What is Dynamic Programming?

Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems

Example : 1

$$1 + 1 + 1 + 1 + 1 + 1 + 1 = 7$$

$$\underbrace{1 + 1 + 1 + 1 + 1 + 1 + 1}_{7} + 2 = 9$$

What is Dynamic Programming?

Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems

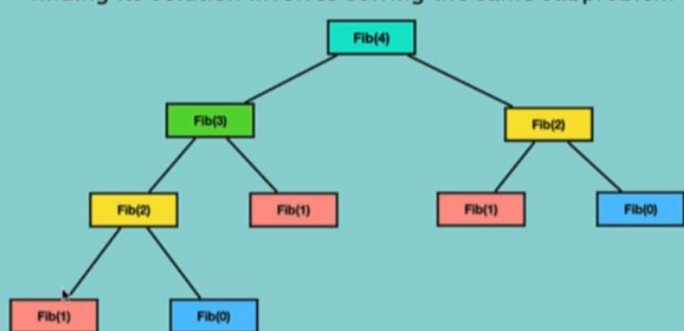
Optimal Substructure:

If any problem's overall optimal solution can be constructed from the optimal solutions of its subproblem then this problem has optimal substructure

Example: Fib(n) = Fib(n-1) + Fib(n-2)

Overlapping Subproblem:

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



AppMillers
www.appmillers.com

Top Down with Memoization

Solve the bigger problem by recursively finding the solution to smaller subproblems. 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. This technique of storing the results of already solved subproblems is called **Memoization.**

Example : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)

```
Fibonacci(N):
   If n < 1 return error message
   If n = 1 return 0
   If n = 2 return 1
   Else
       return Fibonacci(N-1) + Fibonacci(N-2)
```

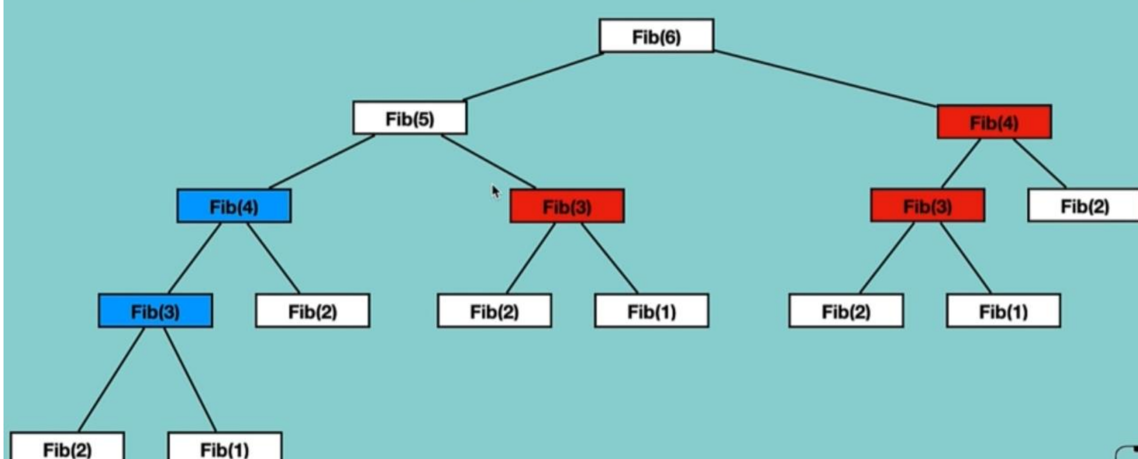
Time complexity : $O(c^n)$
Space complexity : O(n)

