DP-2

→ Recursive - soln

→ TD - Dp

→ Bup - Dp

✓ Number of ways problems

-364:54:9 Edit

Description

Sandhya is running up a staircase with N steps, and can hop(jump) either 1 step, 2 steps or 3 steps at a time. You have to count, how many possible ways Sandhya can run up to the stairs.

Input

input Format

Input contains integer N that is number of steps

Constraints

N <= 30

$n = 30 \Rightarrow 2^{n} = 2^{30}$

Output

Output Format

Output for each integer N the no of possible ways w.

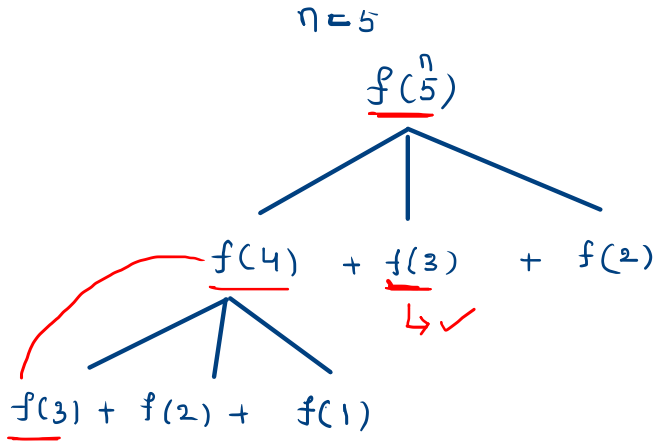
Sample Input 1

```
4
```

Sample Output 1

```
7
```

```
function fun(n)
{
     if(n<0)
          return 0;
     if(n==0)
         return 1;
     return fun(n-1)+fun(n-2)+fun(n-3);
}
```



DP :-
overlapping sub-prob

TopDown DP:-

```
arr[n+1]={-1}// initilize all values with the -1
function fun(n)
{
     if(n<0)
          return 0;
     if(n==0)
         return 1;
     if(arr[n]!=-1)
         return arr[n]
     else
       return arr[n]=fun(n-1)+fun(n-2)+fun(n-3);
}
```

Bottum Up DP:-

```
function fun(n)
{
     if(n<0)
          return 0;
     if(n==0)
         return 1;
     let arr[n+1]
     arr[0]=1, arr[1]=1,arr[2]

    for(i=3;i<=n;i++)
      arr[i]=arr[i-1]+arr[i-2]+arr[i-3]
    return arr[n]
}
```

→ optimal ss DP ✓

↳ Recursive code (Recursion + overlapping subproblems)

↳ Top-down DP (Recursion + Memoization)

↳ Bottoum up DP (Loops + Memoization)

* 0/1 KS problem [Greedy fails]

	O_1	O_2	O_3
P_i	10	12	28
w_i	1	2	4

$w = 6$

$12 + 28 = 40$

$\Rightarrow$ O_1 O_2 O_3 — 0 → don't take, 1 → take

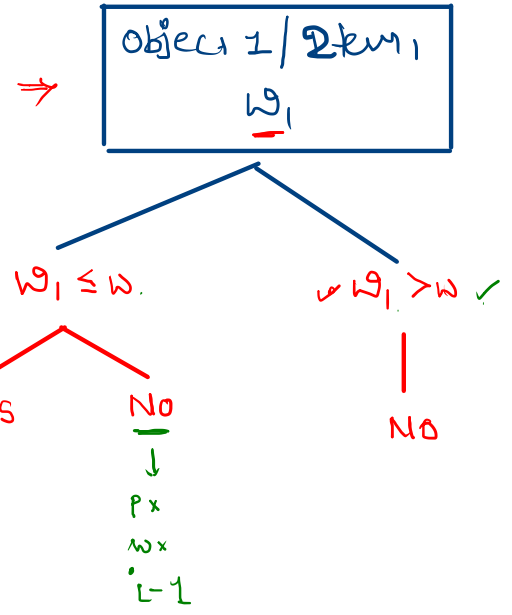
0/1 0/2 0/1

2 2 2 $\Rightarrow 2 \times 2 \times 2 = 2^3$

$\Rightarrow O(2^n)$ ↳ exponential

$O(n^k)$ $k \gg 1$

* optimal sub-structure:



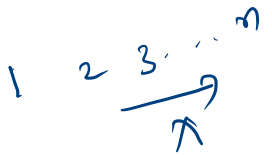
$$ks(i, w) = \begin{cases} 0 & ; i = 0 \,||\, w = 0 \\ ks(i-1, w) & ; w_i > w \\ \max\left(P_i + ks(i-1, w-w_i),\ ks(i-1, w)\right) & ; w_i \le w \end{cases}$$

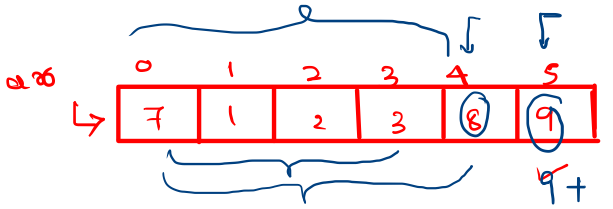
i = # of objects; w = weight of KS

$P_i + ks(i-1, w-w_i)$: include

$ks(i-1, w)$: don't want to include

```
function ks(p[],wt[],w,n)
{
       if(n==0 || w==0)
            return 0
       if(wt[n-1]>w)
           return ks(p,wt,w,n-1)
      else
           return max( p[n-1]+ks(p,wt,w-wt[n-1],n-1), ks(p,wt,w,n-1) )
}
```





