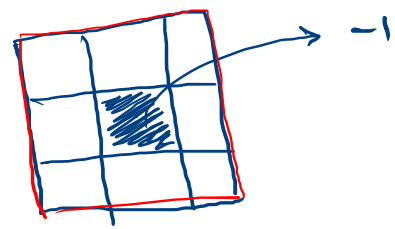
0/1 Knapsack Top-down

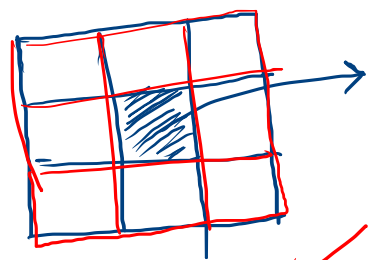
Dynamic Programming

Recursive Soln → Memoization (DP) → Top-Down (DP)

DP = Recursion + Storage

Recursive solution → Base Condition + RC → for smaller input

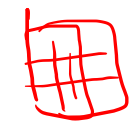
Memoization (DP) → RC +  -1

Top-down (DP) →  -1

Now,
why Top-down is better?

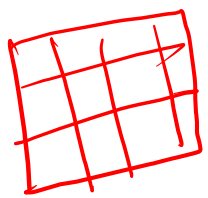
⇒ There is No use of Recursion ☺ ✓

⇒ NO risk. overlimit of function call stack is full



Top down

RC ✓



fun(n)

fun(n-1)

fun(n-2)

fun(n-3)

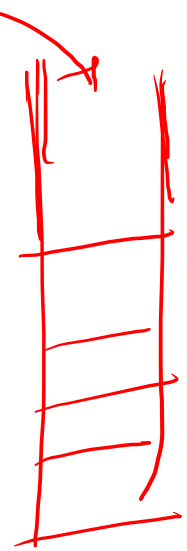
RC

RC

fd

mem

Top



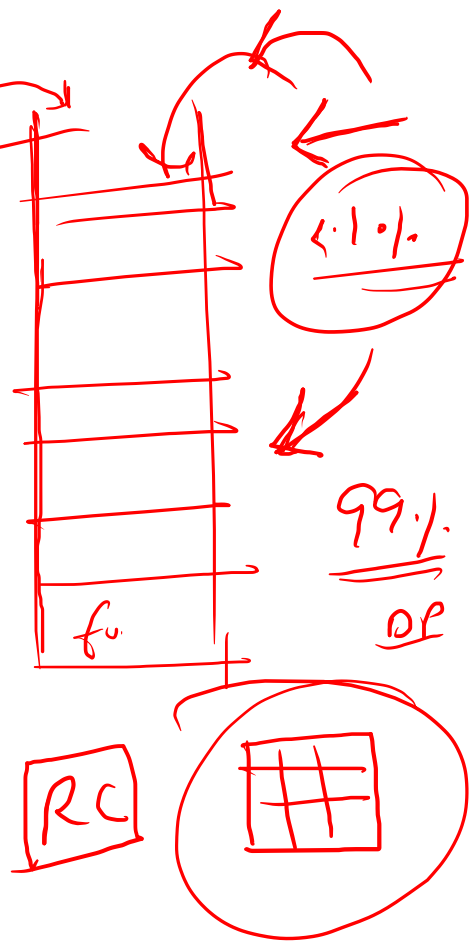
null

data

Complete

X

X



<1%

99%

op

fu

RC

Now understand

Top-down

dp[n+1][w+1]

