10/01/2024

DYNAMIC PROGRAMMING
CLASS - 3

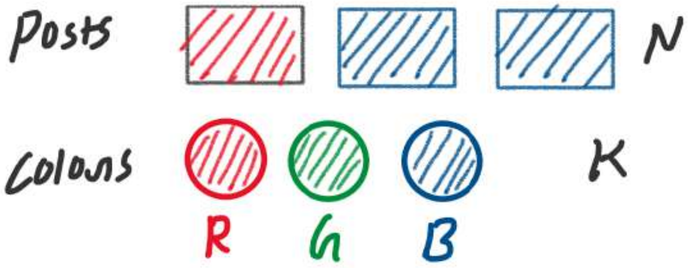
📂 1. Painting Fence Algorithm (GFG)

Problem Statement:

The painting fence algorithm determines the number of ways to paint a fence with multiple 'N' posts and 'K' colours. The algorithm ensures that at most 2 adjacent posts (no more than two adjacent posts) have the same colour. Since answer can be large return it modulo 10^9 + 7 (1000000007).

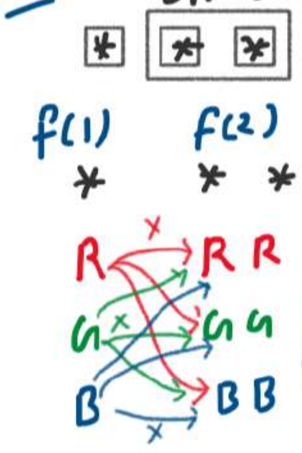
Examples:

```
Input : N = 1 K = 3      Input : N = 2 K = 3      Input : N = 3 K = 3      Input : N = 4 K = 3
Output : 3               Output : 9               Output : 24              Output : 66
```



	N=1	N=2	N=3	N=4
SAME	□ R G B K=3	□ □ R R G G B B K=3	□ □ □ R G G R B B G R R G B B B R R B G G K * (K-1) = 3×2 f(N-2) * K-1	⇒ f(2)(K-1) ⇒ 9 * 2 ⇒ 18 ⇒ $f(3)_{Diff}$
Diff.		□ □ R G R B G R G B B R B G K×(K-1)=3×2	□ □ □ R R G R R B G G R G G B B B R B B G SAME + R G R R G B R B R R B G G R G G R B G B G G B R B R B B R G B G B B G R Diff	⇒ $\overset{f(3)}{same}(K-1) + \overset{f(3)}{Diff}(K-1)$ ⇒ (6×2) + (18×2) ⇒ 12 + 36 ⇒ 48 ways Total ⇒ 48 + 18 = 66

N=3 SAME K=3



Inshort

$f(3)_{same} = f(2)_{Diff}$

$f(3)_{SAME} = f(1) * (K-1)$

$= f(n-2) * (K-1)$

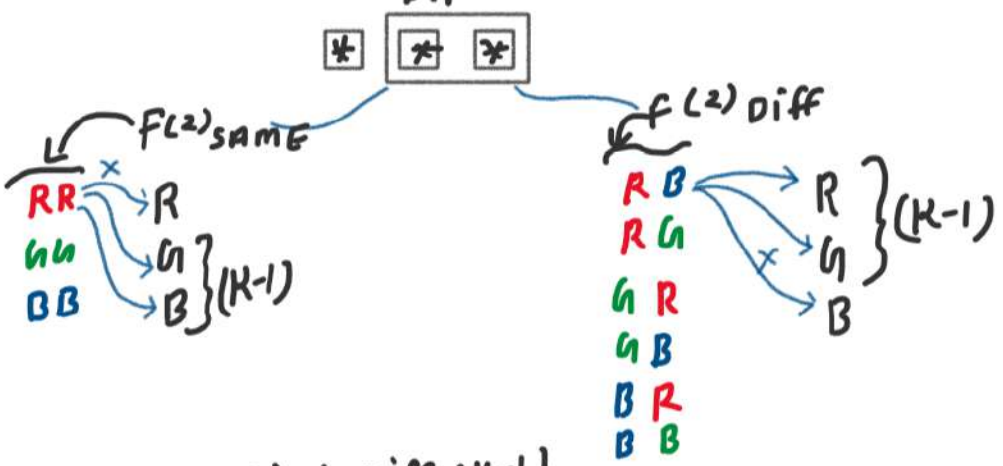
$f(3)^{same}_{way} \Rightarrow 3 * 2$

$\Rightarrow 6$ ways.

+

N=3 K=3

Diff



$f(3)_{Diff} = same(K-1) + Diff(K-1)$

$= (K-1)[same + Diff]$

$= (K-1)[f(2)]$

$f(3)^{Diff}_{ways} \Rightarrow 2[9]$

$\Rightarrow 18$ ways

$$= \text{Total ways } f(3)$$

$$= f(3)_{\text{same}} + f(3)_{\text{Diff}}$$

$$= [f(n-2) * (K-1)] + [f(n-1) * (K-1)]$$

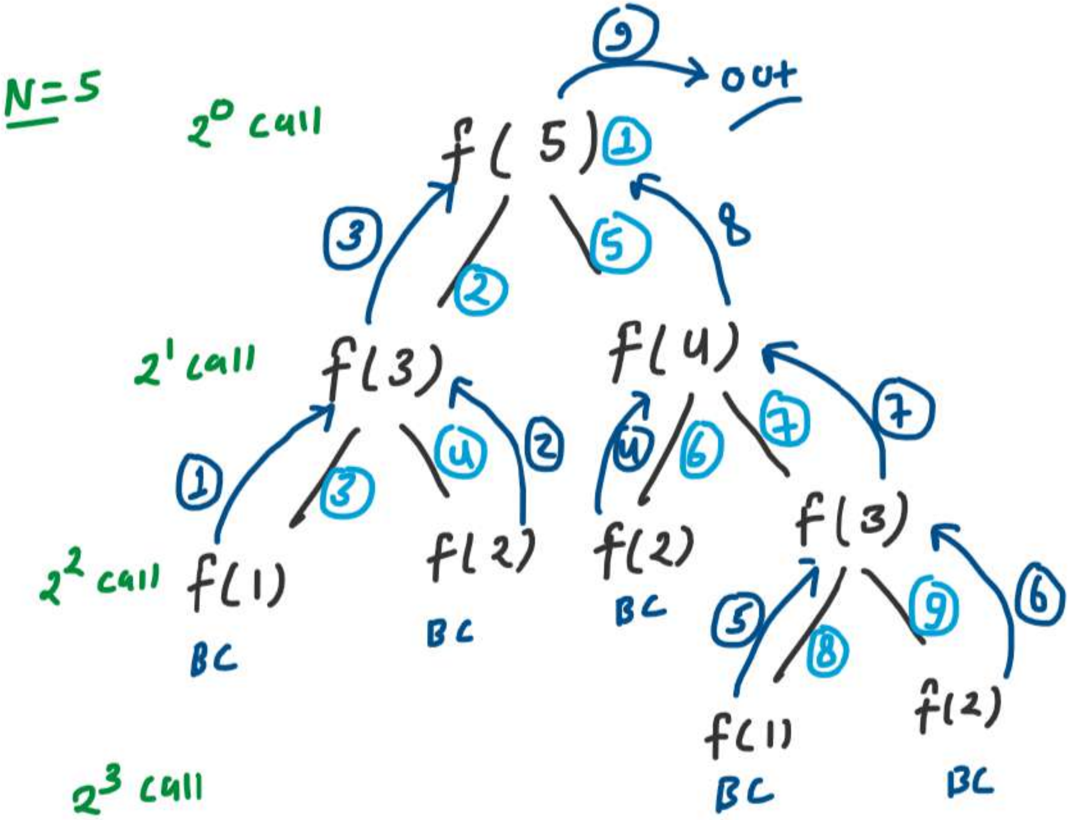
$$\boxed{\text{Total ways } f(N) = (K-1) * [f(n-2) + f(n-1)]}$$

