Longest Increasing Subsequence (LIS)

5) Gap Strategy

6) Partition strategy

7) Kadane's Algorithm

* Approach to solve DP Problems

STEP 1 : Write the Recursion solution

STEP 2 : Memoization / Topdown (Recursive) + few lines of code

STEP 3 : Tabulation / Bottom up / Iterative

STEP 4 : Space Optimisation

* Where to use Dynamic Programming

(1) You are asked to find the optimal solution (longest, maximum, minimum etc.)

2) Problem involves choices → Recursion

especially  → 2 paths

↑ probability for overlapping subproblems.

QUESTIONS :

509. Fibonacci Number

Input = n = 4

Output = 3

Explanation : F(4) = F(3) + F(2) = 2+1 = 3

we know that recursion solution :

```
def fibonacci(n):
```

```
if n <= 1 :
    return n

else :
    return fibonacci(n-1) + fibonacci(n-2)
```

1) Memoization Approach :

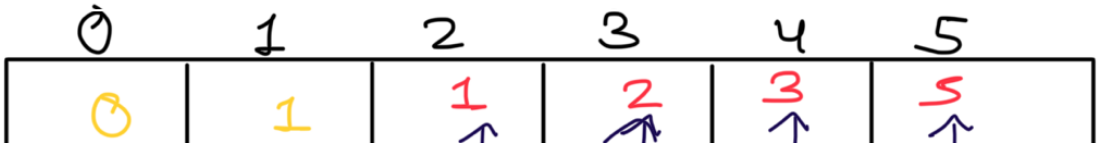
We store the function in hash table to reduce the time complexity [overlapping function are stored in hashed table]

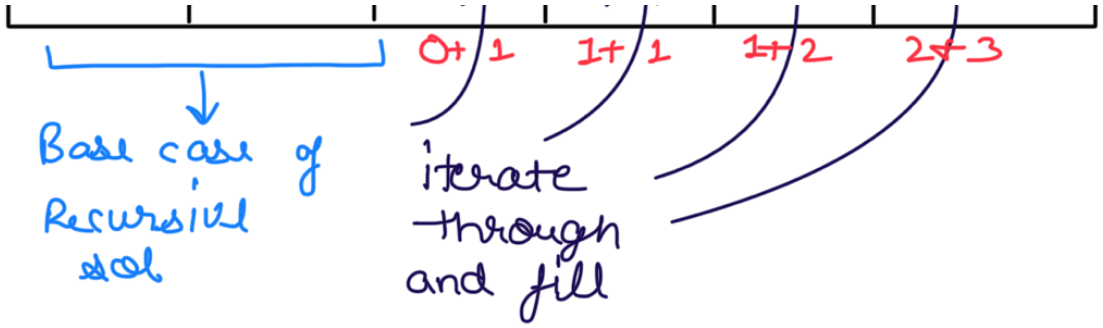
Memoization = Recursion + Storage.

```
def fib(n) :
    hash = {0:0, 1:1}

    if n in hash :
        return hash[n]

    else :
        hash[n] = fib(n-1) + fib(n-2)
        return hash[n]
```

Tabulation Approach :

Base case of recursive sol. = Initial condition of Tabulation





CODE :

```
def fib(n):
    dp = [0] * (n+1)
    if n > 0:
        dp[1] = 1
    count = 1
    while count < n:
        count += 1
        dp[count] = dp[count-1] + dp[count-2]
    return dp[n]
```

Space Optimised Tabulation Approach

CODE :

```
def fib(n):
    if n <= 1:
        return n
    prev = 0
    curr = 1
    counter = 1
```

```
while counter < n :
    next = prev + curr
    counter += 1

    prev = curr
    curr = next

return curr
```

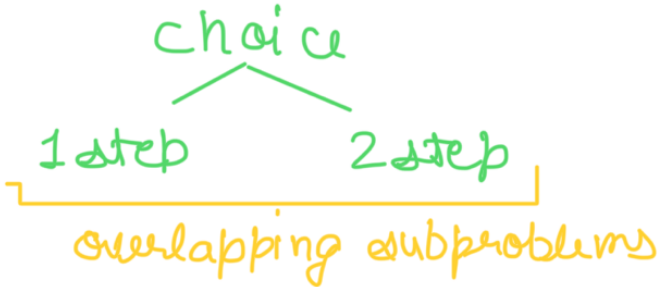
70. Climbing Stairs:

Input = 3 # 3 stair cases
Output = 3

Explanation :
1. 1 + 1 + 1 # steps
2. 2 + 1
3. 1 + 2

3 ways to climb the stairs.

