SCALER

Dynamic Programming 1

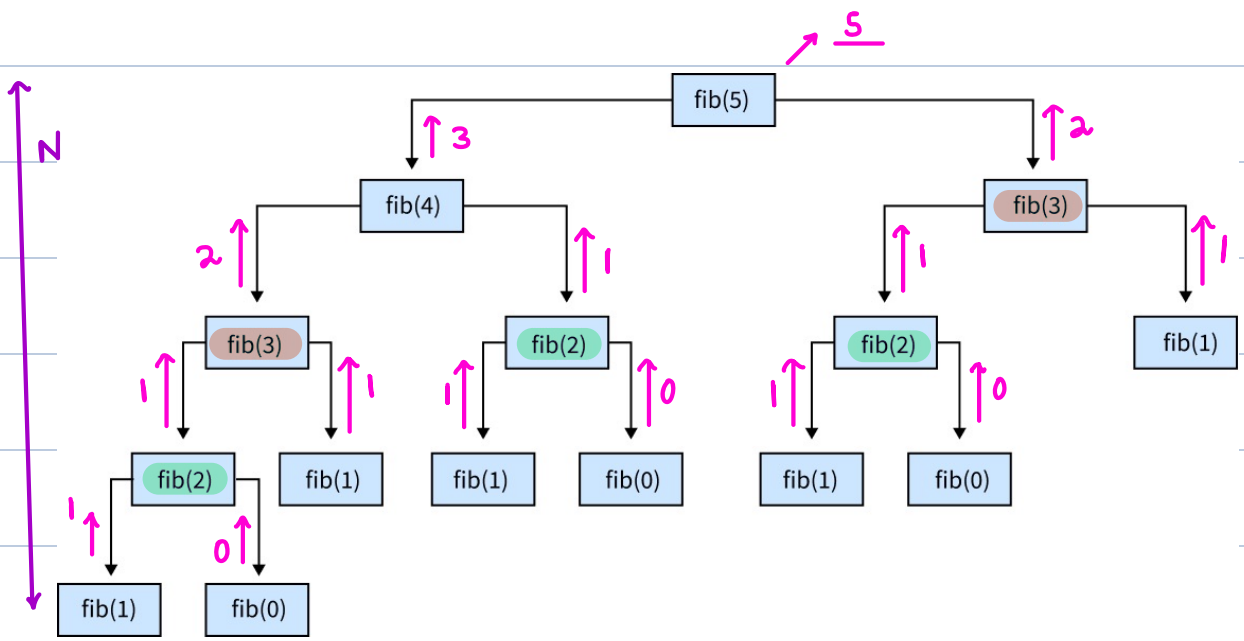
TABLE OF CONTENTS

Nth Fibonacci Number

N →	0	1	2	3	4	5	6	7	8	9	10
fib(N) →	0	1	1	2	3	5	8	13	21	34	55

```
int fib(N) {
    if (N <= 1) return N
    return fib(N-1) + fib(N-2)
}
```

TC = $O(2^N)$ SC = $O(N)$



1. Optimal Structure → A problem can be solved by dividing it into smaller subproblems.

2. Overlapping Subproblems → Same subproblems are calculated multiple times.

store & reuse → Dynamic Programming

```
// ∀i, F[i] = -1
int fib(N) {
    if (N <= 1) return N
    if (F[N] != -1) return F[N]
    F[N] = fib(N-1) + fib(N-2)
    return F[N]
}
```

TC = O(N) SC = O(N + N) = O(N)

$2^N \longrightarrow N$ (using DP)

Top - down Approach → Memoization in recursive solutions.

Start with big problem, break it to smaller subproblems & recursively calculate answer.

→ Easy to understand & implement.

✓ **Bottom - up Approach** → Iterative approach.

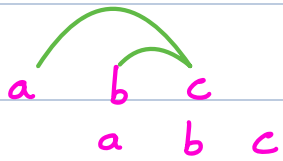
Start with base case & iteratively calculate ans for bigger problem.

→ No recursion space ⇒ potential to optimize SC in some cases.

```
F[0] = 0      F[1] = 1
for i → 2 to N {
        F[i] = F[i-1] + F[i-2]
}
return F[N]
```

TC = O(N) SC = O(N)

Further S.C Optimisation




```
a = 0      b = 1
for i → 2 to N {
    c = a + b
    a = b
    b = c
}
return c
```

TC = O(N) SC = O(1)

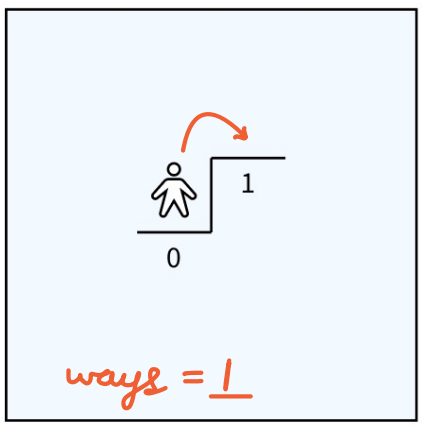
Climbing Stairs

$1 \le N \le 10^5$

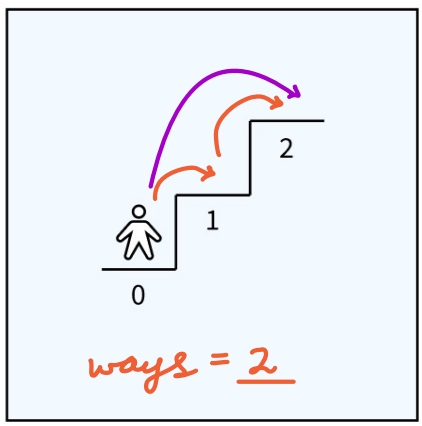
Calculate the number of ways to reach Nth stair.

You can take 1 step at a time or 2 steps at a time.