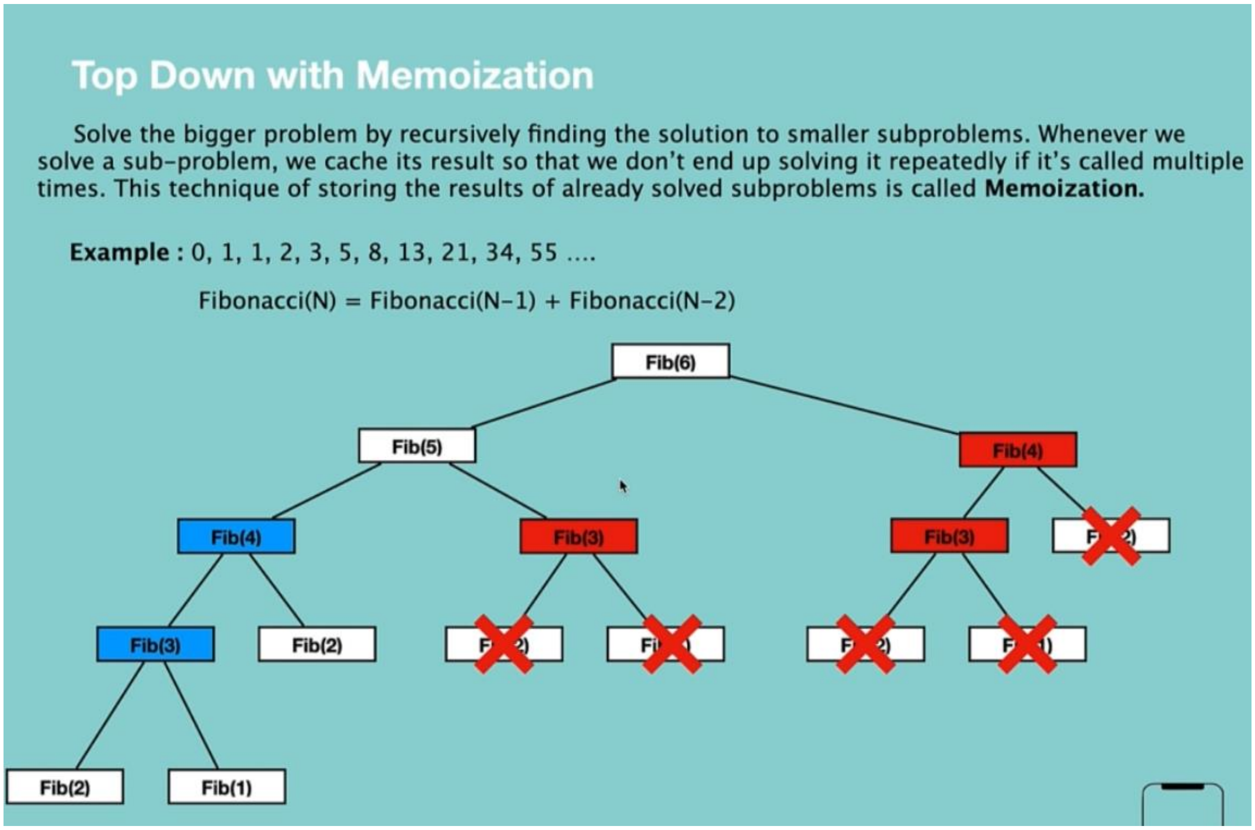
Top Down with Memoization

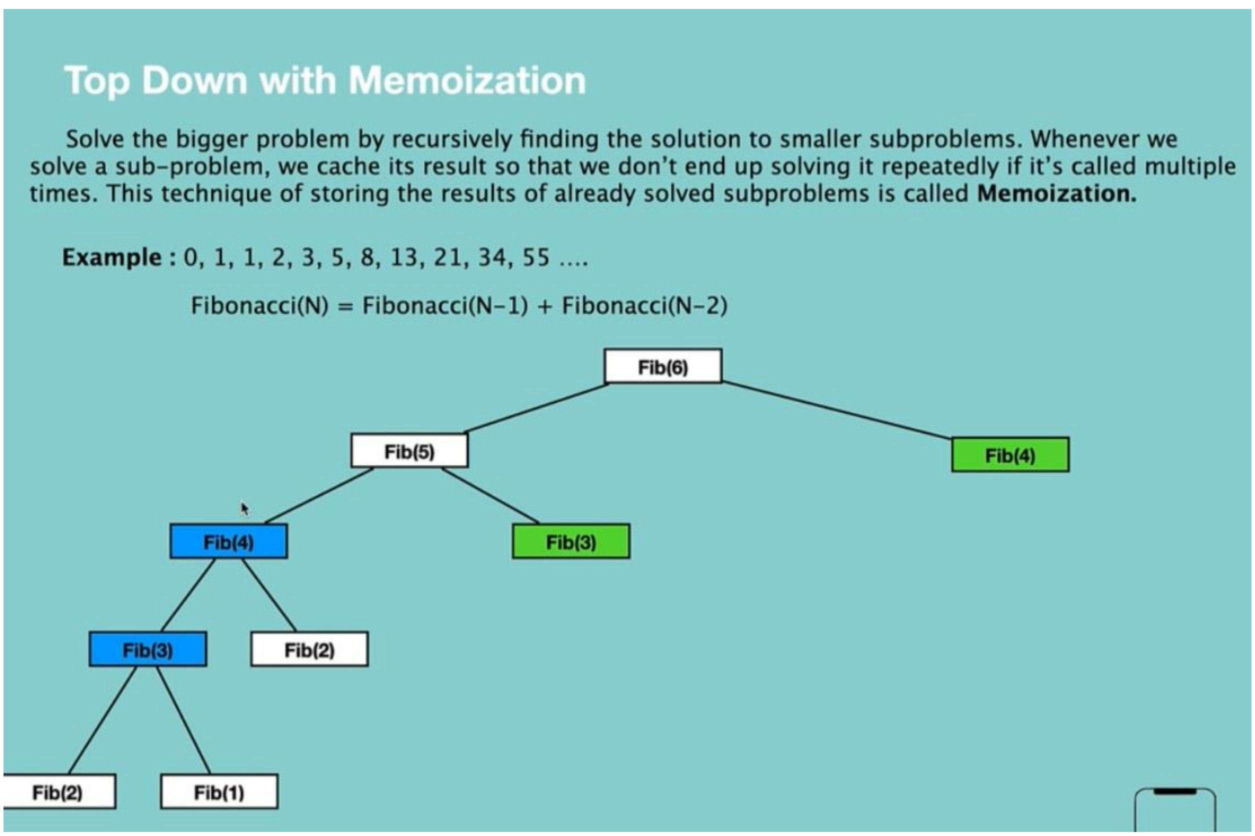
Solve the bigger problem by recursively finding the solution to smaller subproblems. Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. This technique of storing the results of already solved subproblems is called **Memoization.**

Example : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)







The tree has now reduced drastically.

Top Down with Memoization

Solve the bigger problem by recursively finding the solution to smaller subproblems. 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. This technique of storing the results of already solved subproblems is called **Memoization.**

Example : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)

F1	F2	F3	F4	F5	F6
0	1				F4 + F5

F1	F2	F3	F4	F5	F6
0	1			F3 + F4	F4 + F5

F1	F2	F3	F4	F5	F6
0	1		F2 + F3	F3 + F4	F4 + F5

F1	F2	F3	F4	F5	F6
0	1	F1 + F2	F2 + F3	F3 + F4	F4 + F5

F1	F2	F3	F4	F5	F6
0	1	1	F2 + F3	F3 + F4	F4 + F5

F1	F2	F3	F4	F5	F6
0	1	1	2	F3 + F4	F4 + F5

F1	F2	F3	F4	F5	F6
0	1	1	2	3	F4 + F5

F1	F2	F3	F4	F5	F6
0	1	1	2	3	5

It is called as Top-Down approach as we split the problem into smaller sub problems from the top and we go until down starting solving the problem then go up. We can increase the efficiency of divide and conquer logic with the help of Top-Down Approach.

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 subproblems first). This is done by filling up a table. Based on the results in the table, the solution to the top/original problem is then computed.

Example : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)

F1	F2	F3	F4	F5	F6
0	1				F4 + F5

F1	F2	F3	F4	F5	F6
0	1			F3 + F4	F4 + F5

F1	F2	F3	F4	F5	F6
0	1		F2 + F3	F3 + F4	F4 + F5

F1	F2	F3	F4	F5	F6
0	1	F1 + F2	F2 + F3	F3 + F4	F4 + F5

F1	F2	F3	F4	F5	F6
0	1	1	F2 + F3	F3 + F4	F4 + F5

F1	F2	F3	F4	F5	F6
0	1	1	2	F3 + F4	F4 + F5

F1	F2	F3	F4	F5	F6
0	1	1	2	3	F4 + F5

F1	F2	F3	F4	F5	F6
0	1	1	2	3	5

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 subproblems first). This is done by filling up a table. Based on the results in the table, the solution to the top/original problem is then computed.

