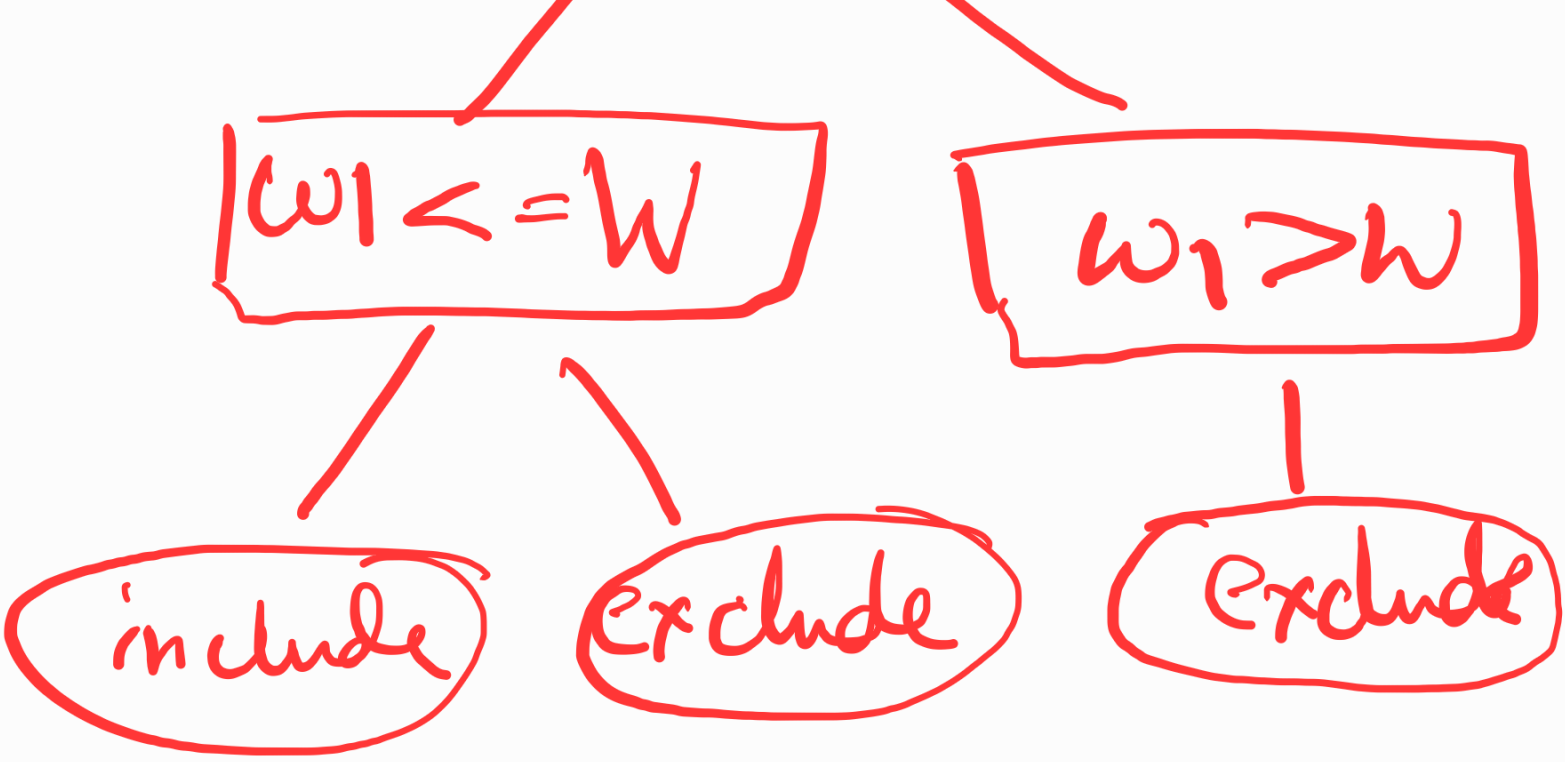
I/P:

w[]: | 1 | 3 | 4 | 5 |

val[]: | 1 | 4 | 5 | 5 |

W: 7

Choice diagram:



max profit

```
int knapsack(int[] wt,
int val[], int W, int n)
{
```

Base Cond^n

Choice Diagram

```
}
```

Think of smallest valid input

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

```
int knapsack(int[] wt,
             int[] val,
             int W,
             int n)
```

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

↓

Base Condⁿ

```
if( wt[n-1]<=W)

include = val[n-1]
          +
          knapsack(
              wt,
              val,
```