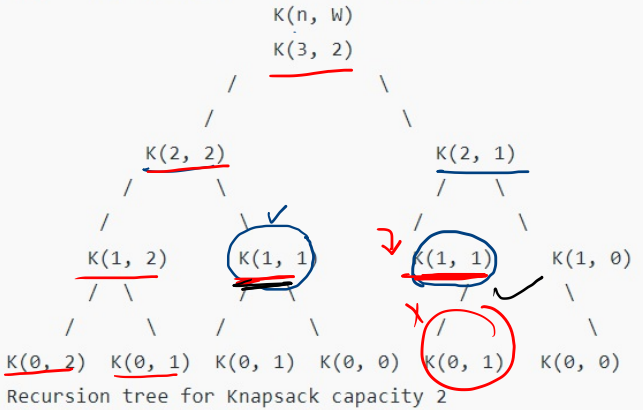
9+8+

```
fun(arr[], n)
{
    if(n == 1)
        return arr[0]

    else
        return arr[n-1]+
               fun(arr, n-1)

}
```

In the following recursion tree, K() refers to knapSack(). The two parameters indicated in the following recursion tree are n and W.
The recursion tree is for following sample inputs.
wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}



Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.

$n \rightarrow (n+1) \rightarrow LD$

⇒ Overlapping sub-problems

DP → Memoization
↳ create space

$n = 3$

	O_1	O_2	O_3
P_i	10	12	28
W_i	1	2	4

$W = 6$ ✓

$\Rightarrow (n+1, W+1)$

4×7

Col. represent weight & KS

$\Rightarrow$ Cells, We will fill profit

Rows rep. # of objects

$\Rightarrow$

W →	0	1	2	3	4	5	6
→ 0	0	0	0	0	0	0	0
✓ 1	0						
✓ 2	0						
✓ 3	0						

4×7

Contains final answer

```
Top Down DP 0-1 KS
dp[n+1][w+1]={-1}
function ks(p[],wt[],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]=ks(p,wt,w,n-1)
      else
           return dp[n][w]=max( p[n-1]+ks(p,wt,w-wt[n-1],n-1), ks(p,wt,w,n-1) )
}
```

Bottom up DP

row (n) ⇒ i

Col (w) ⇒ j

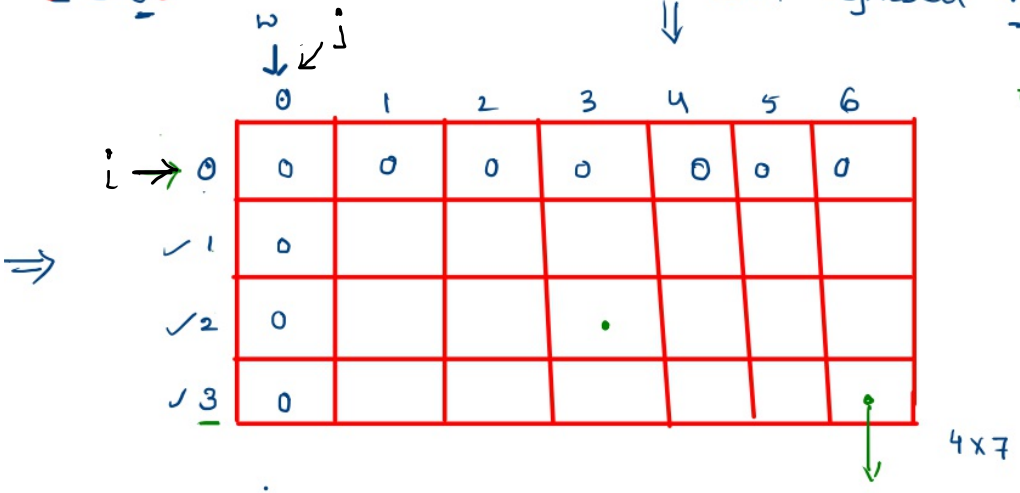
⇒ (n+1, w+1)

4 x 7

W = 6 ✓

Col. represent weight & KS

⇒ cells, We will fill profit



Contains final answer

p[],

wt[],

n, w

Bottum Up DP 0-1 KS

```
function ks(p[],wt[],w,n)
{
      dp[n+1][w+1];

      for(i=0;i<=w;i++)// 1st row all zeros
         dp[0][i]=0
      for(i=0;i<=n;i++)
         dp[i][0]=0

      for(i=1;i<=n;i++)
      {
           for(j=1;j<=w;j++)
           {
               if(wt[i-1]>j)
                   dp[i][j]=dp[i-1][j]
               else
                  dp[i][j]=max( p[i-1]+ dp[i-1][w-wt[i-1]], dp[i-1][j] )
           }
      }

     return dp[n][w]
}
```

→ O(w)

→ O(n)

→ O(n)

→ O(w)

} O(1)

} O(1)

T.C :-

n → i
w → j
} replace

⇒ w + n + n x w ⇒ O(n x w)

S.C.

O(n x w)