RECURSIVE
RELATION

Approach 1: Recursion

```cpp
// 1. Painting Fence Algorithm (GFG)

// ================== Approach 1: Normal Recursion Approach ================== //

#include<iostream>
using namespace std;

int solveUsingRec(int n, int k){
    // Base Case
    if(n == 1){
        return k;
    }
    if(n == 2){
        return (k + (k*(k-1)));
    }

    // Recursive Relation
    int ans = (k-1) * (solveUsingRec(n-2,k) + solveUsingRec(n-1, k));
    return ans;
}

int main(){
    int n = 3; // Posts
    int k = 3; // Colors

    int ans = solveUsingRec(n,k);
    cout<<"Total Ways: "<< ans <<endl;
}
```

Output ⇒ SAME = 6
⇒ Diff = 18
Total = 24

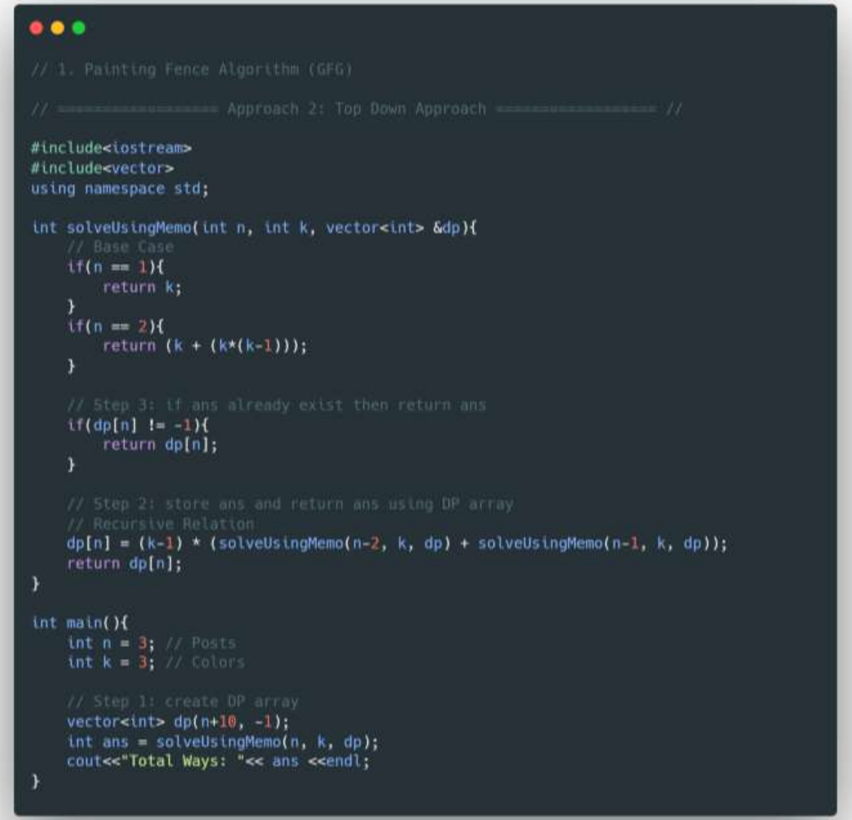


Total Entry in stack = 5
= N
S.C. = O(N)

$$T.C. = O(2^3)$$
$$= O(2^{N-2}) \Rightarrow O(2^N)$$

Approach 2: Top Down

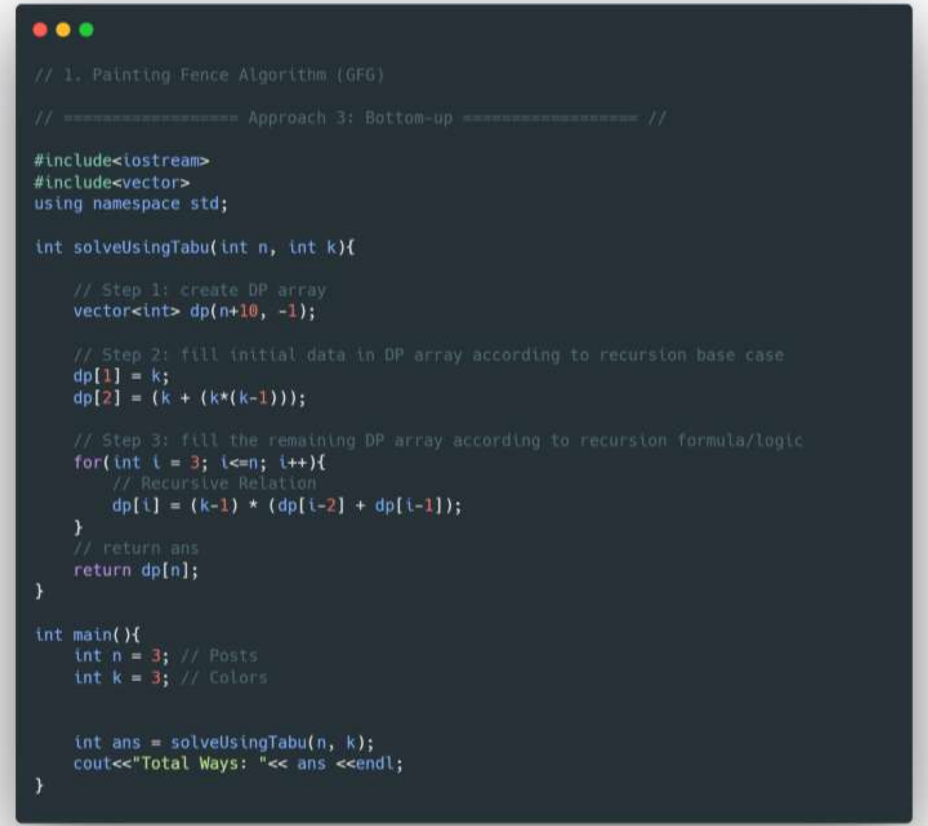
TOP DOWN: Traverse from N to 1



```cpp
// 1. Painting Fence Algorithm (GFG)

// ================== Approach 2: Top Down Approach ================== //

#include<iostream>
#include<vector>
using namespace std;

int solveUsingMemo(int n, int k, vector<int> &dp){
    // Base Case
    if(n == 1){
        return k;
    }
    if(n == 2){
        return (k + (k*(k-1)));
    }

    // Step 3: if ans already exist then return ans
    if(dp[n] != -1){
        return dp[n];
    }

    // Step 2: store ans and return ans using DP array
    // Recursive Relation
    dp[n] = (k-1) * (solveUsingMemo(n-2, k, dp) + solveUsingMemo(n-1, k, dp));
    return dp[n];
}

int main(){
    int n = 3; // Posts
    int k = 3; // Colors

    // Step 1: create DP array
    vector<int> dp(n+10, -1);
    int ans = solveUsingMemo(n, k, dp);
    cout<<"Total Ways: "<< ans <<endl;
}
```

Approach 3: Bottom Up

TOP DOWN: Traverse from 1 to N



```cpp
// 1. Painting Fence Algorithm (GFG)

// ================== Approach 3: Bottom-up ================== //

#include<iostream>
#include<vector>
using namespace std;

int solveUsingTabu(int n, int k){

    // Step 1: create DP array
    vector<int> dp(n+10, -1);

    // Step 2: fill initial data in DP array according to recursion base case
    dp[1] = k;
    dp[2] = (k + (k*(k-1)));

    // Step 3: fill the remaining DP array according to recursion formula/logic
    for(int i = 3; i<=n; i++){
        // Recursive Relation
        dp[i] = (k-1) * (dp[i-2] + dp[i-1]);
    }
    // return ans
    return dp[n];
}

int main(){
    int n = 3; // Posts
    int k = 3; // Colors


    int ans = solveUsingTabu(n, k);
    cout<<"Total Ways: "<< ans <<endl;
}
```