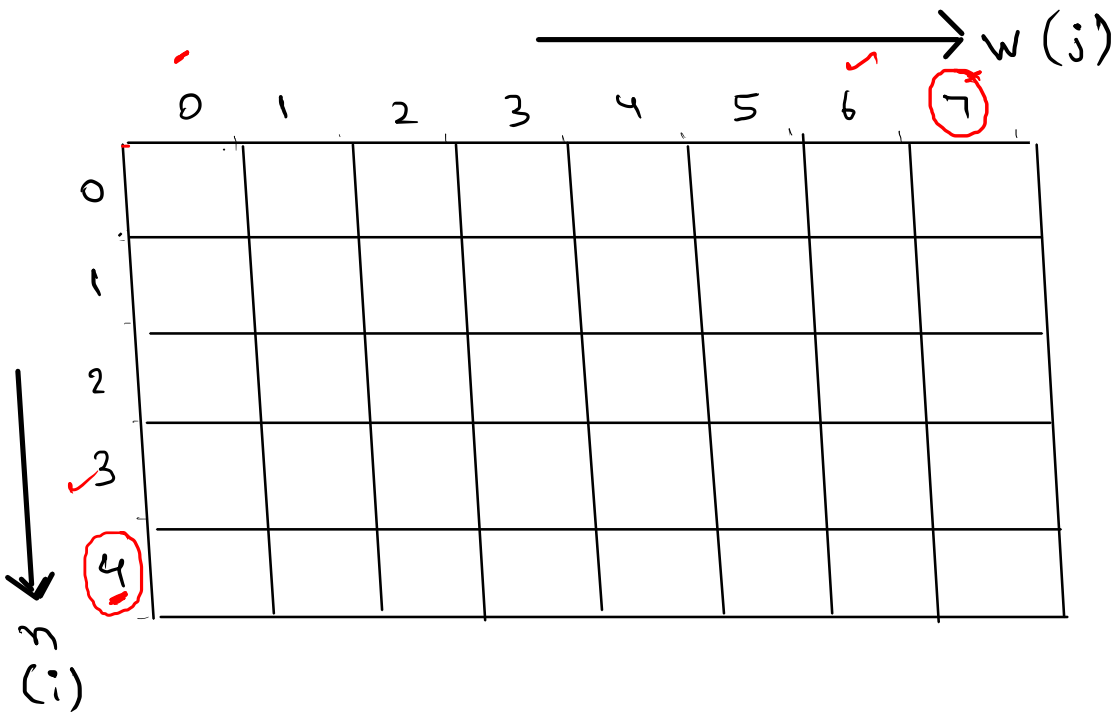
dp[4+1][7+1] = dp[5][8]

Items

wt[] = | 1 | 3 | 4 | 5 |

Pr[] = | 1 | 4 | 5 | 7 |

W = 7



Now understand Top-down

dp[n+1][w+1]

dp[4+1][7+1] = dp[5][8]

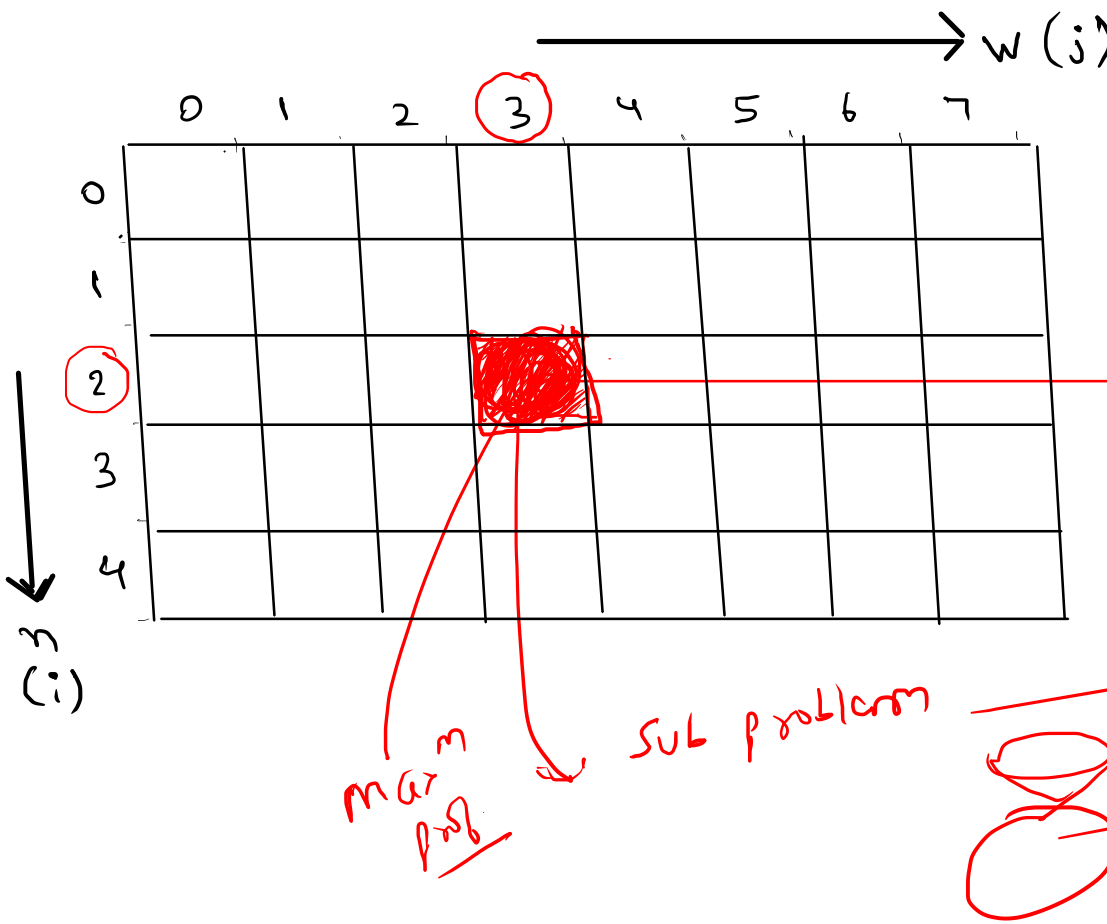
dp[5][8]