Example : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)

F1	F2	F3	F4	F5	F6
0	1	F1 + F2			

F1	F2	F3	F4	F5	F6
0	1	1			

F1	F2	F3	F4	F5	F6
0	1	1	F2 + F3		

F1	F2	F3	F4	F5	F6
0	1	1	2		

F1	F2	F3	F4	F5	F6
0	1	1	2	F3 + F4	

F1	F2	F3	F4	F5	F6
0	1	1	2	3	

F1	F2	F3	F4	F5	F6
0	1	1	2	3	F4 + F5

F1	F2	F3	F4	F5	F6
0	1	1	2	3	5

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 subproblems first). This is done by filling up a table. Based on the results in the table, the solution to the top/original problem is then computed.

Example : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

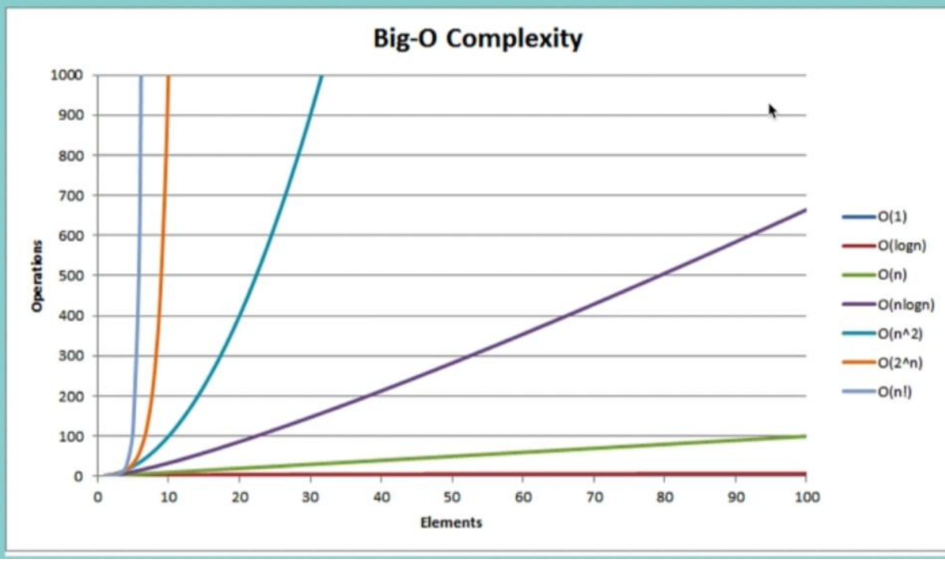
Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)

```
def fibonacciTab(n):
  tb = [0, 1]
  for i in range(2, n + 1):
      tb.append(tb[i - 1] + tb[i - 2])
  return tb[n-1]
```

Time complexity : O(n)
Space complexity : O(n)

Top Down vs Bottom Up

Problem	Divide and Conquer	Top Down	Bottom Up
Fibonacci numbers	$O(c^n)$	O(n)	O(n)