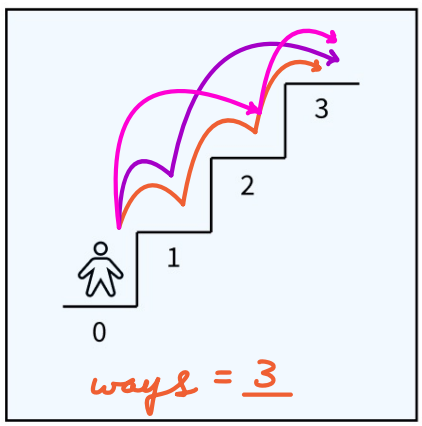
N = 1



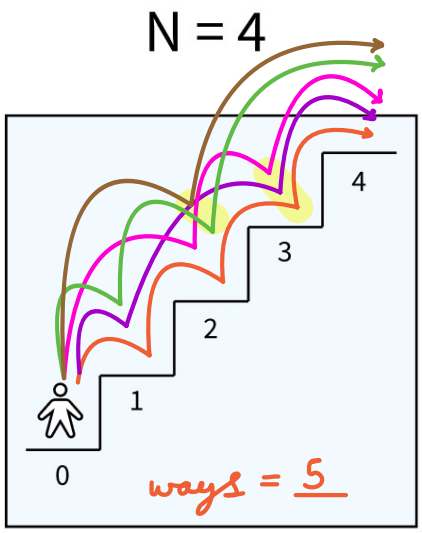
N = 2



N = 3

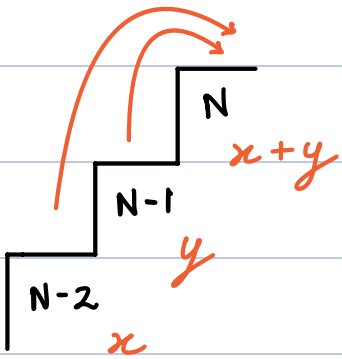


N = 4



ways(0) = 1

way to do a task = 0
⇒ impossible



ways (N) = ways (N-1) + ways (N-2)

ways (0) = ways (1) = 1

code → same as fibonacci number.

< Question > : Find the minimum number of perfect squares required to get sum = N?

[numbers can repeat]

N = 6 $1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2$

$2^2 + 1^2 + 1^2$ Ans = 3

N = 10 $1^2 + 1^2 + \ldots$ 10 times | $2^2 + 2^2 + 1^2 + 1^2$

$2^2 + 1^2 + 1^2 + \ldots$ 6 times | $3^2 + 1^2$ Ans = 2

N = 9 3^2 Ans = 1

N = 5 $2^2 + 1^2$ Ans = 2

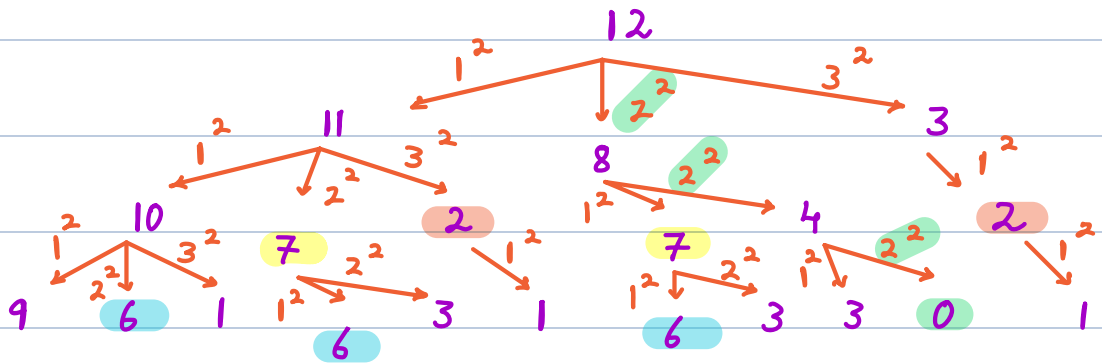
→ use large perfect sq. to reduce count. (greedy)

X

N = 50 $7^2 + 1^2$ Ans = 2

N = 12 $3^2 + 1^2 + 1^2 + 1^2$ X

$2^2 + 2^2 + 2^2$ Ans = 3



optimal substructure ✓
overlapping subproblems ✓
} use DP

$crt(12) = min(crt(12-1^2), crt(12-2^2), crt(12-3^2)) + 1$

$crt(N) = \forall x \; min(crt(N-x^2)) + 1$

$1 <= x^2 <= N$

$\forall i, crt[i] = i$ // $1^2+1^2+\ldots$ i times ($crt[0] = 0$)

```
for i → 1 to N {
    for x → 1 to √i {   // 1 <= x^2 <= i   x → 1, 2
        crt[i] = min(crt[i], crt[i - x^2] + 1)
                                  5 - 2^2   +1
    }
} return crt[N]
```

$TC = O(N\sqrt{N})$ $SC = O(N)$

N = 5

0	1	2	3	4	5
0	1	2	3	~~4~~ 1	~~5~~ 2

DP 2: Two Dimensional

Q → Given an integer array.

A[i] → Money

Find max sum without selecting adjacent elements.

$i-1 \leftarrow i \rightarrow i+1$

A → | 1 | 2 ✓ | 5 | 8 ✓ | 3 |

Ans = 10

A → | 2 ✓ | 7 | 9 ✓ | 3 | 1 ✓ |

Ans = 12

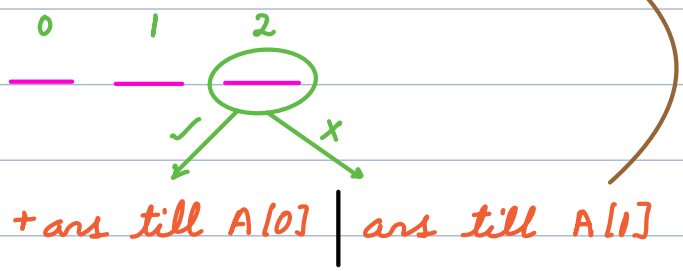
A → | 7 ✓ | 4 | 1 | 3 ✓ |

Ans = 10

N = 1 Ans = A[0]

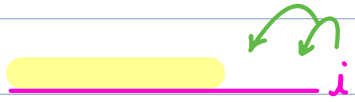
N = 2 Ans = max (A[0], A[1])