W - wt[n-1],

)^{n-1}

exclude = knapsack(
wt,
val,
W,
)^{n-1}

return max(include, exclude);

else if (wt[n-1] > W)

```
return exclude;
}
```

Memoization

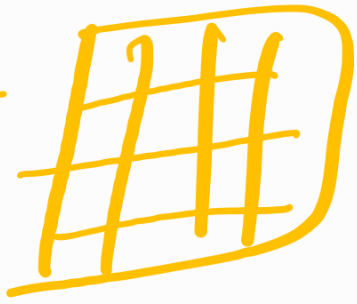
```
int [][] dp;
int knapsack(wt[], val[], W, n)
{
```

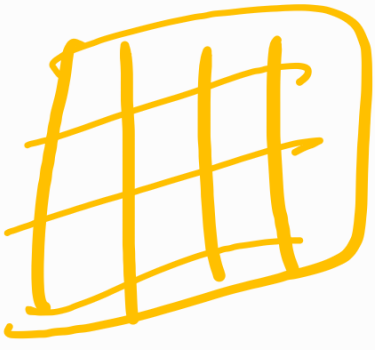
```
{ if(n==0 ||
      W==0)
   return 0;
if(dp[n][W]!=-1)
   return dp[n][W];
if(wt[n-1]<=W)
   return
dp[n][W] = max(
      include,
```

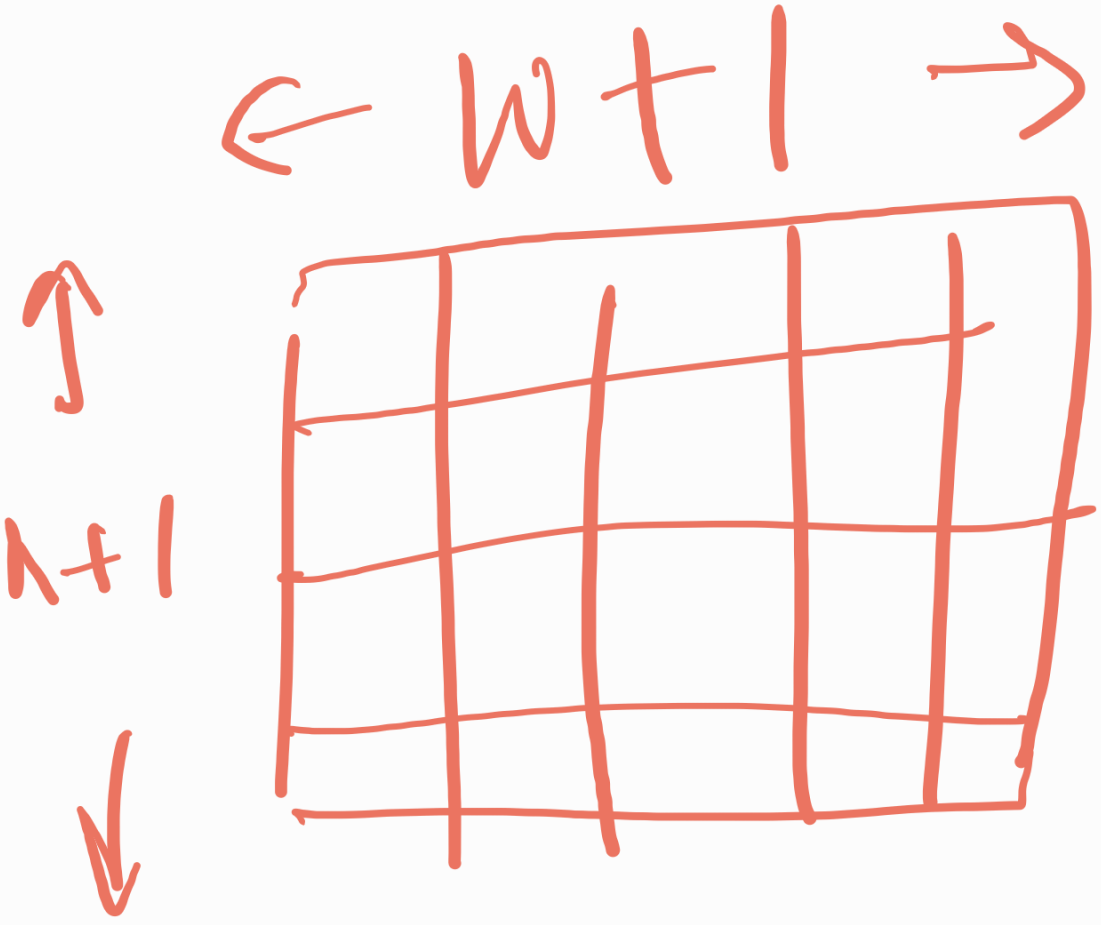
(exclude)

```
else if (wt[n-1] > W)
    return
    dp[n][W] = knapsack
                (wt, val, W, n-1);
}.
```