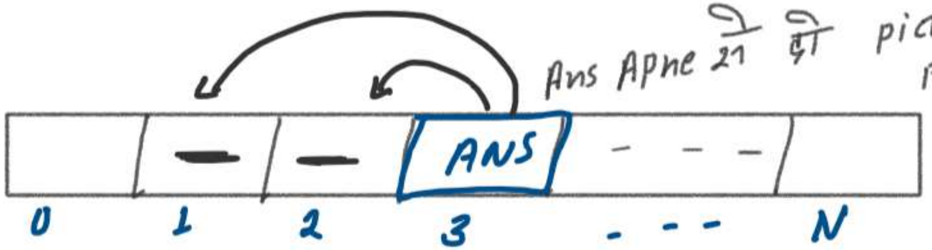
Approach 4: Space Optimization

RECURSIVE RELATION

$$DP[i] = \{DP[i-1] + DP[i-2]\} * (K-1)$$

ANS

DP[i] → Depends on → DP[i-1], DP[i-2]



Ans Apne दो ही picnle elements par Depend kar rha hai

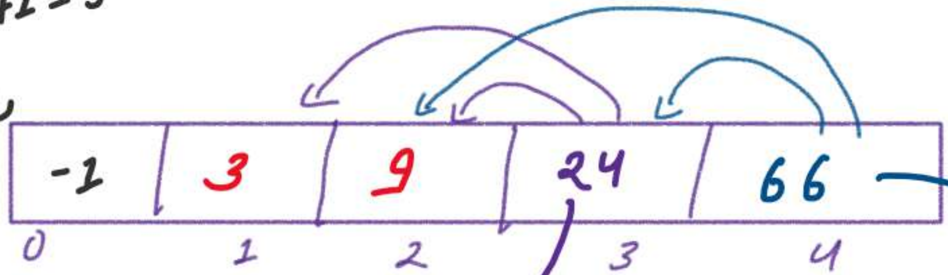
↳ MTLB EK PaHurn to BANTA HUAA DIKH RHA HAI

EX

K=3
N=4

size = N+1 = 5

DP



0	1	2	3	4
-1	3	9	24	66

DP[3] = [DP[2] + DP[1]] * (K-1)

= (9 + 3) * (3-1) ⇒ (12) * (2) = 24

DP[4] = [DP[3] + DP[2]] * (K-1)

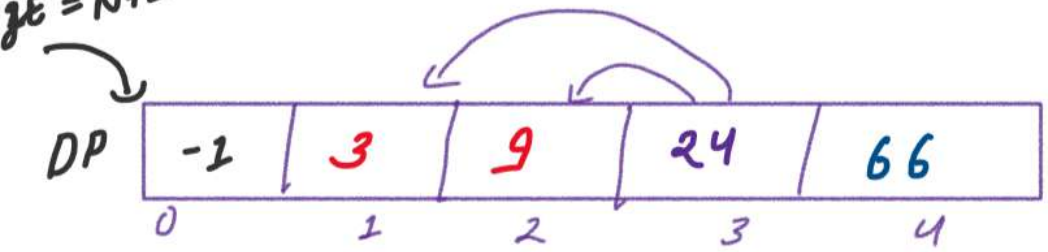
= (24 + 9) * (3-1) ⇒ (33) * (2) = 66

CREAT PATTERN

EX

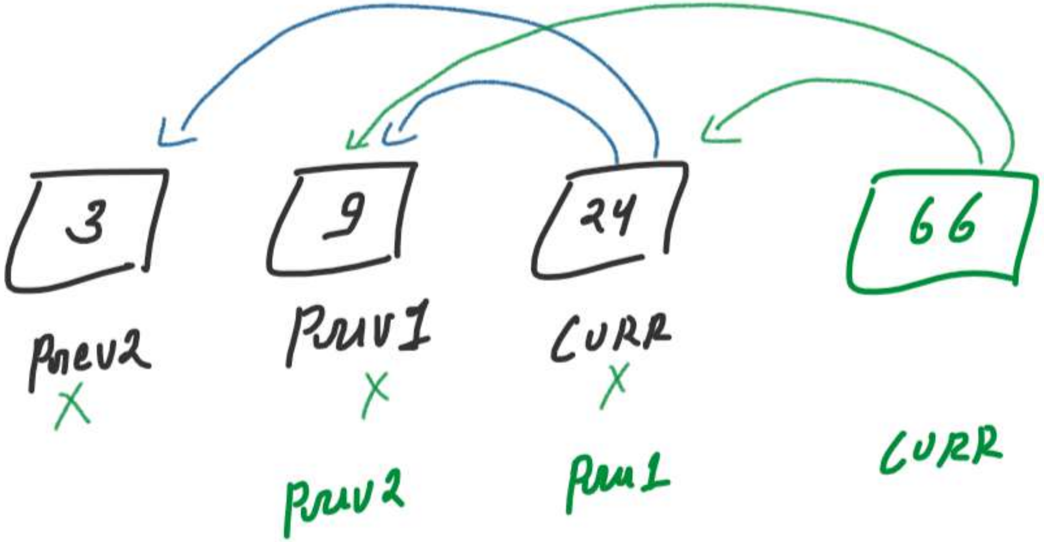
K=3
N=4

size = N+1 = 5



SHIFT करना Always Bhool जाता Hu

Prev2 = Prev1
Prev1 = CURR



```cpp
// 1. Painting Fence Algorithm (GFG)

// ================== Approach 4: Space Optimization Approach ================== //

#include<iostream>
using namespace std;

int solveUsingTabuOS(int n, int k){

    int prev2 = k;
    int prev1 = (k + (k*(k-1)));

    // Corner Cases
    //(agar N ki value at least 3 hoti to corner case ki koi need hi nhi thi)
    if(n == 1){
        return prev2;
    }
    if (n == 2)
    {
        return prev1;
    }

    int curr;

    for(int i = 3; i<=n; i++){
        // Recursive Relation
        curr = (k-1) * (prev1 + prev2);

        // Shift Karna Bhool Jata hu
        prev2 = prev1;
        prev1 = curr;
    }

    // return ans
    return curr;
}

int main(){
    int n = 3; // Posts
    int k = 3; // Colors

    int ans = solveUsingTabuOS(n, k);
    cout<<"Total Ways: "<< ans <<endl;
}
```

Time complexity = $O(N)$

Space complexity = $O(1)$

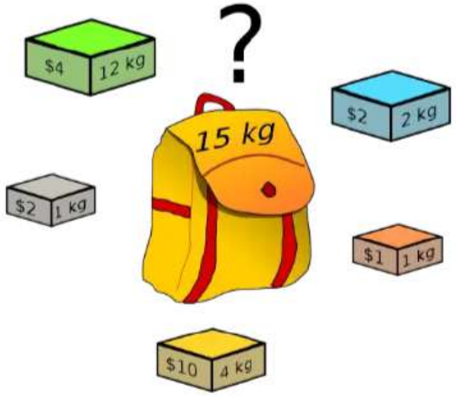
Where N is number of posts

2. 0/1 Knapsack Problem (GFG)

Problem Statement:
*Given **N items** where each item has some **weight** and **profit** associated with it and also given a **bag** with **capacity W**, (**i.e., the bag can hold at most W weight in it**).*

Return Kya Karana Hai:
*The task 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).*