dp[][] store max^m profit

wt[] = | 1 | 3 | 4 | 5 |

pr[] = | 1 | 4 | 5 | 7 |

W = 7



→ [understood the meaning of each block in matrix]

n = 2

W = 3

wt[] = | 1 | 3 |

pr[] = | 1 | 4 |

W = 3

Now understand top-down

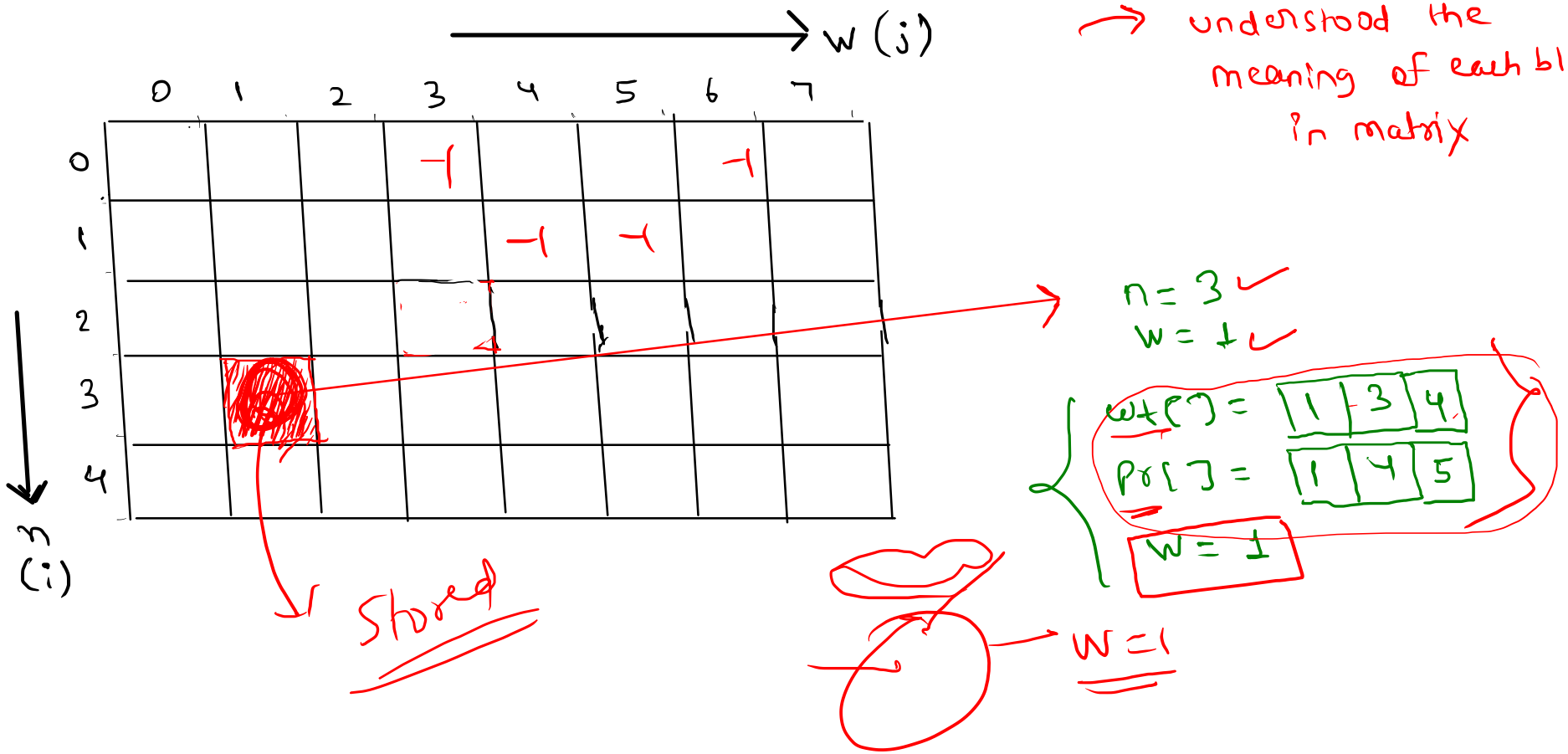
dp[n+1][w+1]