N = 3



sum[N] = max (A[N] + sum[N-2] ✓ , sum[N-1] x)

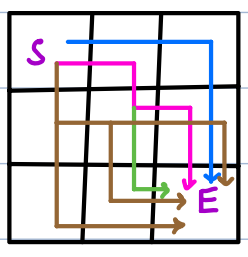
```
sum[0] = A[0]
sum[1] = max(A[0], A[1])
for i → 2 to (N-1) {
|     sum[i] = max( A[i] + sum[i-2] , sum[i-1])
}
return sum[N-1]
```



TC = O(N) SC = O(N)

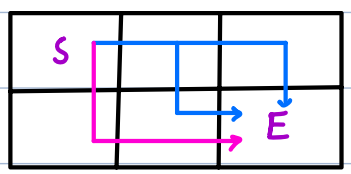
O(1)

Q → Find # ways to travel from (0,0) to (N-1, M-1) s.t in one step we can go → or ↓.

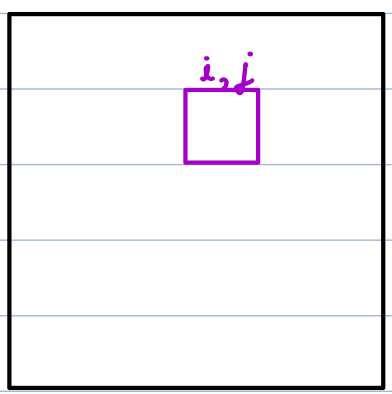


3 * 3

Ans = 6



Ans = 3



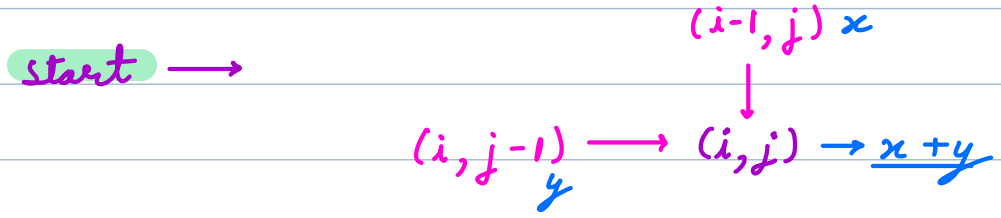
start → (i, j) → end

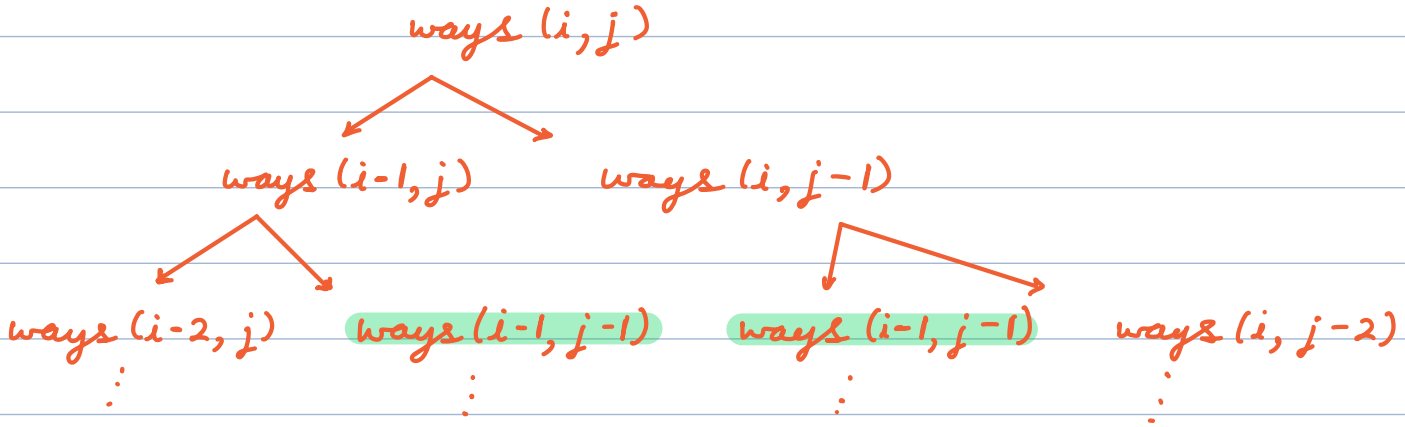
∀ i, j find # ways from start to i, j

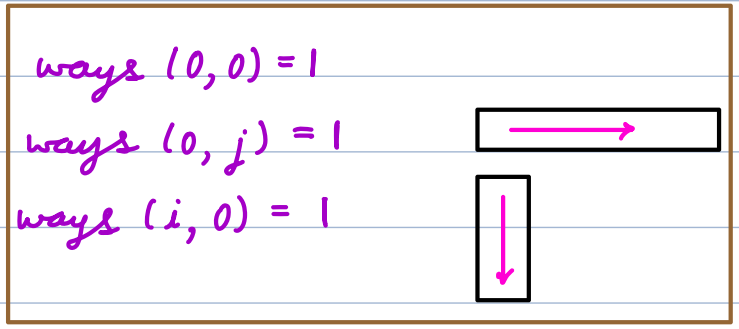
⇒ Ans = (i, j) → End ✓

∀ i, j find # ways from i, j to end

⇒ Ans = (i, j) → start (H.W)







optimal substructure ✓

overlapping subproblems ✓

use DP.

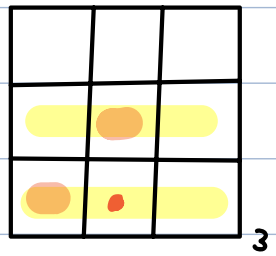
Bottom – Up

```
for i → 0 to (N-1) {
    for j → 0 to (M-1) {
        if (i == 0 || j == 0)  ways[i][j] = 1
        else   ways[i][j] = ways[i-1][j] + ways[i][j-1]
    }
} return ways[N-1][M-1]
```

TC = O(N * M)

SC = O(N * M)

$O(2M) \rightarrow O(M)$




```
∀j, pre[j] = 1      // row 0
for i → 1 to (N-1) {
    for j → 0 to (M-1) {
        if (j == 0)     cur[j] = 1
        else            cur[j] = pre[j] + cur[j-1]
    }
    for j → 0 to (M-1) {
        pre[j] = cur[j]
    }
} return cur[M-1]
```

TC = $O(N * M)$

SC = $O(M)$

Q → Find the count of N digit numbers with digit sum = S.