AppMillers

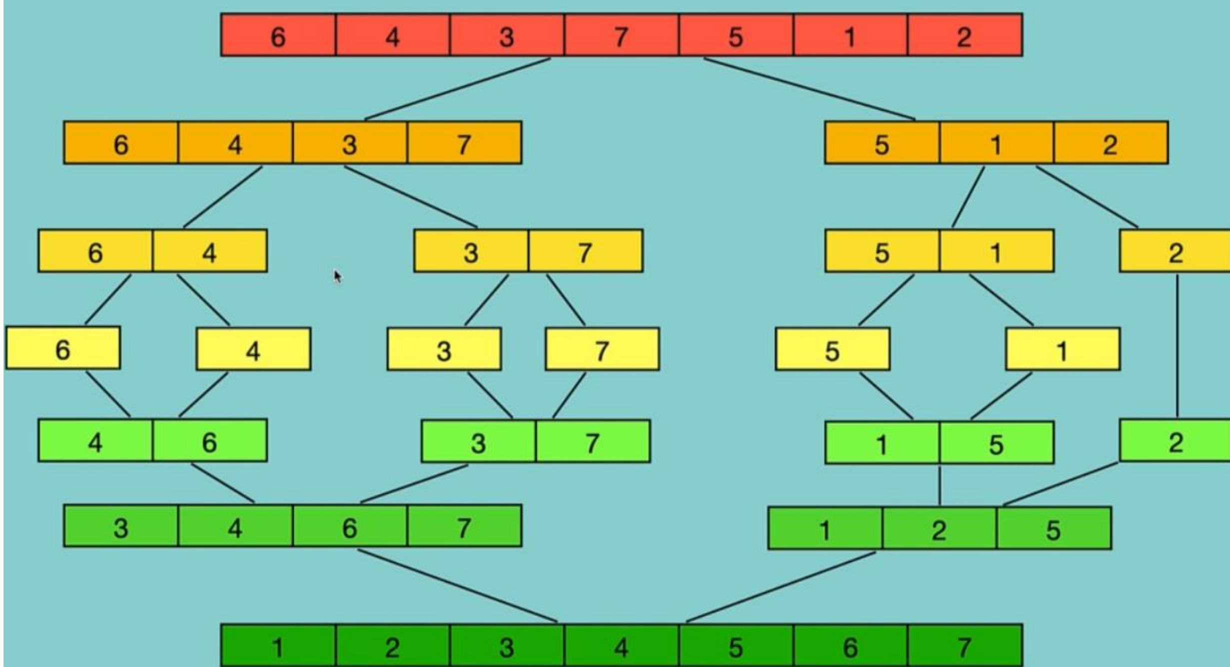
Top Down vs Bottom Up

	Top Down	Bottom Up
Easyness	Easy to come up with solution as it is extension of divide and conquer	Difficult to come up with solution
Runtime	Slow	Fast
Space efficiency	Unnecessary use of stack space	Stack is not used
When to use	Need a quick solution	Need an efficient solution

It is very hard to find the logic for Bottom-Up Approach but it is very efficient.

Is Merge Sort Dynamic Programming?

1.Does it have Optimal Substructure property?
2.Does it have Overlapping Subproblems property?



Dynamic Programming - Number Factor

Problem Statement:
Given N, find the number of ways to express N as a sum of 1, 3 and 4.

Example 1

- N = 4
- Number of ways = 4
- Explanation : There are 4 ways we can express N. {4},{1,3},{3,1},{1,1,1,1}

Example 2

- N = 5
- Number of ways = 6
- Explanation : There are 6 ways we can express N. {4,1},{1,4},{1,3,1},{3,1,1},{1,1,3},{1,1,1,1,1}