dp[4+1][7+1] = dp[5][8]

wt[] = | 1 | 3 | 4 | 5 |

pr[] = | 1 | 4 | 5 | 7 |

W = 7



→ understood the meaning of each block in matrix

Now understand Top-down

dp[n+1][w+1]

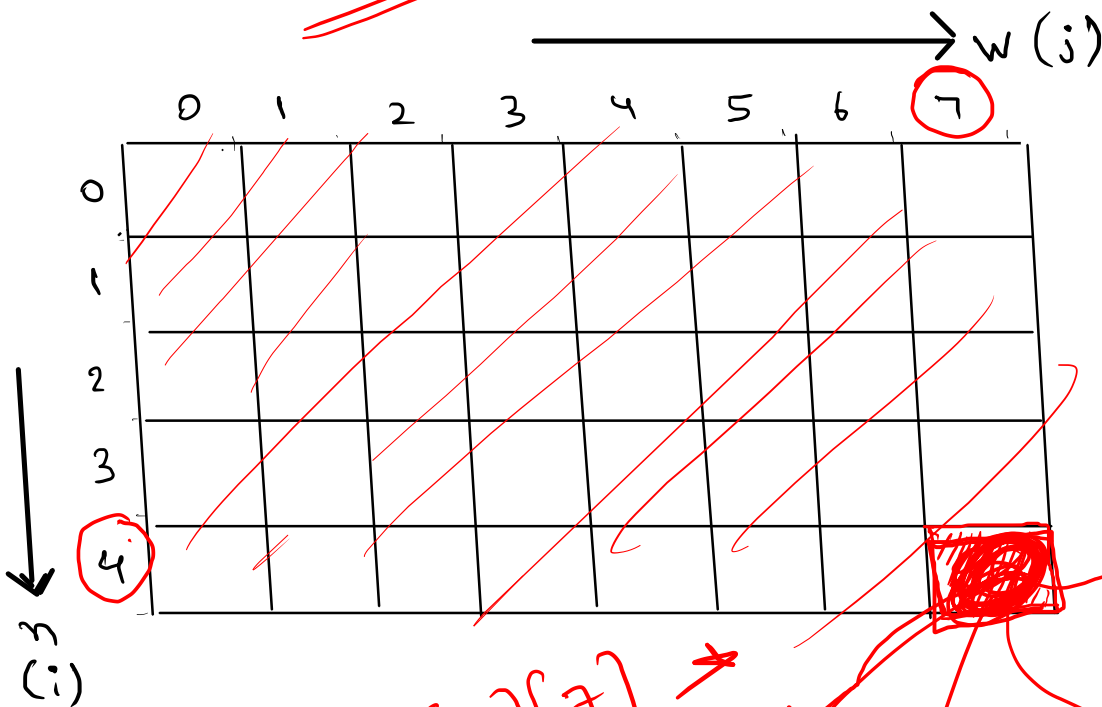
dp[4+1][7+1] = dp[5][8]