Knapsack Problem

```
Examples:

Input:
N = 3, W = 50,  weight[] = {10, 20, 30}, profit[] = {60, 100, 120}
Output: 220

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

Input:
N = 3, W = 6,  weight[] = {1, 2, 3}, profit[] = {10, 15, 40}
Output: 65

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

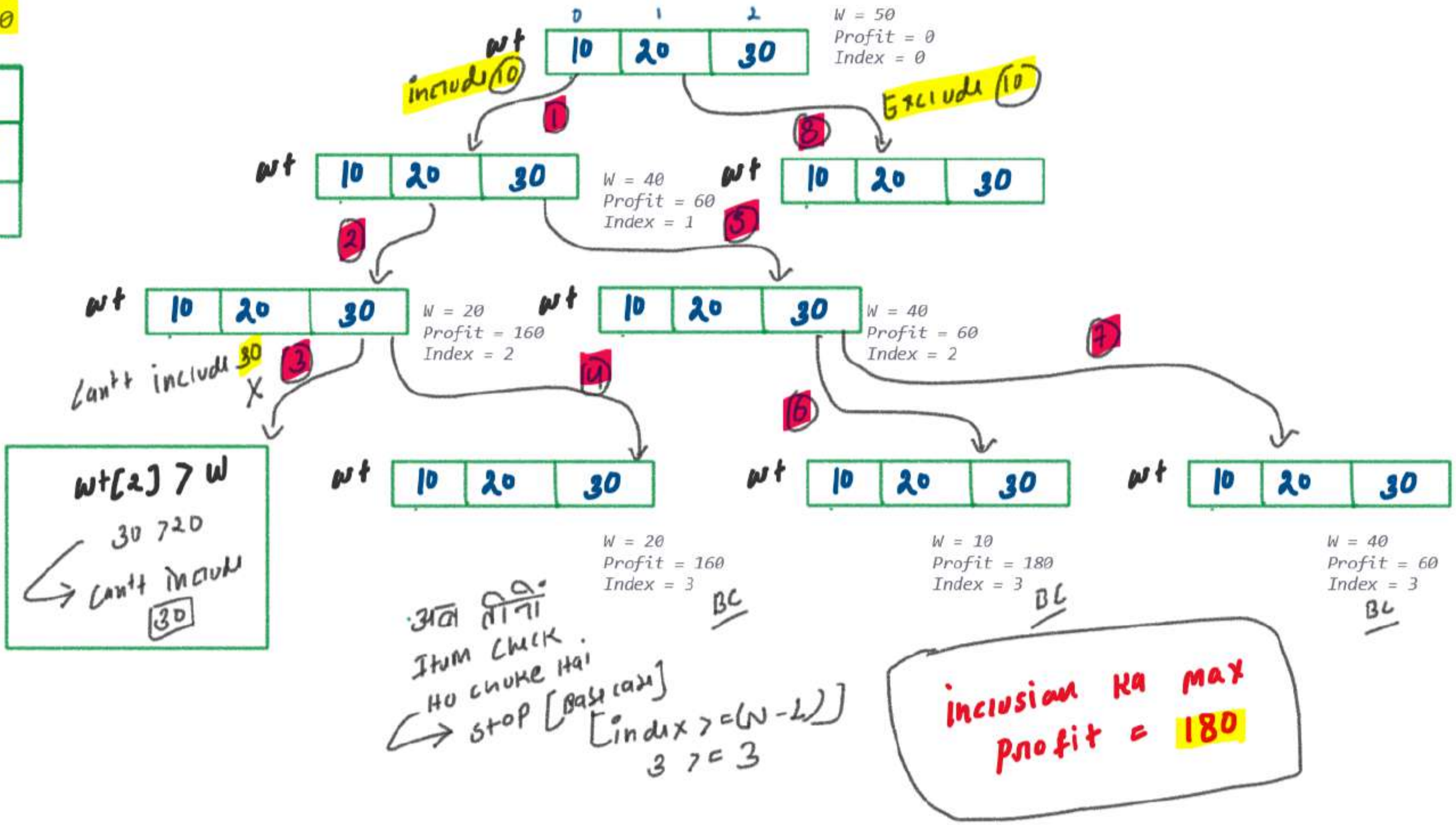
Approach 1: Recursion
Inclusive and Exclusive Pattern

Input: **Output: 220**

weight	10	20	30
profit	60	100	120
Index	0	1	2

Items n = 3

Capacity W = 50



wt[2] > W
30 > 20
→ can't include 30

अब तीनों Item Check Ho chuke Hai.
→ stop [Base case]
[index >= (N-1)]
3 >= 3

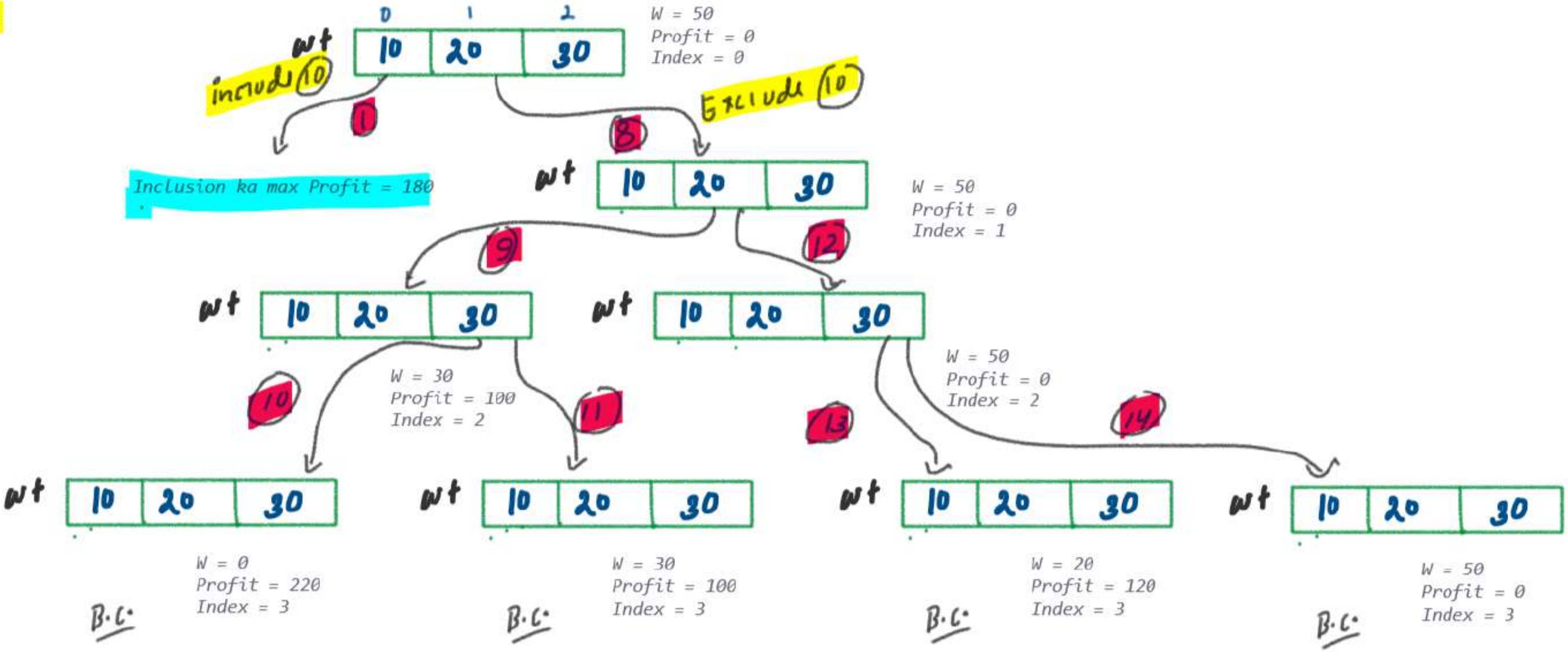
inclusion ka max Profit = 180

Input: **Output:** 220

weight	10	20	30
Profit	60	100	120
Index	0	1	2

Items n= 3

Capacity W = 50



Inclusion ka max Profit = 180

Exclusion ka max Profit = 220

wt

0	1	2
10	20	30

W = 50
Profit = 0
Index = 0

include (10)

Exclude (10)

Inclusion ka max Profit = 180

Exclusion ka max Profit = 220

Inclusion and Exclusion ka max Profit = 220

output

BAG [KNAPSACK]

RECURSIVE RELATION

inclusion = put curr item ka profit + $\boxed{REC}$

Exclusion = put zero profit + $\boxed{REC}$

maxProfit = max(inclusion , Exclusion)

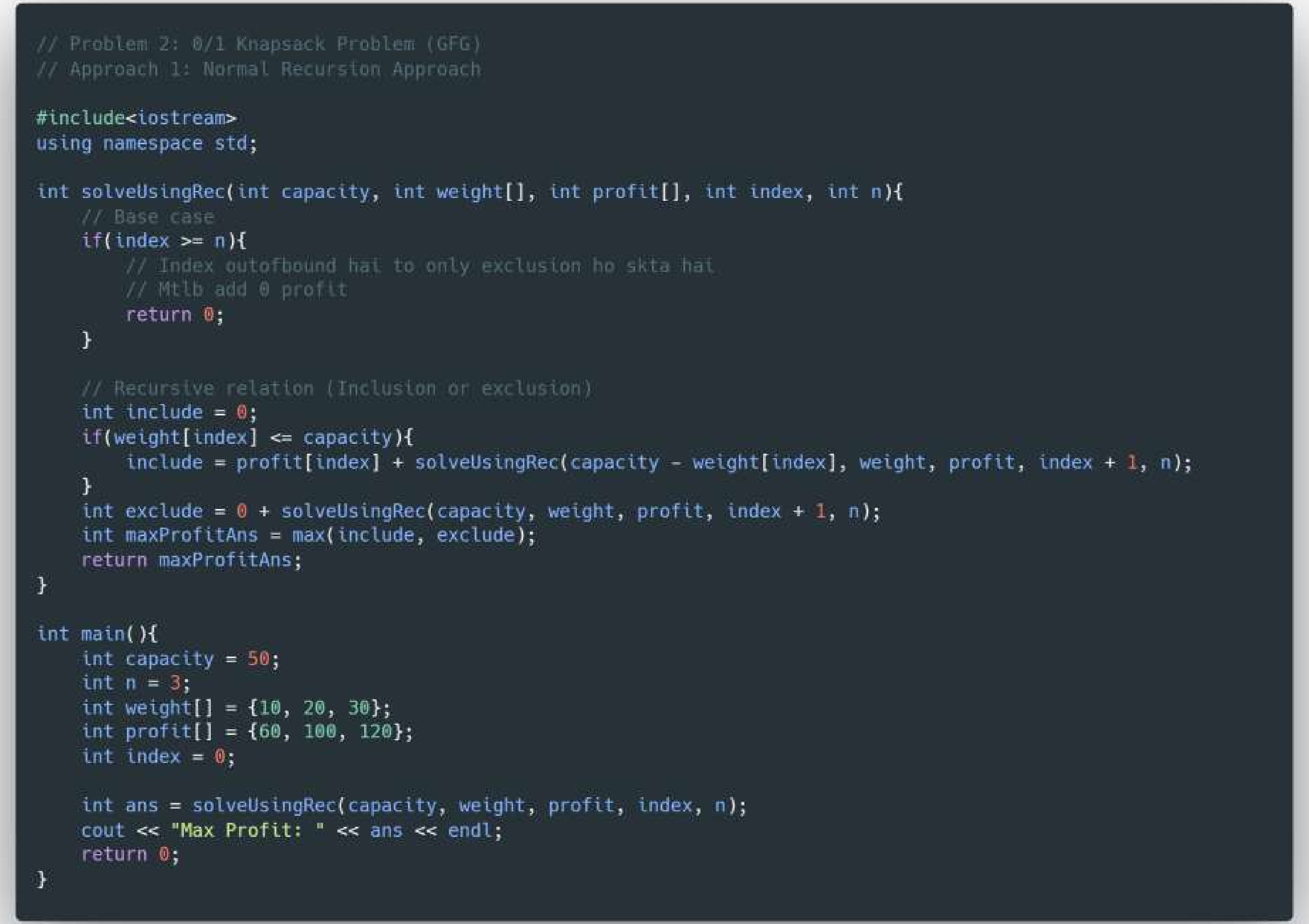
Base case

