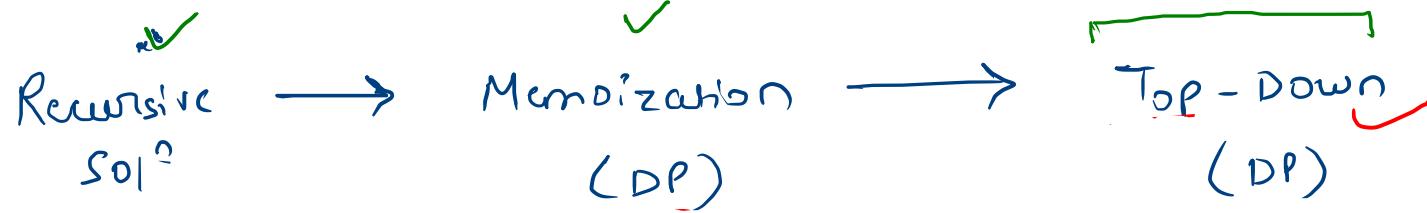


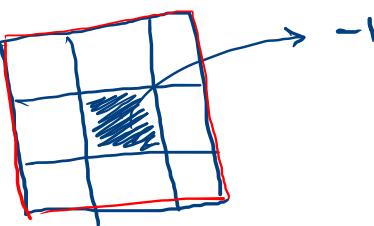
Dynamic
programming

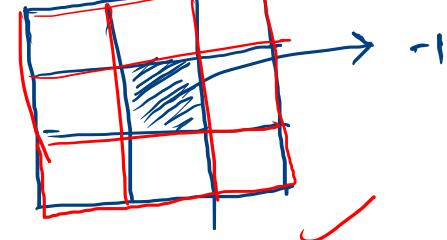
0/1 Knapsack Top-down



{ DP = Recursion
+ Storage ✓

Recursive solution → Base Condition + RC → for smaller input

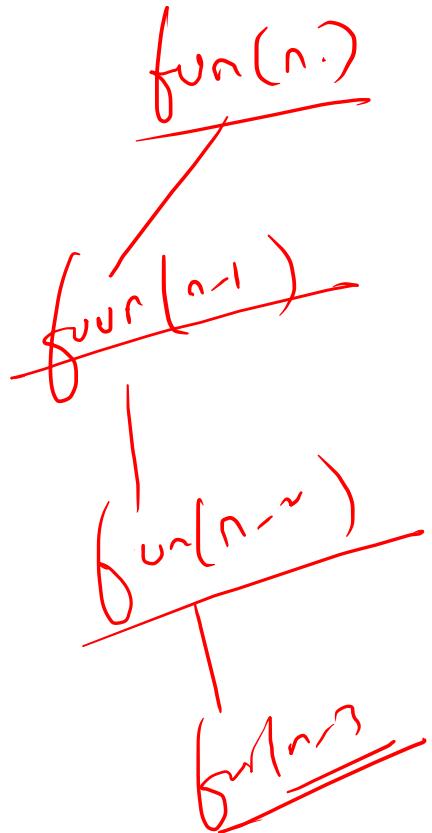
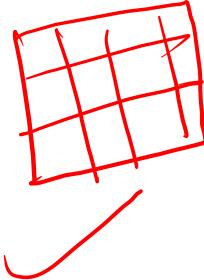
Memoization (DP) → RC + 

Top-down (DP) → 

Now,
why top-down
is better?