wt[] = | 1 | 3 | 4 | 5 |

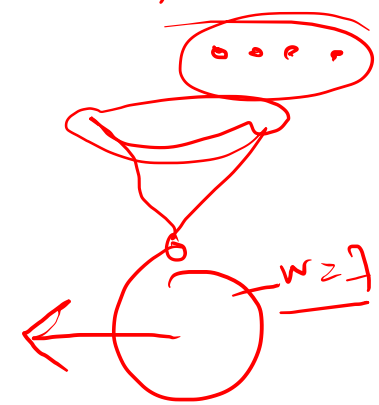
Pr[] = | 1 | 4 | 5 | 7 |

W = 7

n, w



→ understood the meaning of each block in matrix



return dp[4][7] →

Ans

Stored

Answer

Now understand Top-down

dp[n+1][w+1]

dp[4+1][7+1] = dp[5][8]

wt[] = | 1 | 3 | 4 | 5 |

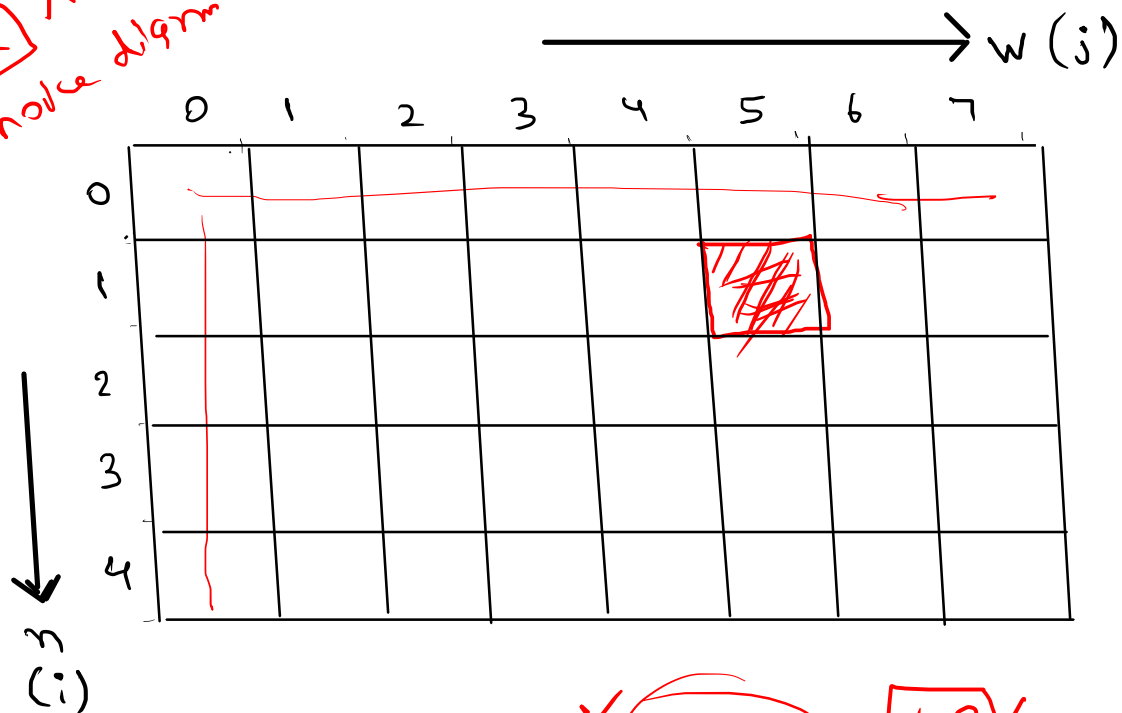
pr[] = | 1 | 4 | 5 | 7 |

W = 7

KP( ) → Base → Choice diagram

X

Y → Omit X



- In every block we will fill with maximum profit

→ we will fill first row and first column

↳ we do initialisation from the help of Base Condition.