Top Down Approach

Memorized → RC + 

Top down →  no recursion

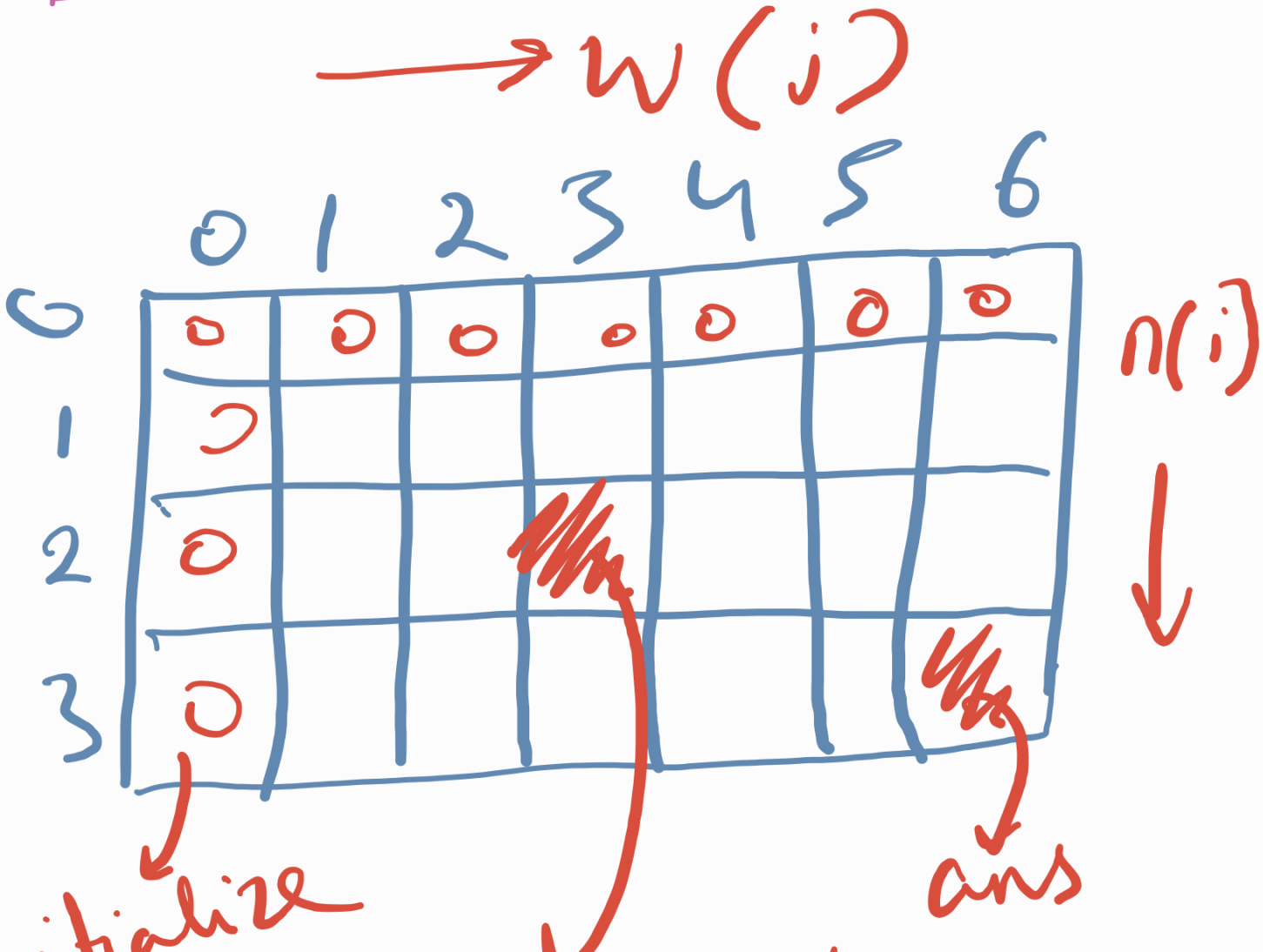


int[][] dp

= new int[n+1][W+1].

Step 1 : initialize

Step 2 : Recursive
& iterative



Knapsack problem
Solution

```
int knapsack(wt[],
             val[],
             W, n)
{
  // initialize
  for(i=0 to n){
    for(j=0 to W){
      if(i==0 ||
```

```
        j == 0 )
    dp[i][j] = 0
  }
}
// convert Recursive
      to iterative
for (i = 1 to n) {
  for (j = 1 to w) {
```

```
if (wt[i-1] <= j)
{
dp[i][j] = max(
val[i-1] +
dp[i-1][j - wt[i-1]],
dp[i-1][j]);
} else
```

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

}

return dp[n][w];

}