Top down

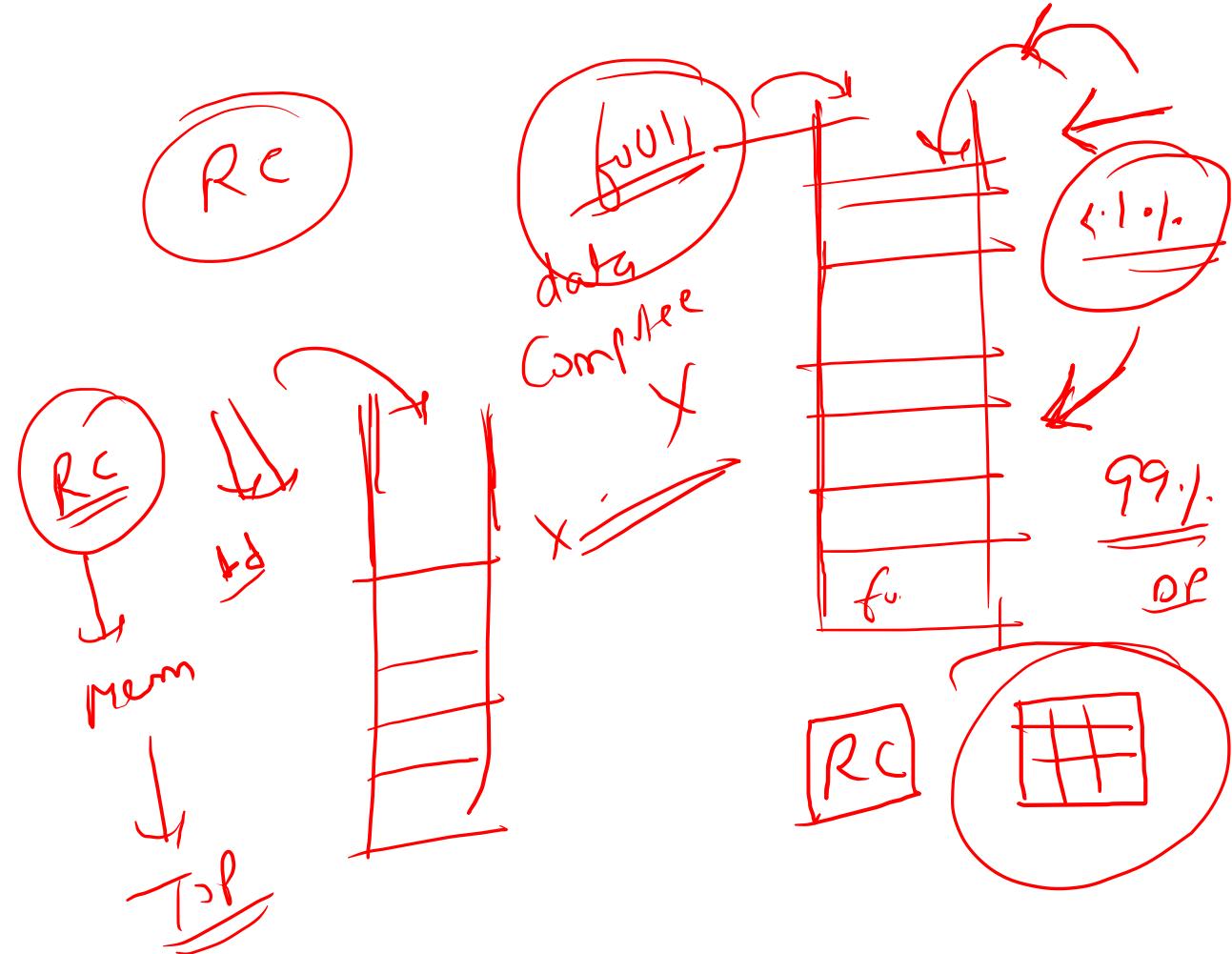
RC +



⇒ There is no use of recursion



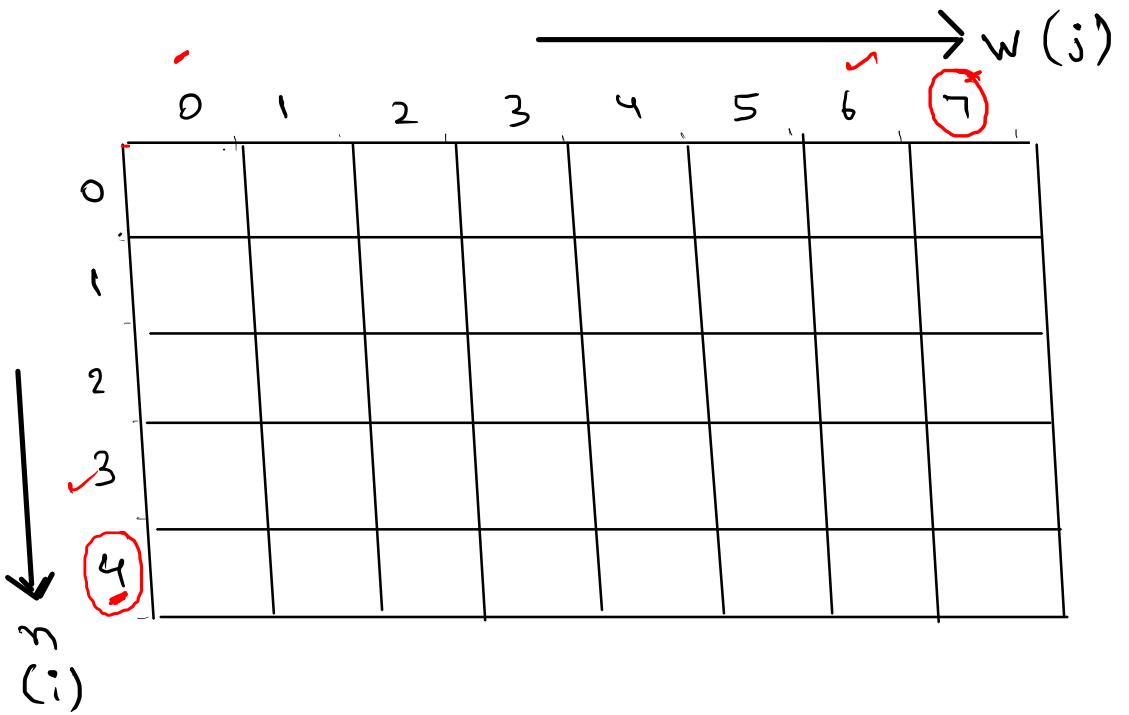
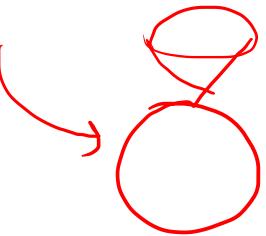
⇒ no risk, overlimit of function call stack is full