N, S > 0

[0 — 9]

N = 2

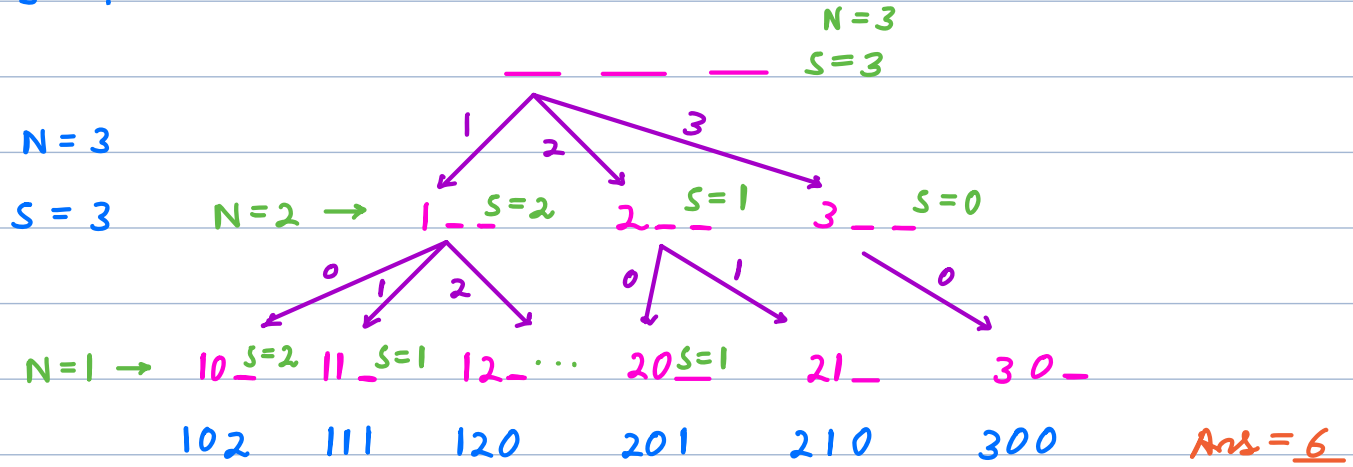
S = 4

13 22 31 40

Ans = 4

N = 3

S = 3

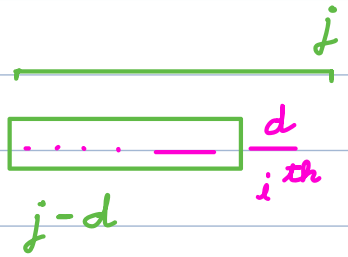


use DP ← { optimal substructure ✓, overlapping subproblems ✓

count[i][j] → count of i digit nos. with digit sum j.

count[1][j] → 1 if 1 <= j <= 9
→ 0 else

```
for j → 1 to 9 {
    count[1][j] = 1          ✓
}
for i → 2 to N {
    for j → 0 to S {
        for d → 0 to 9 {
            if (d <= j) {
                count[i][j] += count[i-1][j-d]
                // count[i][j] = (count[i][j] + count[i-1][j-d]) % M
            }
        }
    }
}
return count[N][S]
```



TC = O(N*S)

SC = O(N*S)

↳ optimize SC → H.W

i \ j	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0	1	2	3
3	0	1	3	6 → Ans

Catalan Numbers

The Catalan numbers form a sequence of natural numbers that have numerous applications in combinatorial mathematics. Each number in the sequence is a solution to a variety of counting problems. The Nth Catalan number, denoted as Cn, can be used to determine:

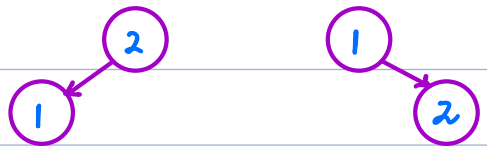
- The number of correct combinations of N pairs of parentheses.
- The number of distinct binary search trees with N nodes, etc.

Q → Find the count of unique BST with N distinct nodes.

N = 1

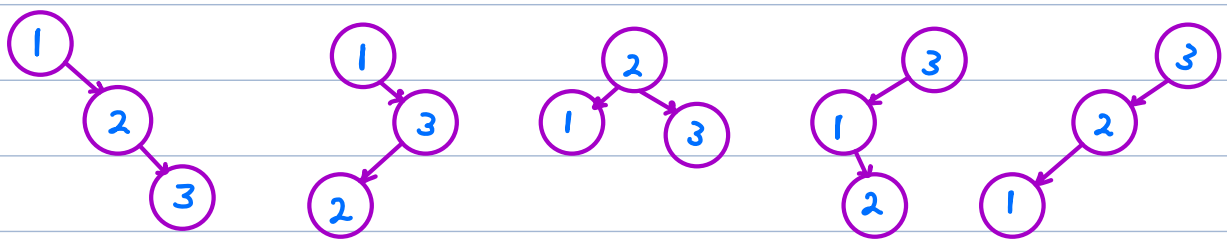
Ans = 1

N = 2



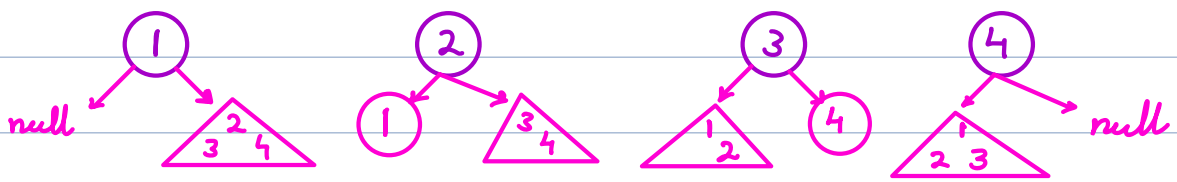
Ans = 2

N = 3



Ans = 5

N = 4



1

c(0) * c(3) + c(1) * c(2) + c(2) * c(1) + c(3) * c(0)