Since there are choices included in the question [1 step and 2 step] it indicates to use D.P.



It is an application of fibonacci series where $f(n) = f(n-1) + f(n-2)$

Same solution as fibonacci series

746. Min Cost Climbing Stairs

Input : cost = [10, 15, 20]

You can either start from index 0 or 1

Output = 15

We start from index 0 and climb two steps to reach the top.

Note : You can either climb 1 or 2 step at a time

Min Cost → Optimal solution → Hint of Using D.P.

Recursive Approach

```
def minCost(cost):
    n = len(cost)
    def costFrom(index):
        if index > n-1 : return 0
        onestep = cost[index] + costFrom(index+1)
        twostep = cost[index] + costFrom(index+2)
```

```
        return min(onestep, twostep )
    return min(costFrom(0) , costFrom(1) )
```

Memoization Approach :

```
def minCost( cost):
    n = len(cost)
    mincost = [-1]*n
    def costFrom(index):
        if index > n-1 : return 0
        if mincost[index] != -1 :
            return mincost[index]
        onestep = cost[index] + costFrom(index+1)
        twostep = cost[index] + costFrom(index+2)
        mincost[index] = min(onestep, twostep )
        return mincost[index]
    return min(costFrom(0) , costFrom(1))
```

TABULATION APPROACH :

0 1 2 3 4 5 6 7

[1, 2, 1, 3, 4, 2, 1, 1] → cost

0 1 2 3 4 5 6 7 destination

n + 1 array

mincost[i] → min[mincost[i-1] + cost[i-1]
mincost[i-2] + cost[i-2]

```
def minCost(cost):
    n = len(cost)
    mincost = [0] * (n+1)
    for i in range(2, n+1):
        onestep = cost[i-1] + mincost[i-1]
        twostep = cost[i-2] + mincost[i-2]
        mincost[i] = min(onestep, twostep)
    return mincost[n]
```

Knapsack Type Questions

0-1 Knapsack

You are given weights and values of **N** items, put these items in a knapsack of capacity **W** to get the maximum total value in the knapsack. Note that we have only **one quantity of each item**.
In other words, given two integer arrays **val[0..N-1]** and **wt[0..N-1]** which represent values and weights associated with **N** items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of **val[]** such that sum of the weights of this subset is smaller than or equal to **W.** You cannot break an item, **either pick the complete item or dont pick it (0-1 property)**.

Example : $N=3$, $W=8$

val → [2, 3, 9]

wt → [8, 2, 5]

Output = 3+9 = 12

- Identify DP —— choices → Recursion

  optimal solution → Maximum value → DP.

Recursion Approach :

```
def knapsack(W, wt, val, n):
    def helper(index, rem_weight):
        if index > n-1 or rem_weight == 0:
            return 0

        exclude = helper(index+1, rem_weight)
        include = 0
        if wt[index] <= rem_weight:
            include = val[index] +
                      helper(index+1, rem_weight -
                             wt[index])
        return max(include, exclude)

    return helper(0, W)
```