Now
understand
Top-down

✓ $\text{dp}[n+1][w+1]$
 $\text{dp}[4+1][7+1] = \boxed{\text{dp}[5][8]}$

Here |
 $w\in[] =$
 $p\in[] =$
 $w = 7$



Now understand
Top-down
~~dp~~

$dp[n+1][w+1]$

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

$\cancel{dp[s][8]}$

$\cancel{dp[j]}$
store

$\cancel{\max^n \text{ prof}}$

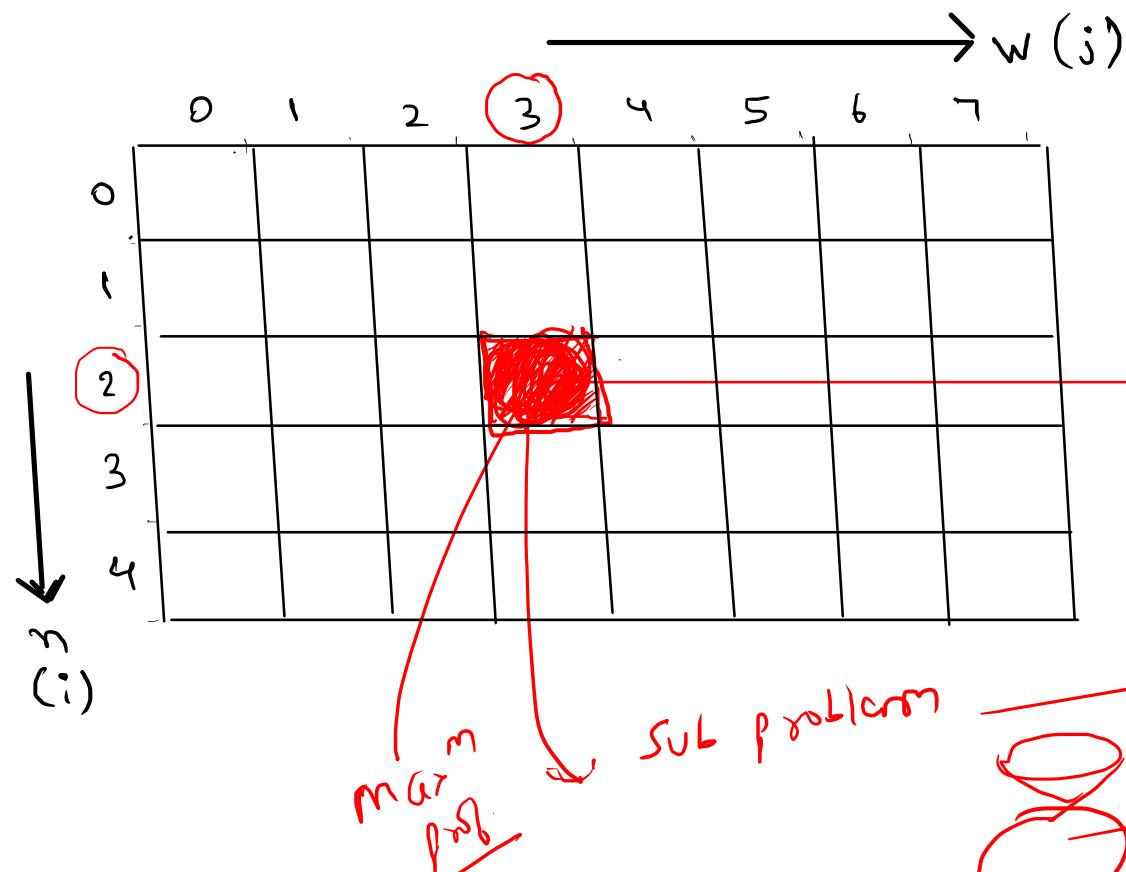
$wt[] =$

1	3	4	5
---	---	---	---

$pr[] =$

1	4	5	7
---	---	---	---

$w = 7$



$n=2$
 $w=3$

$wt[] =$
 $pr[] =$

1	3
1	4

$w=3$

Now
understand
Top-down

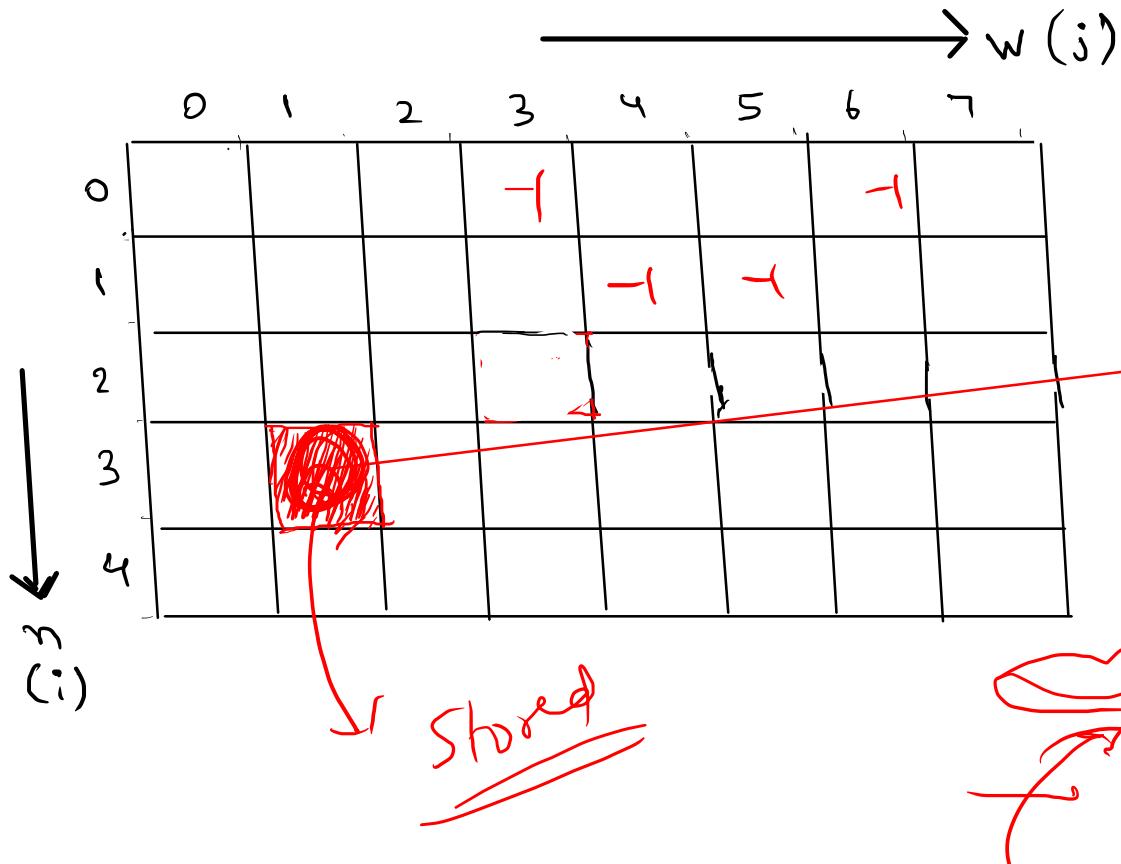
$$dp[n+1][w+1]$$

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

$$wt[] = \boxed{1 \ 3 \ 4 \ 5}$$

$$pr[] = \boxed{1 \ 4 \ 5 \ 7}$$

$$w = 7$$



→ understood the meaning of each block in matrix

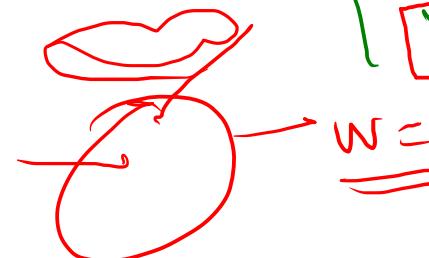