```
if(index >= N){
    return 0 profit
}
```

OR

Base case

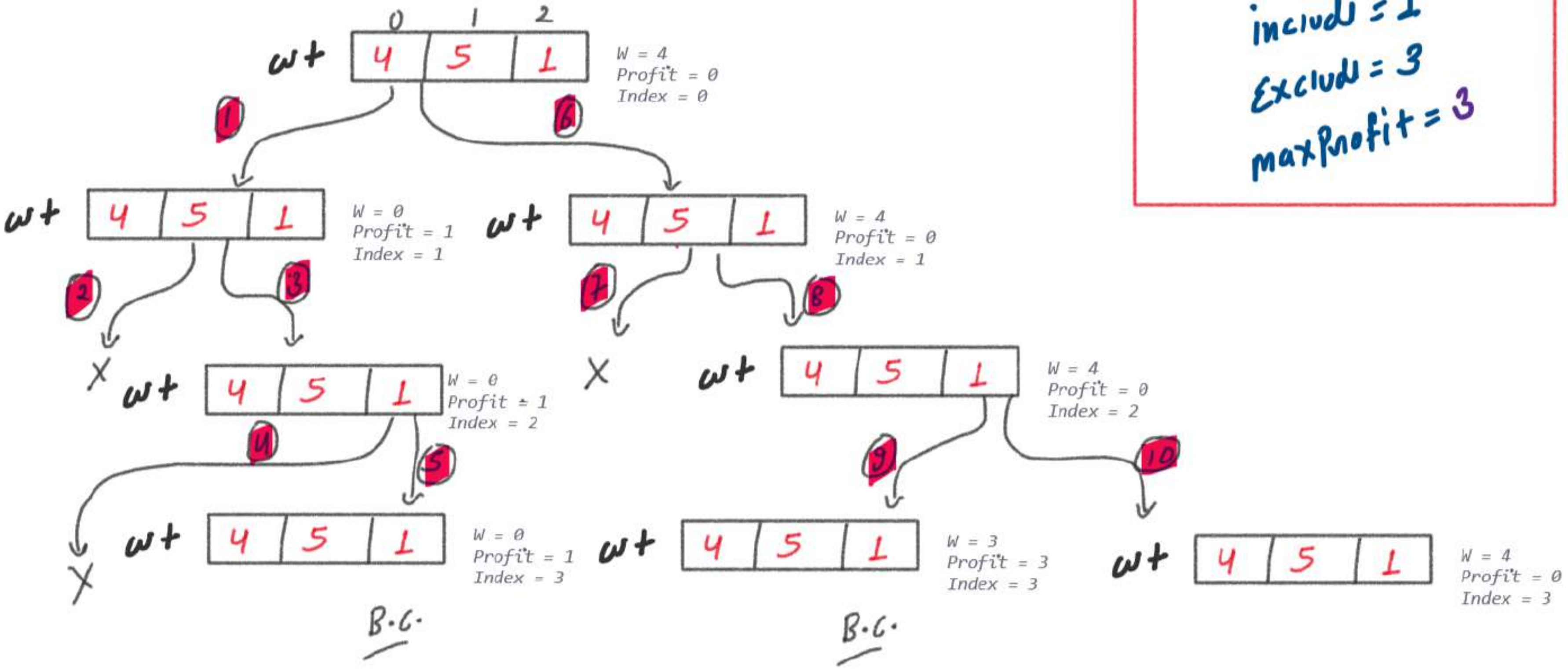
```
if( index >= N-1) {
    if( wt[index] <= W) {
        ↳ include profit
    }
    else{
        ↳ return 0 profit
    }
}
```



```
// Problem 2: 0/1 Knapsack Problem (GFG)
// Approach 1: Normal Recursion Approach

#include<iostream>
using namespace std;

int solveUsingRec(int capacity, int weight[], int profit[], int index, int n){
    // Base case
    if(index >= n){
        // Index outofbound hai to only exclusion ho skta hai
        // Mtlb add 0 profit
        return 0;
    }