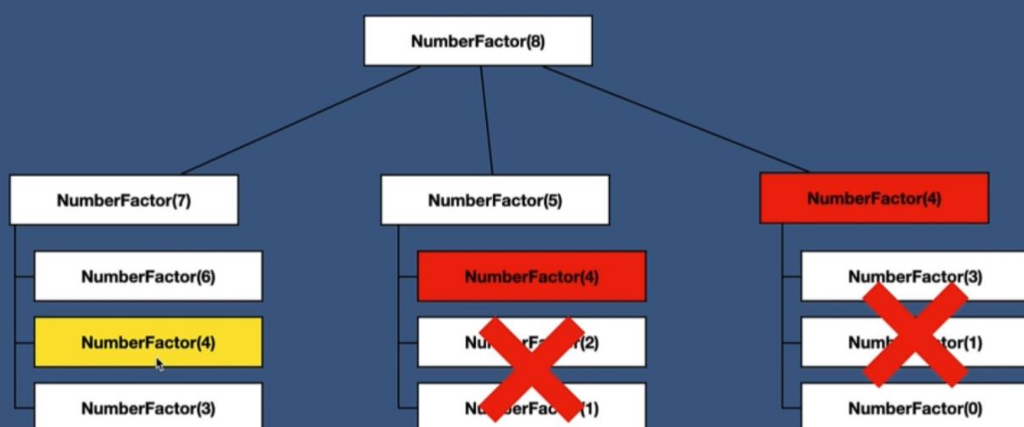
Dynamic Programming - Number Factor

Problem Statement:
Given N, find the number of ways to express N as a sum of 1, 3 and 4.

```
NumberFactor(N)
   If N in (0,1,2) return 1
   If N = 3 return 2
  Else
      return NumberFactor(N-1) + NumberFactor(N-3) + NumberFactor(N-4)
```

Dynamic Programming - Number Factor

Problem Statement:
Given N, find the number of ways to express N as a sum of 1, 3 and 4.



Dynamic Programming - Number Factor

Problem Statement:
Given N, find the number of ways to express N as a sum of 1, 3 and 4.

```
NumberFactor(N):

   If N in (0,1,2) return 1
   If N = 3 return 2

   Else
      rec1 = NumberFactor(N-1)
      rec2 = NumberFactor(N-3)
      rec3 = NumberFactor(N-4)

      return rec1 + rec2 + rec3
```

Dynamic Programming - Number Factor

Problem Statement:
Given N, find the number of ways to express N as a sum of 1, 3 and 4.

```
NumberFactor(N, dp):  ------------------------------>  Step 1

   If N in (0,1,2) return 1
   If N = 3 return 2
   Elif N in dp return dp[N]  --------------------->  Step 2
   Else
      rec1 = NumberFactor(N-1)
      rec2 = NumberFactor(N-3)
      rec3 = NumberFactor(N-4)
      dp[N] = rec1 + rec2 + rec3  ----------------->  Step 3
      return dp[N]  ------------------------------->  Step 4
```

The above are the four steps that are required to convert the program to dynamic programming.

Dynamic Programming - Number Factor

Problem Statement:
Given N, find the number of ways to express N as a sum of 1, 3 and 4.

Top Down Approach

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2				NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2			NF5+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2		NF4+NF2+NF1	NF5+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	NF3+NF2+NF0	NF4+NF2+NF1	NF5+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	2+1+1=4	NF4+NF2+NF1	NF5+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	2+1+1=4	4+1+1=6	NF5+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	2+1+1=4	4+1+1=6	6+2+1=9	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	2+1+1=4	4+1+1=6	6+2+1=9	9+4+2=15

Dynamic Programming - Number Factor

Problem Statement:
Given N, find the number of ways to express N as a sum of 1, 3 and 4.

Bottom Up Approach

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	NF3+NF1+NF0			

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	NF3+NF1+NF0	NF4+NF2+NF1		

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	NF3+NF1+NF0	NF4+NF2+NF1	NF6+NF3+NF2	

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	NF3+NF1+NF0	NF4+NF2+NF1	NF6+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	2+1+1=4	NF4+NF2+NF1	NF6+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	2+1+1=4	4+1+1=6	NF6+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	2+1+1=4	4+1+1=6	6+2+1=9	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	2+1+1=4	4+1+1=6	6+2+1=9	9+4+2=15

Dynamic Programming - Number Factor

Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.

Bottom Up Approach

```
numberFactor(n)
  tb = {1,1,1,2}
  for i in range(4, n+1):
    tb.append(tb[i-1]+tb[i-3]+tb[i-4])
  return tb[n]
```

House Robber

Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

Example 1

Answer

- Maximum amount = 41
- Houses that are stolen : 7, 30, 4

Option1 = 6 + f(5) → **Max(Option1, Option2)**

Option2 = 0 + f(6)

House Robber

Problem Statement:

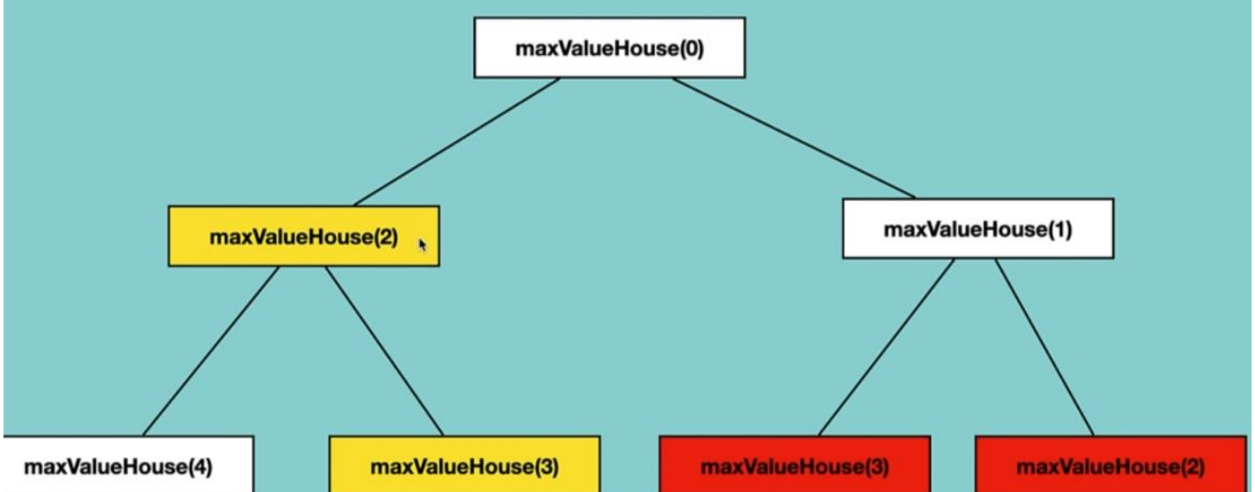
- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

```
maxValueHouse(houses, currentHouse):
   If currentHouse > length of houses
       return 0
  Else
      stealFirstHouse = currentHouse + maxValueHouse(houses, currentHouse+2)
      skipFirstHouse = maxValueHouse(houses, currentHouse+1)
      return max(stealFirstHouse, skipFirstHouse)
```

House Robber

Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen



House Robber

Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

```
maxValueHouse(houses, currentHouse, tempDict): ——————> Step 1
   If currentHouse > length of houses
       return 0
  Else
      If currentHouse not in tempDict: ——————> Step 2
          stealFirstHouse = currentHouse + maxValueHouse(houses, currentHouse+2)
          skipFirstHouse = maxValueHouse(houses, currentHouse+1)
          tempDict[currentHouse] = max(stealFirstHouse, skipFirstHouse)——> Step 3
      return tempDict[currentHouse] ——————> Step 4
```

House Robber

Top Down Approach

Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

H0	H1	H2	H3	H4	H5	H5
6	7	1	30	8	2	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)						

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)					

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)				

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)			

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)		

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(H5+HR7, HR6)	

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(H5+HR7, HR6)	max(H6+HR8, HR7)

House Robber

Top Down Approach

Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

H0	H1	H2	H3	H4	H5	H5
6	7	1	30	8	2	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(H5+HR7, HR6)	max(H6+HR8, HR7)

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(H5+HR7, HR6)	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(2+0, 4)=4	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(8+4,4)=12	4	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	12	4	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	34	12	4	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
41	41	34	34	12	4	4

House Robber

Bottom Up Approach

Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

H0	H1	H2	H3	H4	H5	H5
6	7	1	30	8	2	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(H5+HR7, HR6)	max(H6+HR8, HR7)	0	0

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(H5+HR7, HR6)	4	0	0

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	4	4	0	0

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	12	4	4	0	0

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	34	12	4	4	0	0

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	34	34	12	4	4	0	0

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
max(H0+HR2, HR1)	41	34	34	12	4	4	0	0

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
41	41	34	34	12	4	4	0	0

House Robber

Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

Bottom Up Approach

```
def houseRobberBU(houses, currentIndex):
    tempAr = [0]*(len(houses)+2)
    for i in range(len(houses)-1, -1, -1):
        tempAr[i] = max(houses[i]+tempAr[i+2], tempAr[i+1])
    return tempAr[0]
```

Convert String

Problem Statement:

- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations

Example 1

- S1 = “catch”
- S2 = “carch”
- Output = 1
- Explanation : Replace “r” with “t”

Example 2

- S1 = “table”
- S2 = “tbres”
- Output = 3
- Explanation : Insert “a” to second position, replace “r” with “l” and delete “s”

Convert String

Problem Statement:

- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations

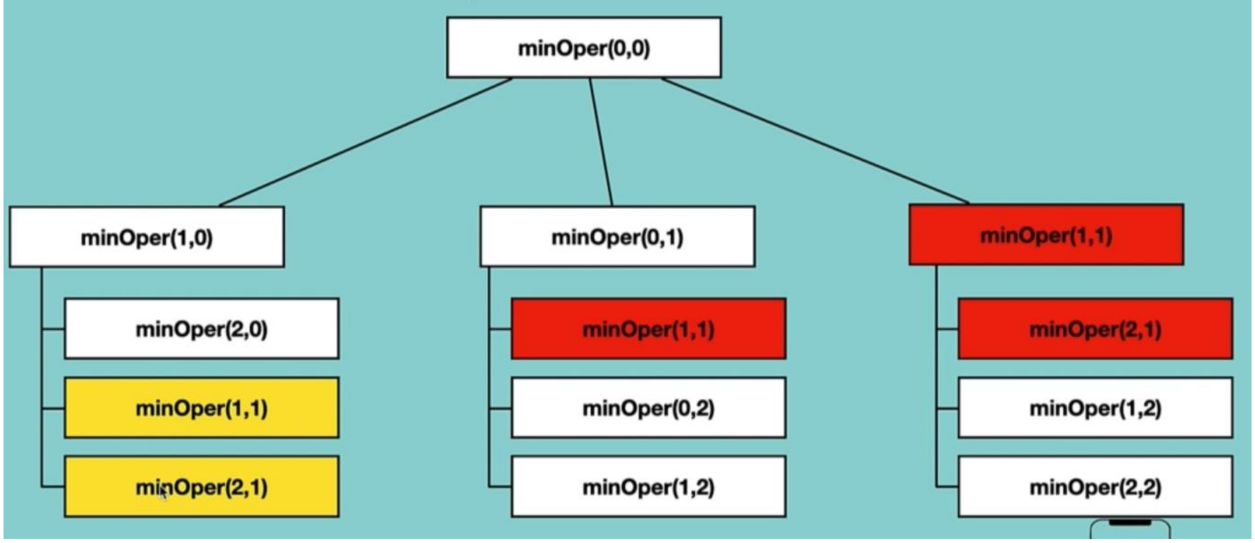
```
findMinOperation(s1, s2, index1, index2):
   If index1 == len(s1)
       return len(s2)-index2
   If index2 == len(s2)
       return len(s1)-index1
   If s1[index1] == s2[index2]
       return findMinOperation(s1, s2, index1+1, index2+1)

  Else
      deleteOp = 1 + findMinOperation(s1, s2, index1, index2+1)
      insertOp = 1 + findMinOperation(s1, s2, index1+1, index2)
      replaceOp = 1 + findMinOperation(s1, s2, index1+1, index2+1)
     return min(deleteOp, insertOp, replaceOp)
```

Convert String

Problem Statement:

- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations



Convert String

Problem Statement:

- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations

```
findMinOperation(s1, s2, index1, index2, tempDict):          → Step 1
   If index1 == len(s1)
       return len(s2)-index2
   If index2 == len(s2)
       return len(s1)-index1
   If s1[index1] == s2[index2]
       return findMinOperation(s1, s2, index1+1, index2+1)

  Else
       dictKey = str(index1)+str(index2)
       if dictKey not in tempDict:                            → Step 2
           deleteOp = 1 + findMinOperation(s1, s2, index1, index2+1)
           insertOp = 1 + findMinOperation(s1, s2, index1+1, index2)
           replaceOp = 1 + findMinOperation(s1, s2, index1+1, index2+1)
           tempDict[dictKey] = min (deleteOp, insertOp, replaceOp) → Step 3
       return tempDict[dictKey]                               → Step 4
```