$$n = 3 \checkmark$$

$$w = 1 \checkmark$$

$$wt[] = \boxed{1 \ 3 \ 4}$$

$$pr[] = \boxed{1 \ 4 \ 5}$$

$$w = 1$$



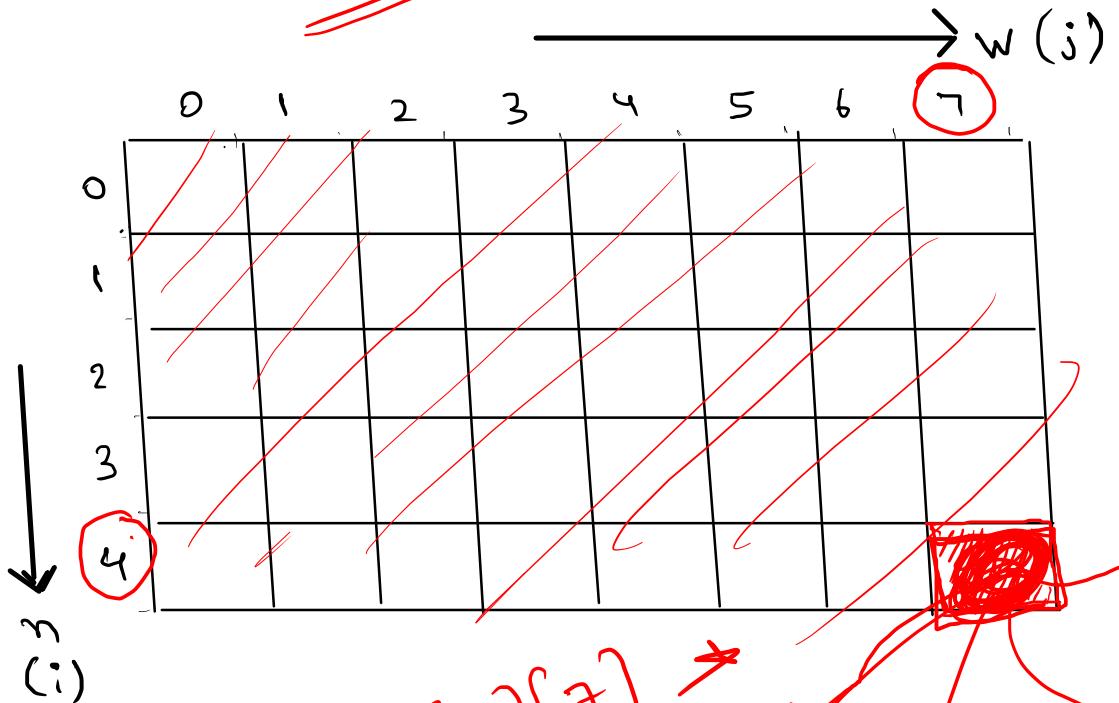
Now understand
Top-down

Ans

$dp[n+1][w+1]$

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

n, w



return $dp[4][7]$

Ans

stored

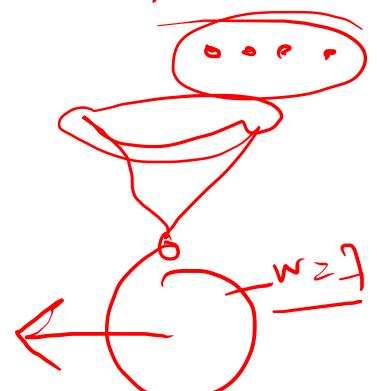
Answer

$$wt[] = \boxed{1 \ 3 \ 4 \ 5}$$

$$pr[] = \boxed{1 \ 4 \ 5 \ 7}$$

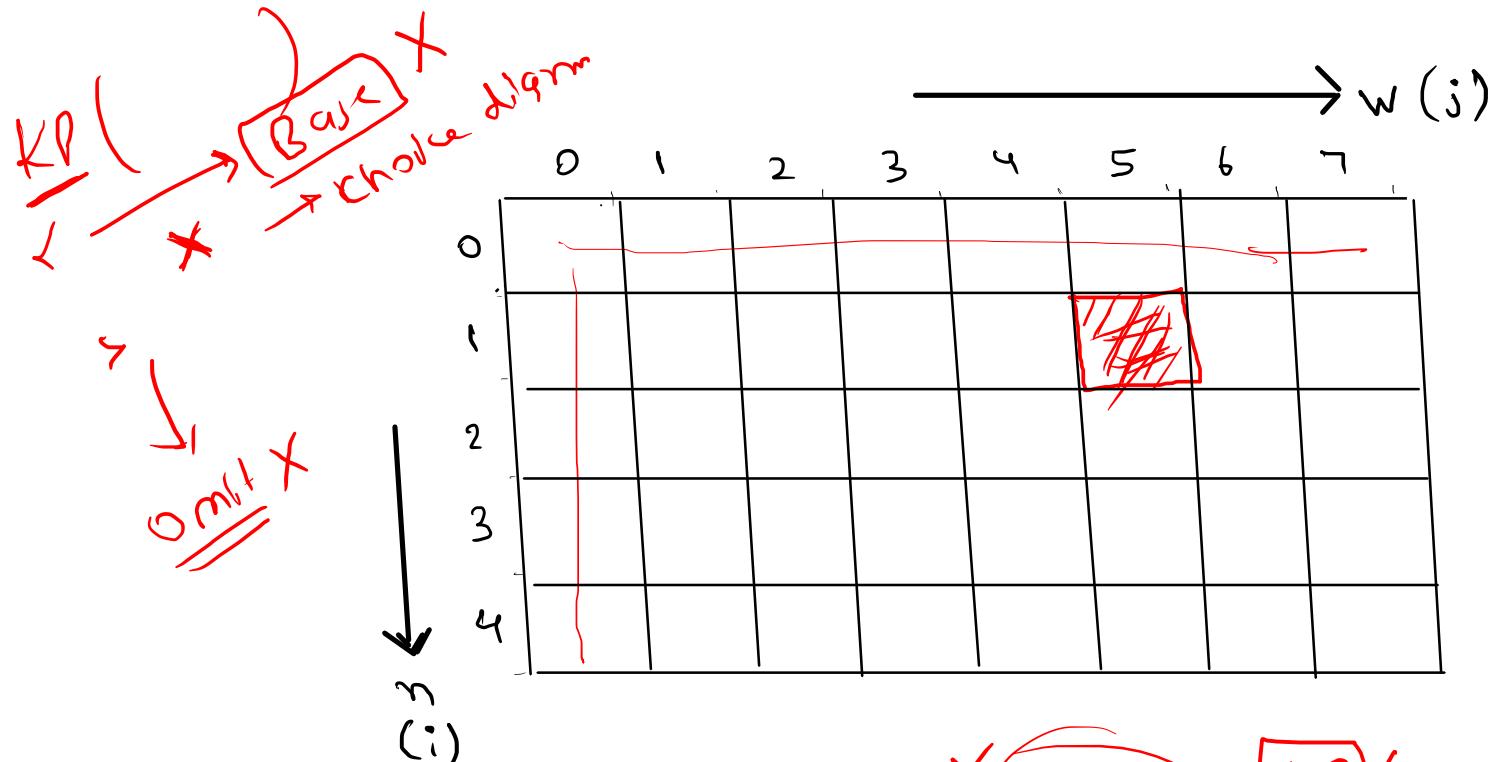
$$w = 7$$

understood the
meaning of each block
in matrix



Now understand
Top-down

$$\begin{aligned} dp[n+1][w+1] \\ dp[4+1][7+1] = dp[5][8] \end{aligned}$$



X $n=1$
 $w=8$ KP

$$\begin{aligned} wt[] &= [1 \ 3 \ 4 \ 5] \\ pr[] &= [1 \ 4 \ 5 \ 7] \\ w &= 7 \end{aligned}$$

- 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

Base condition