    // Recursive relation (Inclusion or exclusion)
    int include = 0;
    if(weight[index] <= capacity){
        include = profit[index] + solveUsingRec(capacity - weight[index], weight, profit, index + 1, n);
    }
    int exclude = 0 + solveUsingRec(capacity, weight, profit, index + 1, n);
    int maxProfitAns = max(include, exclude);
    return maxProfitAns;
}

int main(){
    int capacity = 50;
    int n = 3;
    int weight[] = {10, 20, 30};
    int profit[] = {60, 100, 120};
    int index = 0;

    int ans = solveUsingRec(capacity, weight, profit, index, n);
    cout << "Max Profit: " << ans << endl;
    return 0;
}
```

DRY RUN

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

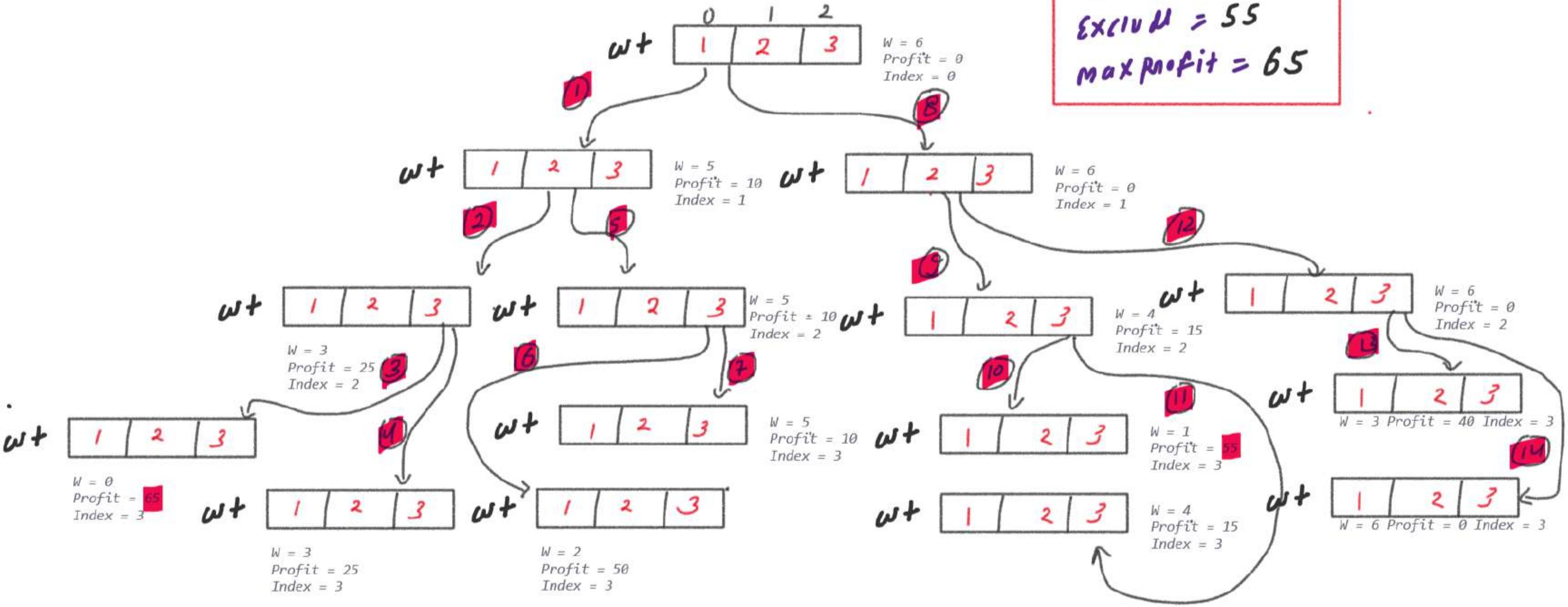


include = 1
Exclude = 3
maxProfit = 3

DRY RUN ✓

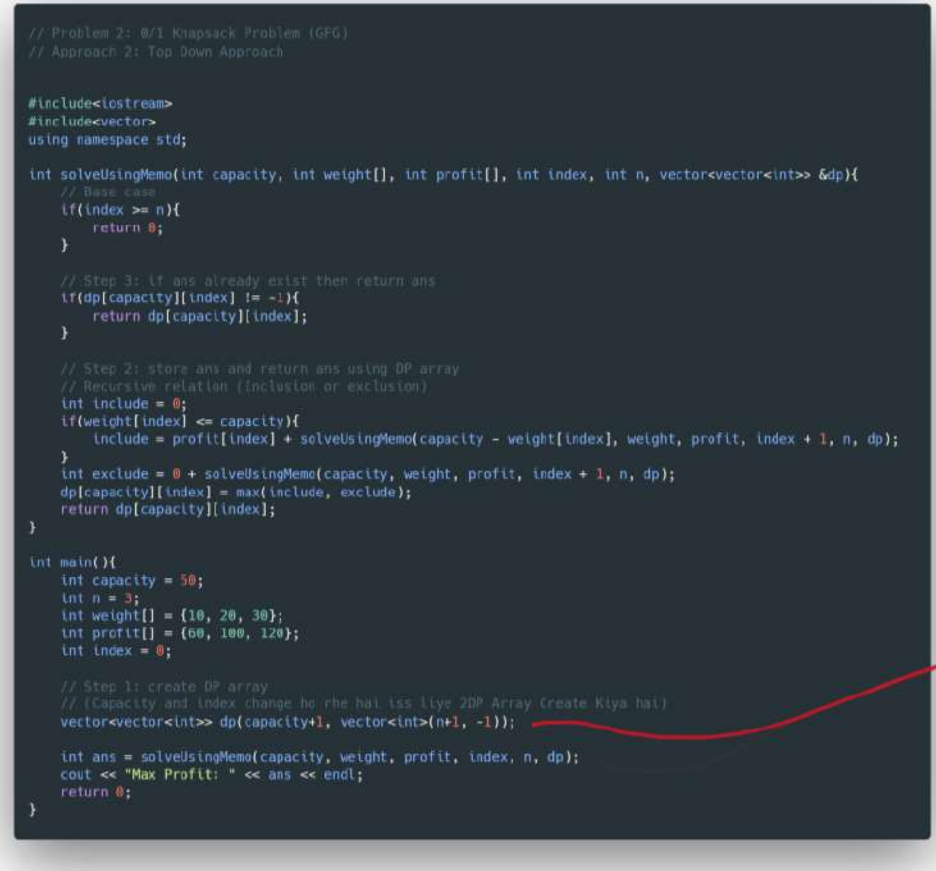
Input:
N = 3, W = 6, weight[] = {1, 2, 3}, profit[] = {10, 15, 40}
Output: 65

include = 65
exclude = 55
maxProfit = 65



Approach 2: Top Down

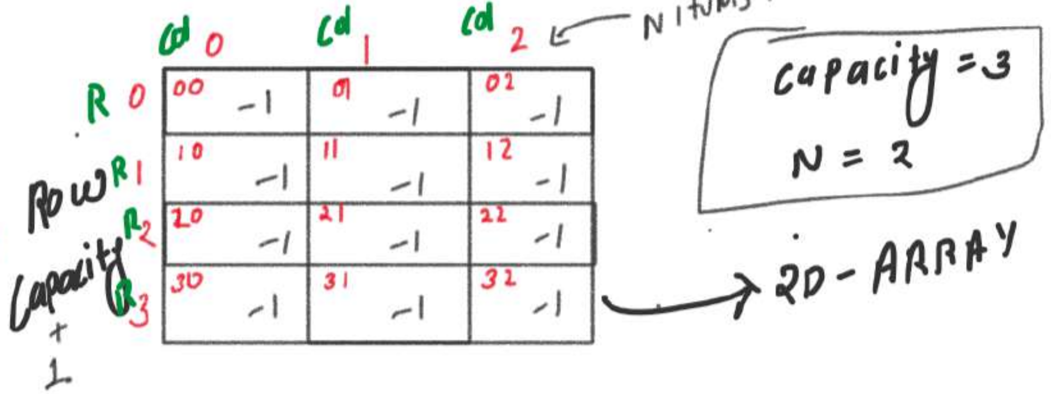
Inclusive and Exclusive Pattern