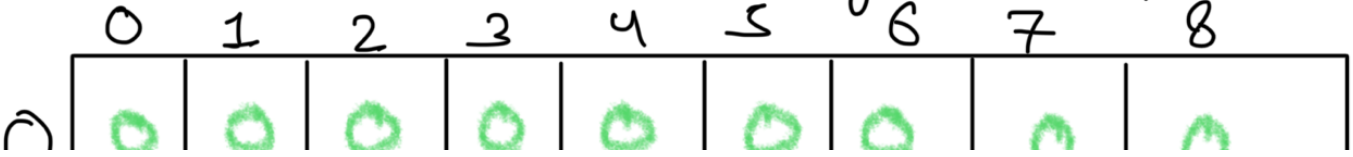
Memoization Approach

In this we use a 2D array as the helper function (#Recursive function) has two inputs

```
def knapsack(W, wt, val, n):

    dp = [[-1]*(W+1) for _ in range(n)]

    def helper(index, rem_weight):

        if index > n-1 or rem_weight == 0:
            return 0

        if dp[index][rem_weight] != -1
            return dp[index][rem_weight]

        exclude = helper(index+1, rem_weight)
        include = 0
        if wt[index] <= rem_weight:
            include = val[index] +
                      helper(index+1, rem_weight -
                             wt[index])
        dp[index][rem_weight] = max(include, exclude)
        return dp[index][rem_weight]

    return helper(0,W)
```

TABULATION APPROACH

We will use D.P. table for this approach:



I_1	0	0	0	0	0	0	0	2
I_2	0	0	3	3	3	3	3	3
I_3	0	0	3	3	3	9	9	12	12

Initial condition is base condition for recursive approach

dp[i][j]

exclude = dp[i-1][j]

include = val[i-1] + dp[i-1][j-wt[i-1]]

CODE:

```
def KnapSack (W, wt, val, n):
    dp = [ [0] x(w+1) for _ in range (n+1) ]
    for i in range (1, n+1):
        for j in range (1, w+1):
            exclude = dp[i-1][j]
            include = 0
            if wt[i-1] <= j :
                include = val [i-1] +
                          dp [i-1] [j - wt [i-1]]
```

```
            dp[i][j] = max(include, exclude)
    return  dp[n][w]
```

✱ Space Optimised Tabulation Approach.

Observation based on Tabulation Approach:

(i) We only need previous row values to calculate current row value.

```
def Knapsack(W, wt, val, n):
    prev = [0]*(W+1)
    curr = [0]*(W+1)

    for i in range(1, n+1):
        for j in range(1, W+1):

            exclude = prev[j]
            include = 0
            if wt[i-1] <= j:
                include = val[i-1] + prev[j-wt[i-1]]

            curr[j] = max(include, exclude)

        prev = curr[:]

    return curr[W]
```

✱ Unbounded Knapsack

Given a set of **N** items, each with a weight and a value, represented by the array **w** and **val** respectively. Also, a knapsack with weight limit **W**.
The task is to fill the knapsack in such a way that we can get the maximum profit. Return the maximum profit.
Note: Each item can be taken any number of times.

val = [5, 10] wt = [2, 2]