if $(\underline{n == 0}) \text{ || } (\underline{w == 0})$
return $\underline{\underline{0}}$, -max profit

Diagram illustrating a sparse matrix representation:

	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0
1	0						
2	0						
3	0						
4	0						

Annotations:

- A red arrow points to the first row (0) with a red circle at index 1.
- A red arrow points to the first column (:) with a red circle at index 0.
- A red arrow points to the last column (:) with a red checkmark.
- A red bracket above the last three columns (4, 5, 6, 7) is labeled $w(j)$.
- A red arrow points to the first row (0) with a red circle at index 1.

Base Condition
0/1 knapsack

if $((n==0) \text{ || } (w==0))$
return 0;

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

✓ → $w(j)$

↓ i

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0
1	0	•	•	•	✓	✓	✓
2	0	✓					
3	0						
4	0						Q

Ans

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

for(int i=0 ; i<=w ; i++)
dp[0][i] = 0;

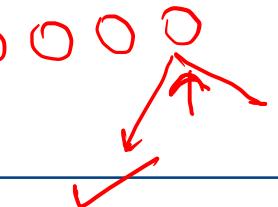
initialization



base condition

$KP(wt[], pr[], n)$

Recursive
code



if ($wt[n-1] \leq w$)

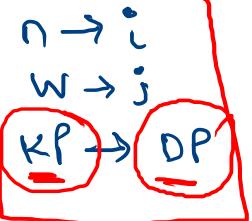
$$dp[n][w] = \max$$

$$\left(pr[n-1] + KP(wt[], pr[], w - wt[n-1], n-1) \right)$$

else

$$KP(wt[], pr[], w, n-1)$$

Top down
Code



if ($wt[i-1] \leq j$)

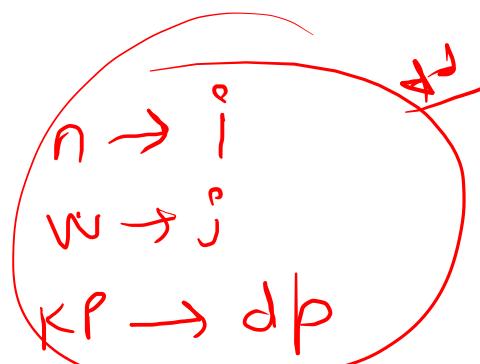
$$dp[i][j] = \max$$

$$\left(pr[i-1] + dp[i-1][j - wt[i-1]] \right)$$

$$dp[i-1][j]$$

else

$$dp[i-1][j] =$$



99.1.

		0	1	2	3	4	5	6	7	$w(j)$
0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	0	

```

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