c(4) = 1 * 5 + 1 * 2 + 2 * 1 + 5 * 1 = 14

$$c(N) = \sum_{i=0}^{N-1} c(i) * c(N-1-i)$$

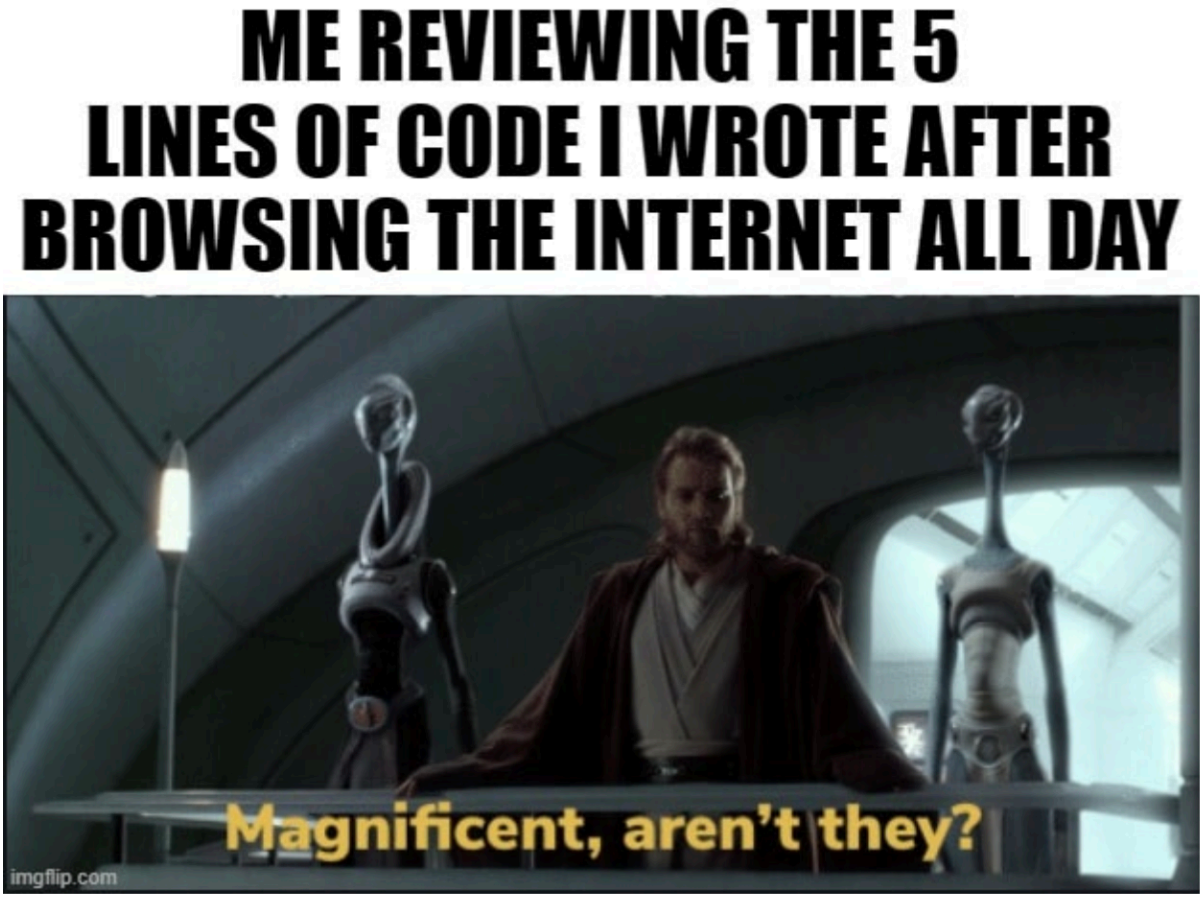
$$\frac{^{2N}C_N}{(N+1)}$$

TC = O(N²) SC = O(N)

```
C[0] = C[1] = 1
for i → 2 to N {
    for j → 0 to (i-1) {
        C[i] += C[j] * C[i-1-j]
    }
} return C[N]
```

DP 3 - KnapSack

TABLE OF CONTENTS



Q → Given an integer array & a target sum K.

Check if there exists a subset sum = K. (A[i] > 0)

subset → may / may not be continuous

	0	1	2	3	4
A =	3	4	11	5	2

K = 8

A[0] + A[3] = 8

	0	1	2	3	4
A =	1	2	1	3	5

K = 4

A[0] + A[1] + A[2] = 4

Bruteforce → check sum for every subset.

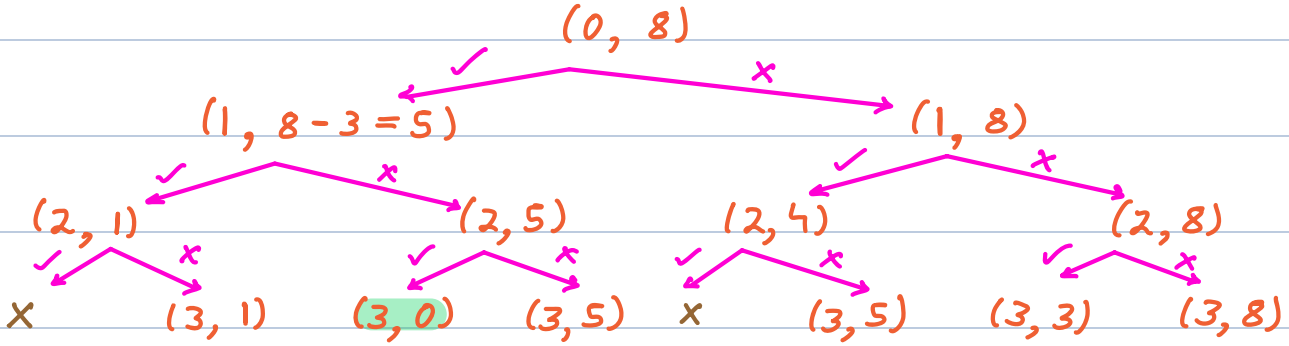
TC = $O(2^n)$

∀ A[i] → select / reject

	0	1	2	3
A =	3	4	5	2

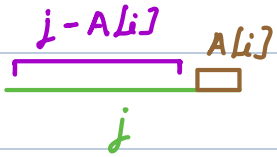
K = 8 Ans = true

(index, K)



optimal substructure
overlapping subproblems } use DP

dp[i][j] → check if subset sum = j considering elements from 0 to i.