X n=1 W=8

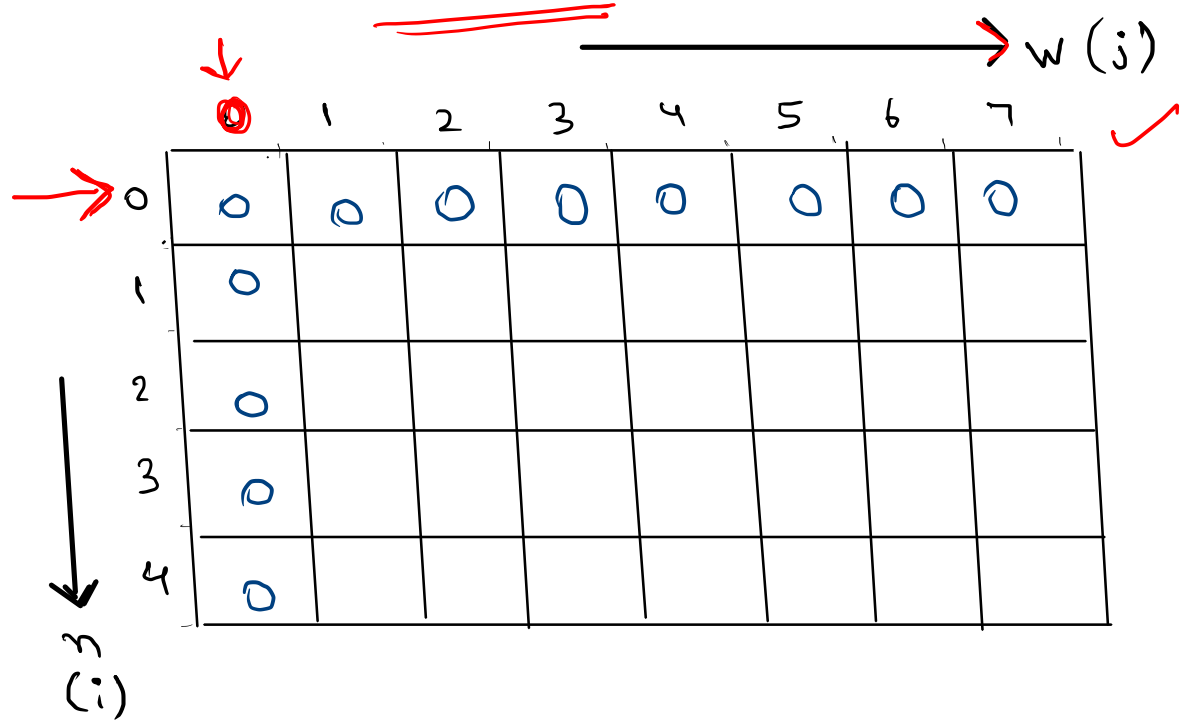
KP( ) {

Y

Base Condition of 0/1 knapsack

```
if ((n==0) || (W==0))
    return 0;   — max Profit
```