N = 2 W = 6

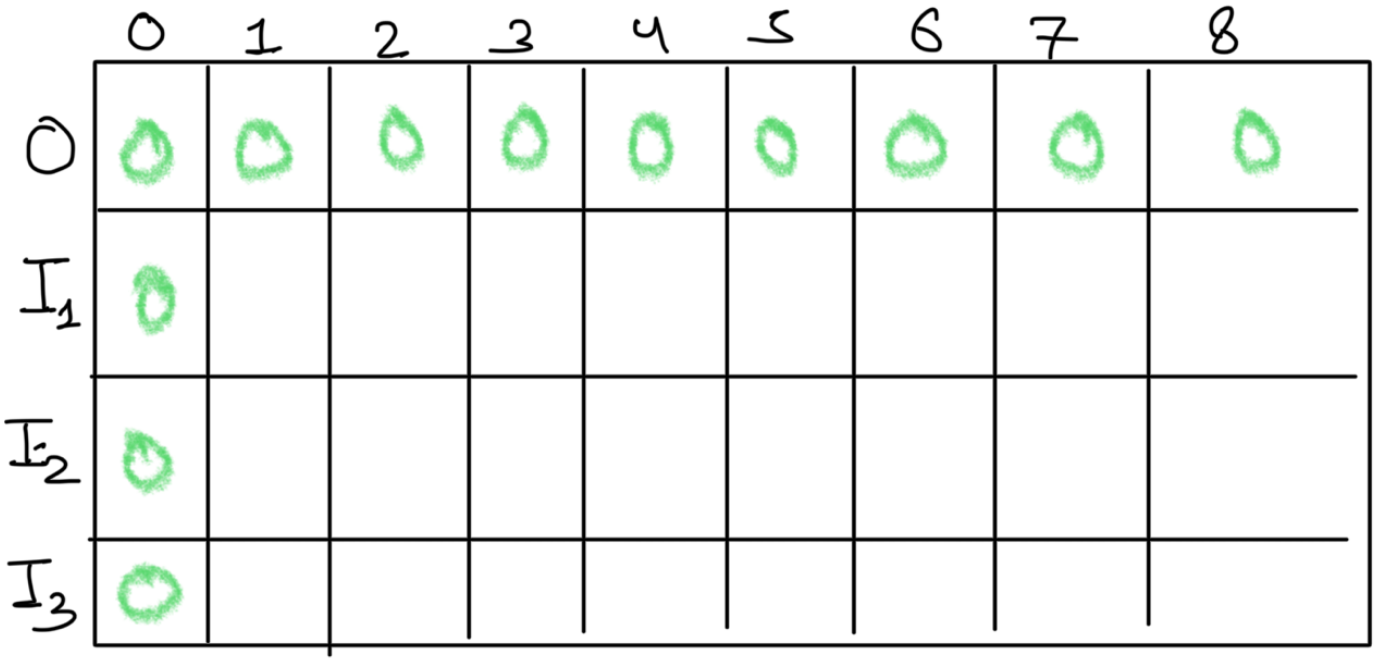
OUTPUT : 30

Tabulation Approach :

val = [2, 4, 9] N = 3

wt = [8, 2, 5]

W = 8



Similar to 0-1 knapsack problem

changes: in the include part instead of dp[i-1] we replace it with dp[i] as repetition is allowed

494. Target Sum

nums = [1, 1, 1, 1, 1], target = 3

Output = 5

5 ways to assign symbols to nums array so that it can reach to target

	0	1	2	3	4	5	6	7	8	9	10
	-5	-4	-3	-2	1	0	1	2	3	4	5
1											
1											
1											
1											
1											

Since indicies can't be negative, we did

[sum_nums + summation] = indicies

Let sum_nums = -4

then according to formula:

$-4 + 5 = \boxed{1}$ indices

- The sum ranges from -sum to +sum, so we build the dp table as

[[None]*(2*summation+1)]

```python
class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        n = len(nums)
        summation = sum(nums)
        dp = [[None]*(2*summation+1) for _ in range(n)]

        def helper(index, sum_nums):
            if index < 0:
                if sum_nums == target:
                    return 1
                else:
                    return 0

            if dp[index][sum_nums + summation] != None:
                return dp[index][sum_nums + summation]
```

Question: Partition Equal Subset Sum

Input : [1, 5, 11, 15]
Output : True

Explanation : [1,5,5] and [11]

Input : [1, 2, 3, 5]
Output : False.

Memoization Approach :

CODE :

```
def canPartition(nums):
    total_sum = sum(nums):
    if total_sum % 2 != 0
        return False

    target = total_sum // 2

    memo = {}

    def helper(index, current_sum):
        #Base Cases

        if current_sum == target:
            return True

        if index == n or current_sum > target:
            return False

        if (index, current_sum) in memo:

            return memo[(index, current_sum)]

        # including the current number

        if helper(index + 1, current_sum + nums[index]):

            memo[index, current_sum] = True
            return True

        # Exclude the current number

        if helper[index + 1, current_sum]:

            memo[index, current_sum)] = True
            return True
```

```
        memo[(index, current_sum)] = False
        return False

    return helper(0, 0)
```

TABULATION APPROACH

CODE:

```
def canPartition(nums):
    n = len(nums):
    total_sum = sum(nums)
    if total_sum % 2 != 0
        return False.
    target = total_sum // 2
    # intialising the DP table.
    dp = [[False] * (target+1) for _
                in range(n+1)]

    # base case; it is always possible to make
      sum of 0
    for i in range(n+1):
        dp[i][0] = True
```

```
# Filling the dp table
for i in range(1, n+1)
    for j in range(1, target+1):
        if j < nums[i-1]
            # if the current number is
              greater than target, sum(j),
              we can't include it.
            dp[i][j] = dp[i-1] j
        else:
            # two choice include or exclude
            dp[i][j] = dp[i-1][j] or
                       dp[i-1][j-nums[i-1]]
return dp[n][target]
```