dp[i][j] ✓ → dp[i-1][j - A[i]] OR
dp[i-1][j]
✗

∀i, dp[i][0] = true (empty subset)

for i → 0 to (N-1) 0—(N-1), 0—K
dp[i][0] = true
if (K >= A[0]) dp[0][A[0]] = true
for i → 1 to (N-1) {
  for j → 1 to K {
    if (j >= A[i]) dp[i][j] = dp[i-1][j - A[i]] || dp[i-1][j]
    (select) (reject)
    else dp[i][j] = dp[i-1][j]
  }
}
return dp[N-1][K]

TC = O(N * K) SC = O(N * K)
O(2K) = O(K) ✓

KnapSack Problem

profit / loss

Given N objects with their values Vi and their weights Wi. (cost)

A bag with capacity W that can be used to carry some objects such that →

budget

total sum of objects weights ≤ W , and

sum of values in the bag is maximised.

Fractional KnapSack

Given N cakes with happiness and weight.

Find max total happiness that can be kept in a bag with capacity = W

(cakes can be divided)

N = 5 happiness[ ] → [3 8 10 2 5] sum = 23

W = 40 weight[ ] → [10 4 20 8 15]

20 - 4 = 16 40 - 20 = 20 16 - 15 = 1

→ use bag completely

→ How to select ⇒ h[i] / w[i]

N = 5

W = 40

	0	1	2	3	4
happiness[ ] → [	3	8	10	2	5]
weight[ ] → [	10	4	20	8	15]

h/w → 0.3 | 2 | 0.5 | 0.25 | 0.33

parts → 10 | 4 | 20 | 8 | 15

4 | 1 | 2 | | 3

Sum = 2 * 4 + 0.5 * 20 + 0.33 * 15 + 0.3 * 1

= 8 + 10 + 5 + 0.3 = 23.3 (Ans)

Sol → Select items in descending order wrt h[i]/w[i] till complete capacity is used. (Greedy)

TC = O(N log(N)) SC = O(N)

Practical Scenario

Flipkart is planning a special promotional event where they need to create an exclusive combo offer. The goal is to create combination of individual items that together offer the highest possible level of customer satisfaction (indicating its popularity and customer ratings) while ensuring the total cost of the item in the combo doesn't exceed a predefined.

0 - 1 KnapSack

Given N toys with their happiness and weight.

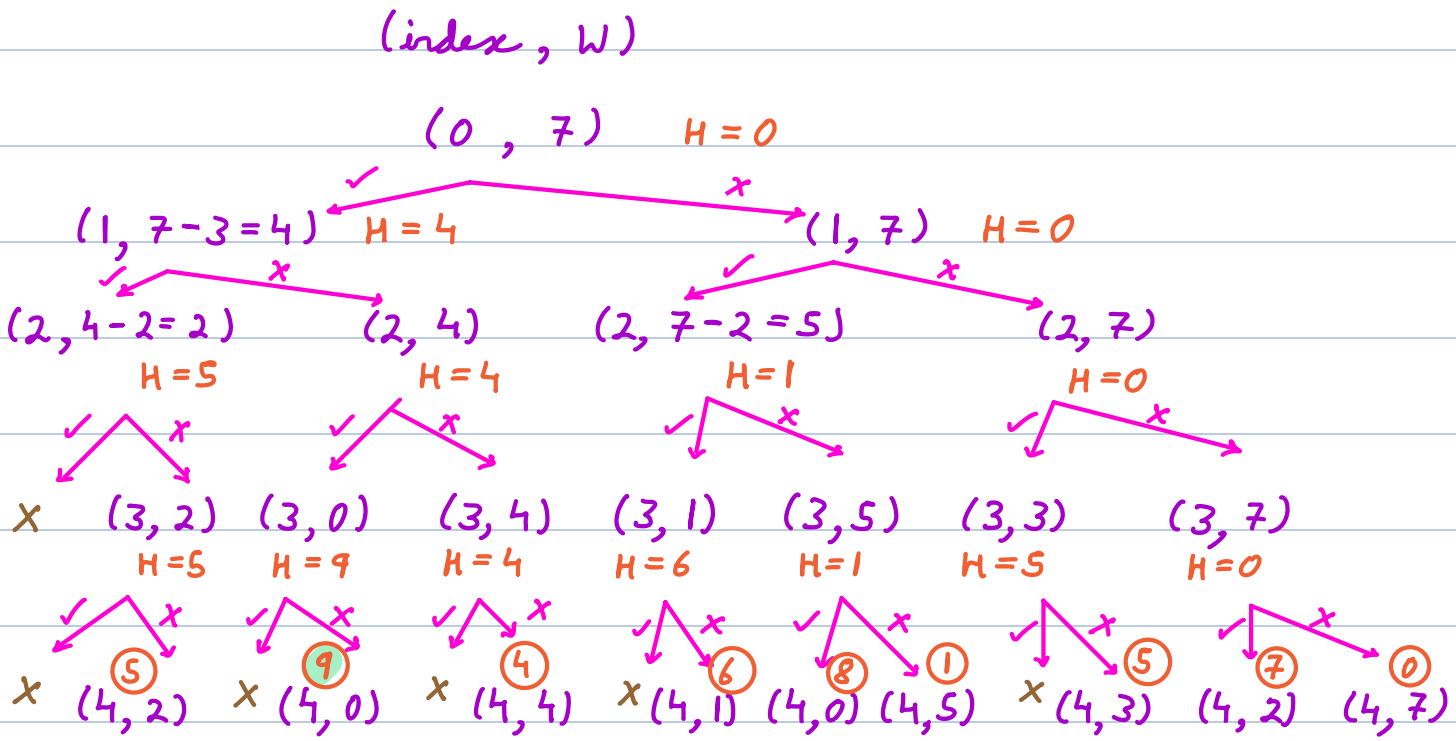
Find max total happiness that can be kept in a bag with capacity W.