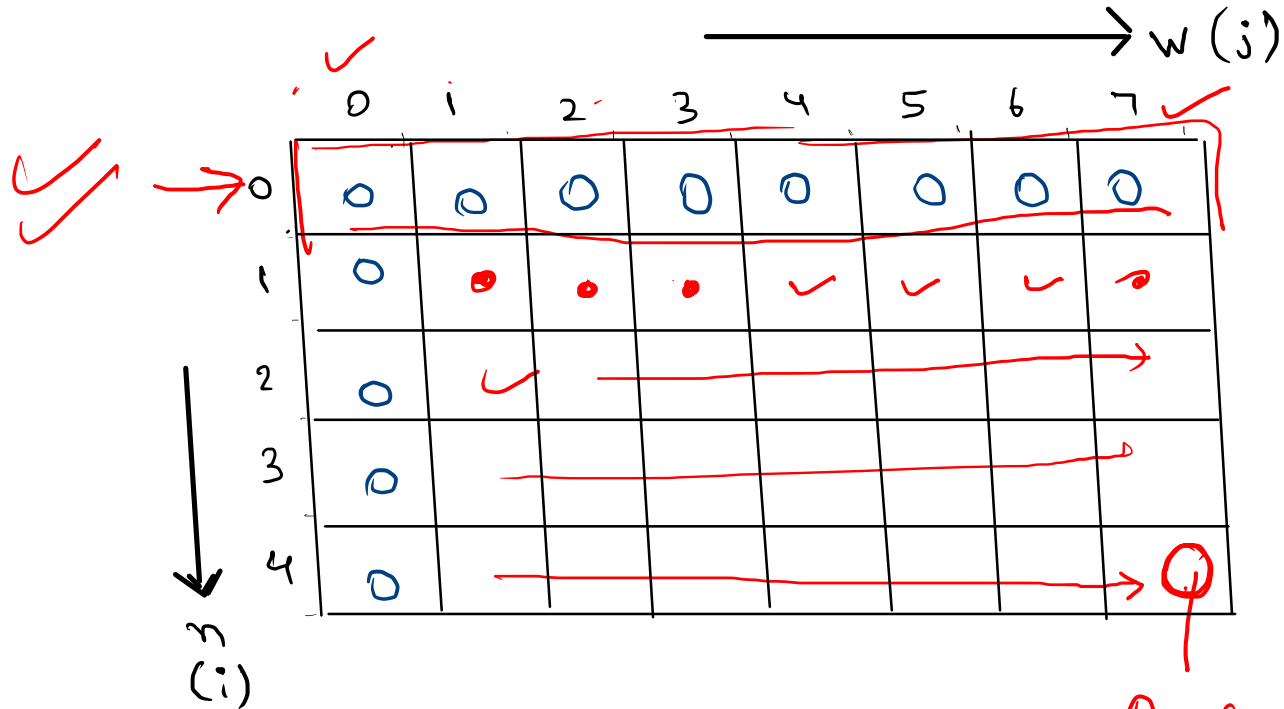
→ Every cell of matrix represent the max^m Profit.



Base Condition of 0/1 Knapsack

```
if ((n==0) || (W==0))
    return 0;
```

→ Every cell of matrix represent the maxm Profit.




```
→ for(int i=0; i<=n; i++)
    dp[i][0] = 0;
→ for(int i=0; i<=W; i++)
    dp[0][i] = 0;
```

initialization ↕ Base condition

KP(wt[], Pr[], W, n)

Recursive Code

Top down Code

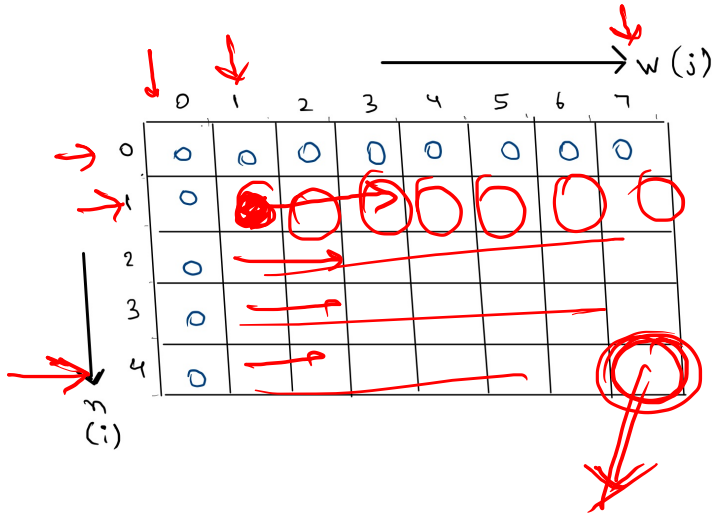
n → i
W → j
KP → DP

```
if(wt[n-1] <= W)
    dp[n][W] = max
    (
      Pr[n-1] + KP(wt[], Pr[], W - wt[n-1], n-1),
      KP(wt[], Pr[], W, n-1)
    )
else
    KP(wt[], Pr[], W, n-1)
```

```
if( wt[i-1] <= j)
    dp[i][j] = max
    (
      pr[i-1] + dp[i-1][j - wt[i-1]],
      dp[i-1][j]
    )
else
    dp[i-1][j]
```

n → i
W → j
KP → dp

99%




```
for ( int i=1 ; i<=n ; i++)
< for ( int j=1 ; j<=w ; j++)
  {
    if ( wt[i-1] <= j)
      dp[i][j] = max
      ( pr[i-1] + dp[i-1][j- wt[i-1]] ,
        dp[i-1][j]

    else
        dp[i-1][j]
  }
}

return dp[n][w]
```

Top down Code