```cpp
// Problem 2: 0/1 Knapsack Problem (GFG)
// Approach 2: Top Down Approach


#include<iostream>
#include<vector>
using namespace std;

int solveUsingMemo(int capacity, int weight[], int profit[], int index, int n, vector<vector<int>> &dp){
    // Base case
    if(index >= n){
        return 0;
    }

    // Step 3: if ans already exist then return ans
    if(dp[capacity][index] != -1){
        return dp[capacity][index];
    }

    // Step 2: store ans and return ans using DP array
    // Recursive relation (Inclusion or exclusion)
    int include = 0;
    if(weight[index] <= capacity){
        include = profit[index] + solveUsingMemo(capacity - weight[index], weight, profit, index + 1, n, dp);
    }
    int exclude = 0 + solveUsingMemo(capacity, weight, profit, index + 1, n, dp);
    dp[capacity][index] = max(include, exclude);
    return dp[capacity][index];
}

int main(){
    int capacity = 50;
    int n = 3;
    int weight[] = {10, 20, 30};
    int profit[] = {60, 100, 120};
    int index = 0;

    // Step 1: create DP array
    // (Capacity and index change ho rhe hai iss liye 2DP Array Create Kiya hai)
    vector<vector<int>> dp(capacity+1, vector<int>(n+1, -1));

    int ans = solveUsingMemo(capacity, weight, profit, index, n, dp);
    cout << "Max Profit: " << ans << endl;
    return 0;
}
```

TRAVERSE FROM CAPACITY
TO 0

TRAVERSE FROM 0 index
TO N index



	Col 0	Col 1	Col 2
R 0	00 -1	01 -1	02 -1
R 1	10 -1	11 -1	12 -1
R 2	20 -1	21 -1	22 -1
R 3	30 -1	31 -1	32 -1

capacity = 3
N = 2

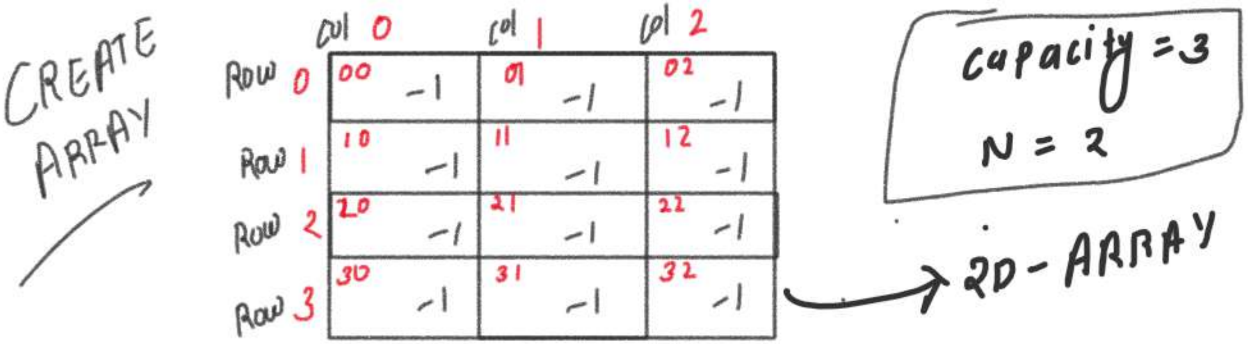
Approach 3: Bottom Up

Inclusive and Exclusive Pattern

```cpp
// Problem 2: 0/1 Knapsack Problem (GFG)
// Approach 3: Bottom Up Approach


#include<iostream>
#include<vector>
using namespace std;

int solveUsingTabu(int capacity, int weight[], int profit[], int index, int n){
    // Step 1: create DP array
    // (Capacity and index change ho rhe hai iss liye 2DP Array Create Kiya hai)
    vector<vector<int>> dp(capacity+1, vector<int>(n+1, -1));

    // Step 2: fill initial data in DP array according to recursion base case
    // Last Column ko zero se fill krna hai according to base case
    for(int row = 0; row<=capacity; row++){
        dp[row][n] = 0;
    }

    // Step 3: fill the remaining DP array according to recursion formula/logic
    for(int row=0; row<=capacity; row++){
        for(int col=n-1; col>=0; col--){
            // Recursive relation (Inclusion or exclusion)
            int include = 0;
            if(weight[col] <= row){
                include = profit[col] + dp[row - weight[col]][col + 1];
            }
            int exclude = 0 + dp[row][col + 1];
            dp[row][col] = max(include, exclude);
        }
    }
    // return ans
    return dp[capacity][0];
}

int main(){
    int capacity = 50;
    int n = 3;
    int weight[] = {10, 20, 30};
    int profit[] = {60, 100, 120};
    int index = 0;