[Toys can't be divided]

N = 4

W = 7

	0	1	2	3
happiness[ ] →	4	1	5	7
weight[ ] →	3	2	4	5



dp[i][j] → max total happiness considering elements till index i & capacity j.

dp[i][j] ✓ → h[i] + dp[i-1][j-w[i]]

dp[i][j] x → dp[i-1][j]

∀ i, dp[i][0] = 0 (no capacity)

∀ j, dp[0][j] = 0 (nothing to buy)

```
// 1-based index
∀ i, dp[i][0] = 0
∀ j, dp[0][j] = 0
for i → 1 to N {
    for j → 1 to W {
        if (j >= w[i])
            dp[i][j] = max(h[i] + dp[i-1][j-w[i]], dp[i-1][j])
        else dp[i][j] = dp[i-1][j]
    }
} return dp[N][W]
```

TC = O(N * W) SC = O(N * W)

O(2W) = O(W)

Bottom - up

0 1 2 3 4

W = 8

N = 5

weight[ ] → [3 6 5 2 4]

wt	h	
3	12	0
6	20	1
5	15	2
2	6	3
4	10	4

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	12	12	12	12	12	12
2	0	0	0	12	12	12	20	20	20
3	0	0	0	12	12	15	20	20	27
4	0	0	6	12	12	18	20	21	27
5	0	0	6	12	12	18	20	22	27

(Ans)

Find the items selected.

$i = N \quad j = W$

if (dp[i][j] == dp[i-1][j]) → i^{th} item is rejected

$i = i - 1$

else → i^{th} item is selected

$i = i - 1 \quad j = j - w[i]$

Unbound KnapSack (0 - ∞ KnapSack)

Given N toys with their happiness and weight.

Find max total happiness that can be kept in a bag with capacity W.

[Toys can't be divided]

Items can be selected ∞ times.

value[ ] → [2 3 5]

weight[ ] → [3 4 7]

W = 8 , N = 3

2 times Ans = 6

val = [1 30]

wt = [1 50]

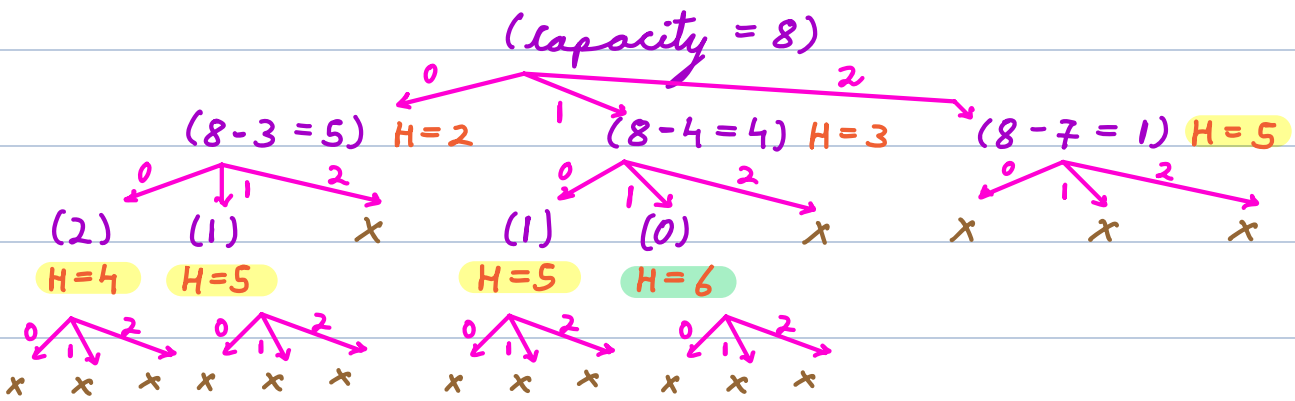
W = 100

100 times Ans = 1 * 100 = 100

value[ ] → [2 3 5]

weight[ ] → [3 4 7]

W = 8 , N = 3



dp[i] → Max happiness with capacity i.

dp[0] = 0

$$dp[i] = \max_{0 \le j \le N-1} (h[j] + dp[i - w[j]])$$

```
dp[0] = 0
for i → 1 to W {
    for j → 0 to (N-1) {
        if (i >= w[j])
            dp[i] = max(dp[i], h[j] + dp[i - w[j]])
    }
} return dp[W]
```

TC = O(N * W) SC = O(W)

knapsack → Category.

Agenda :-

→ Rod cutting

→ coin change

→ 0-1 knapsack - 2.

Agenda :- → val
→ wt

cap □

0-1 knapsack

↳ 1 item of 1 type

unbounded kp.

→ no limit on no of items.

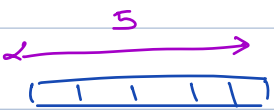
n items

wt.
val

cap.

Ques Rod cutting problem

Given a rod of len n, and an array of len n, A[i] → price of i len rod, find maxm val we can obtain by selling the rod.