    int ans = solveUsingTabu(capacity, weight, profit, index, n);
    cout << "Max Profit: " << ans << endl;
    return 0;
}
```

TRAVERSE FROM 0
TO capacity

TRAVERSE FROM N index
TO 0 index



CREATE ARRAY

	col 0	col 1	col 2
Row 0	00 -1	01 -1	02 -1
Row 1	10 -1	11 -1	12 -1
Row 2	20 -1	21 -1	22 -1
Row 3	30 -1	31 -1	32 -1

→ 2D - ARRAY

capacity = 3
N = 2

DRY RUN

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

STEP 1

Capacity

Items N

	Col 0	Col 1	Col 2	Col 3
Row0	-1	-1	-1	-1
Row1	-1	-1	-1	-1
Row2	-1	-1	-1	-1
Row3	-1	-1	-1	-1
Row4	-1	-1	-1	-1

STEP 2

Capacity

Items N

	Col 0	Col 1	Col 2	Col 3
Row0	-1	-1	-1	0
Row1	-1	-1	-1	0
Row2	-1	-1	-1	0
Row3	-1	-1	-1	0
Row4	-1	-1	-1	0

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

itration 1

Capacity ⇒ Row = 0

Item ⇒ Col = 2

include = 0

Exclude = 0

DP[Row][Col] = max(include , Exclude)

= max(0 , 0)

= 0

x 1 <= 0

(wt[Col] <= Row)

└→ Add Profit

STEP 3

Capacity → Row

Items N → Col

	Col 0	Col 1	Col 2	Col 3
Row0	00 -1	01 -1	02 0	03 0
Row1	10 -1	11 -1	12 -1	13 0
Row2	20 -1	21 -1	22 -1	23 0
Row3	30 -1	31 -1	32 -1	33 0
Row4	40 -1	41 -1	42 -1	43 0

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

weight indices: 0 1 2   profit indices: 0 1 2

iteration 2

Capacity ⇒ Row = 0

Item ⇒ Col = 1

include = 0

Exclude = 0

DP[Row][col] = max(inclu , Exclu)
= max(0 , 0)
= 0

X 5 <= 0
(wt[col] <= Row)
└→ Add Profit

Capacity ↓ Items N →

STEP 3

	Col 0	Col 1	Col 2	Col 3
Row0	00 -1	01 0	02 0	03 0
Row1	10 -1	11 -1	12 -1	13 0
Row2	20 -1	21 -1	22 -1	23 0
Row3	30 -1	31 -1	32 -1	33 0
Row4	40 -1	41 -1	42 -1	43 0

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

0 1 2 0 1 2

iterati 3

Capacity ⇒ Row = 0

Item ⇒ Col = 0

include = 0

Exclude = 0

DP[Row][col] = max(include, Exclude)
= max(0, 0)
= 0

$\overset{x}{(wt}\overset{4}{[}col] \overset{<=0}{<=} Row)$

↳ Add Profit

Capacity

Items N

STEP 3

	Col 0	Col 1	Col 2	Col 3
Row0	00 0	01 0	02 0	03 0
Row1	10 -1	11 -1	12 -1	13 0
Row2	20 -1	21 -1	22 -1	23 0
Row3	30 -1	31 -1	32 -1	33 0
Row4	40 -1	41 -1	42 -1	43 0

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

iteration

Capacity => Row = 1

Item => Col = 2

include = 3

Exclude = DP[Row][col+1] = 0

DP[Row][col] = max(include , Exclude)
= max(3 , 0)
= 3

✓ 1 <= 1
(wt[col] <= Row)
→ Add Profit

STEP 3

Capacity

Items N

	Col 0	Col 1	Col 2	Col 3
Row0	00 0	01 0	02 0	03 0
Row1	10 -1	11 -1	12 3	13 0
Row2	20 -1	21 -1	22 -1	23 0
Row3	30 -1	31 -1	32 -1	33 0
Row4	40 -1	41 -1	42 -1	43 0

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

iteratis

Capacity => Row = 1

Item => Col = 1

include = 0

Exclude = DP[Row][col+1] = 3

DP[Row][col] = max(include, Exclude)

= max(0 , 3)

= 3

∝ 5   <= 1

(wt[col] <= Row)

└→ Add Profit

STEP 3

Capacity

Items N

	Col 0	Col 1	Col 2	Col 3
Row0	00 0	01 0	02 0	03 0
Row1	10 -1	11 3	12 3	13 0
Row2	20 -1	21 -1	22 -1	23 0
Row3	30 -1	31 -1	32 -1	33 0
Row4	40 -1	41 -1	42 -1	43 0

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

iteration 6

Capacity => Row = 1

Item => Col = 0

include = 0

Exclude = DP[Row][col+1] = 3

DP[Row][col] = max(inclu , Exclu)
= max(0 , 3)
= 3

✗ 4 <= 1
(wt[col] <= Row)
└→ Add Profit

STEP 3

Capacity

Items N

	Col 0	Col 1	Col 2	Col 3
Row0	00 0	01 0	02 0	03 0
Row1	10 3	11 3	12 3	13 0
Row2	20 -1	21 -1	22 -1	23 0
Row3	30 -1	31 -1	32 -1	33 0
Row4	40 -1	41 -1	42 -1	43 0

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

weight indices: 0 1 2
profit indices: 0 1 2

iterati 7

Capacity => Row = 2

Item => Col = 2

include = 3

Exclude = DP[Row][col+1] = 0

DP[Row][col] = max(inclu , Exclu)

= max(3 , 0)

= 3

$\checkmark$ $(wt[\overset{1}{col}] \overset{\le 2}{\le} Row)$

↳ Add Profit

STEP 3

Capacity

Items N

	Col 0	Col 1	Col 2	Col 3
Row0	00 0	01 0	02 0	03 0
Row1	10 3	11 3	12 3	13 0
Row2	20 -1	21 -1	22 3	23 0
Row3	30 -1	31 -1	32 -1	33 0
Row4	40 -1	41 -1	42 -1	43 0

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

iteration 8

Capacity => Row = 2

Item => Col = 1

include = 0

Exclude = DP[Row][col+1] = 3

DP[Row][col] = max(include, Exclude)

= max(0, 3)

= 3

✗ 5 <= 2

(wt[col] <= Row)

↳ Add Profit

STEP 3

Capacity

Items N

	Col 0	Col 1	Col 2	Col 3
Row0	00 0	01 0	02 0	03 0
Row1	10 3	11 3	12 3	13 0
Row2	20 -1	21 3	22 3	23 0
Row3	30 -1	31 -1	32 -1	33 0
Row4	40 -1	41 -1	42 -1	43 0

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

weight indices: 0 1 2 — profit indices: 0 1 2

iteration 9

Capacity ⇒ Row = 2

Item ⇒ Col = 0

include = 0

Exclude = DP[Row][col+1] = 3

DP[Row][col] = max(include, exclude)

= max(0 , 3)

= 3

✗ 4 <= 2

(wt[col] <= Row)

↳ Add Profit

Capacity

Items N

STEP 3

	Col 0	Col 1	Col 2	Col 3
Row0	00 0	01 0	02 0	03 0
Row1	10 3	11 3	12 3	13 0
Row2	20 3	21 3	22 3	23 0
Row3	30 -1	31 -1	32 -1	33 0
Row4	40 -1	41 -1	42 -1	43 0

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

iteration 10

Capacity ⇒ Row = 3

Item ⇒ Col = 2

include = 3+0 = 3

Exclude = DP[Row][col+1] = 0

DP[Row][col] = max(include, Exclude)
= max(3 , 0)
= 3

✓ 1 <= 3
(wt[col] <= Row)
↳ Add Profit

STEP 3

Capacity

Items N

	Col 0	Col 1	Col 2	Col 3
Row0	00 0	01 0	02 0	03 0
Row1	10 3	11 3	12 3	13 0
Row2	20 3	21 3	22 3	23 0
Row3	30 -1	31 -1	32 3	33 0
Row4	40 -1	41 -1	42 -1	43 0

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

iteration 11

Capacity ⇒ Row = 3

Item ⇒ Col = 1

include = 0

Exclude = DP[Row][col+1] = 3

DP[Row][col] = max(include, Exclude)
= max(3 , 3)
= 3

✕ 5 <= 3
(wt[col] <= Row)
↳ Add Profit

Capacity

Items N

STEP 3

	Col 0	Col 1	Col 2	Col 3
Row0	00 0	01 0	02 0	03 0
Row1	10 3	11 3	12 3	13 0
Row2	20 3	21 3	22 3	23 0
Row3	30 -1	31 3	32 3	33 0
Row4	40 -1	41 -1	42 -1	43 0

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

iteration 12

Capacity => Row = 3

Item => Col = 0

include = 0

Exclude = DP[Row][col+1] = 3

DP[Row][col] = max(inclu, Exclu)

= max(0 , 3)

= 3

x 4 <= 3

(wt[col] <= Row)

└→ Add Profit

Capacity

Items N

STEP 3

	Col 0	Col 1	Col 2	Col 3
Row0	00 0	01 0	02 0	03 0
Row1	10 3	11 3	12 3	13 0
Row2	20 3	21 3	22 3	23 0
Row3	30 3	31 3	32 3	33 0
Row4	40 -1	41 -1	42 -1	43 0

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

iteration 13

Capacity ⇒ Row = 4

Item ⇒ Col = 2

include = 3 + 0 = 3

Exclude = DP[Row][col+1] = 0

DP[Row][col] = max(include, Exclude)
= max(3, 0)
= 3

✓ 1 <= 4
(wt[col] <= Row)
↳ Add Profit

Capacity

Items N

STEP 3

	Col 0	Col 1	Col 2	Col 3
Row0	00 0	01 0	02 0	03 0
Row1	10 3	11 3	12 3	13 0
Row2	20 3	21 3	22 3	23 0
Row3	30 3	31 3	32 3	33 0
Row4	40 -1	41 -1	42 3	43 0

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

0 1 2   0 1 2

iterati 14

Capacity ⇒ Row = 4

Item ⇒ Col = 1

include = 0

Exclude = DP[Row][col+1] = 3

DP[Row][col] = max(include , Exclude)
= max(0 ,3)
= 3

X5 <= 4
(wt[col] <= Row)
↳ Add Profit

STEP 3

Capacity
Items N

	Col 0	Col 1	Col 2	Col 3
Row0	00 0	01 0	02 0	03 0
Row1	10 3	11 3	12 3	13 0
Row2	20 3	21 3	22 3	23 0
Row3	30 3	31 3	32 3	33 0
Row4	40 -1	41 3	42 3	43 0

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

itwrati 15

Capacity => Row = 4

Item => Col = 0

include = 1

Exclude = DP[Row][col+1] = 3

DP[Row][col] = max(inclu , Exclu)

= max(1 , 3)

= 3

✓ 4 <= 4

(wt[col] <= Row)

└→ Add Profit

STEP 3

Capacity

Items N

	Col 0	Col 1	Col 2	Col 3
Row0	00 0	01 0	02 0	03 0
Row1	10 3	11 3	12 3	13 0
Row2	20 3	21 3	22 3	23 0
Row3	30 3	31 3	32 3	33 0
Row4	40 $\boxed{3}$	41 3	42 3	43 0

output max Profit

Approach 4: Space Optimization

Inclusive and Exclusive Pattern

Pending