N = 5,

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

A = [0, 1, 4, 2, 5, 6]

2 + 3 → 6

4 + 1 → 6

2 + 2 + 1 → 9

3 + 1 + 1 → 4

1 + 1 + 1 + 1 + 1 → 5

5 → 6

Capacity → len of rod

wt → diff. lengths.

price → —

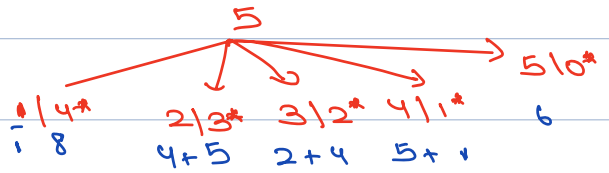
unbounded knapsack.

dp[i] → max profit we can get by selling a rod of len i.

n = 5,

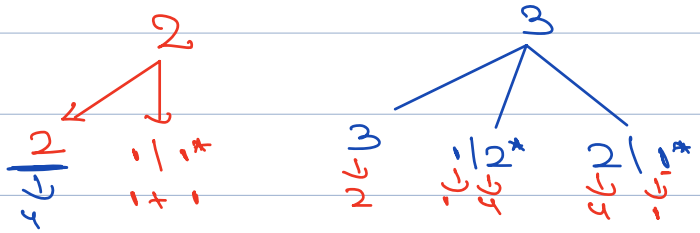
0	1	2	3	4	5
0	1	4	2	5	6

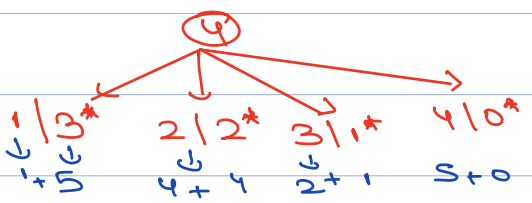
A = [0, 1, 4, 2, 5, 6]



dp →

0	1	2	3	4	5
0	1	4	5	8	9
-	1	2	1,2	2,2	1,2,2





$\forall i$ dp[i] = 0';

T.C → O(n²)

S.C → O(n)

for i → 1 to N → 4

for j → 1 to i → 1 to 4 → 1, 2.

dp[i] = max(dp[i], A[j] + dp[i-j]);

arr[1] + dp[3]

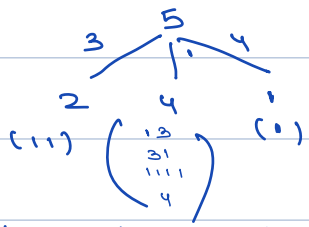
arr[2] + dp[2]

Ques Coin Change Permutation $(x,y) \neq (y,x)$

K = 5 {1,4} {4,1} {1,1,1,1,1}

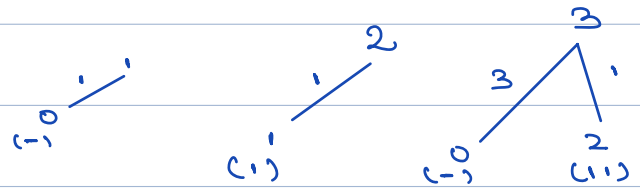
Ans = 6 {1,1,3}, {3,1,1}, {1,3,1}

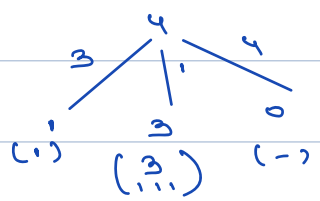
A = [3,1,4]



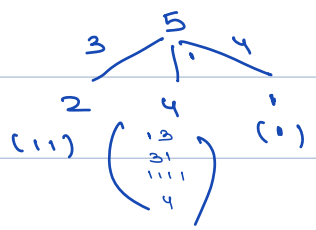
dp[i] → no of permutations to pay i rupees.

0	1	2	3	4	5
dp→ 1	1	1	2	4	6
-	1	1 1	3 1 1 1	1 3 3 1 1 1 1 1 4	1 1 3 1 3 1 3 1 1 1 1 1 1 1 4 1 1 4





A = [3,1,4]



```
int [] dp = new int [k+1];

dp[0]=1;

for (i=1; i<=k; i++) {

    for (j=0; j<n; j++) {

        if( i - arr[j]>=0){

            dp[i]+= dp[i- arr[j]]

        }
    }
}
```

i = 5

j = 0 → 3 times

3 → 3

T.C → O(N*K)

S.C → O(K)

Break 7:59 Am - 8:09 Am

Ques

Ques Coin Change Combination $(x,y) = (y,x)$

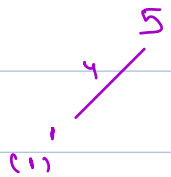
$k = 5$

Ans = 3.

$A = [3, 1, 4]$

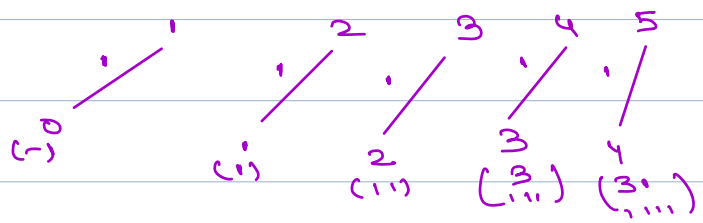
{1,4} ~~{4,1}~~ {1,1,1,1,1}

{1,1,3}, ~~{3,1,1}~~, ~~{1,3,1}~~



dp[i] → no of Combinations to pay i rupees.

	0	1	2	3	4	5
dp →	1	1	1	~~2~~ 2	~~3~~ 3	2 3
	-	1	11	3 111	31 1111 4	311 11111 14



```
int [] dp = new int [k+1];

dp[0]=1;

for(i=1; i<=k; i++){          <-- order of these two.
    for(j=0; j<m; j++){
        if(i-arr[j]>=0){
            dp[i]+= dp[i-arr[j]]
        }
    }
}
```

T.C -> O(N*K)

S.C -> O(K)

Ques 0-1 knapsack - 2.

Given n items, with a wt & val. find max val which can be obtained by picking items s.t, total weight of all items <= x.

Note 1 :- Every item can be picked only 1 time.

Note 2 :- We can't pick partially.

T.C
↓
O(n * cap)
↓
500×10^9
5×10^{11}

$1 <= N <= 500$

$1 <= val[i] <= 50$

$1 <= wt[i] <= 10^9$

$1 <= cap <= 10^9$

dp[i][j] -> Max profit we can get in a bag of cap j s.t we choose 0-i items.

↓

dp[n][x]

$1 <= n <= 500$

$1 <= val[i] <= 50$

$1 <= wt[i] <= 10^9$

$1 <= cap <= 10^9$

max Profit

$500 \times 50 \Rightarrow 25 \times 10^3$

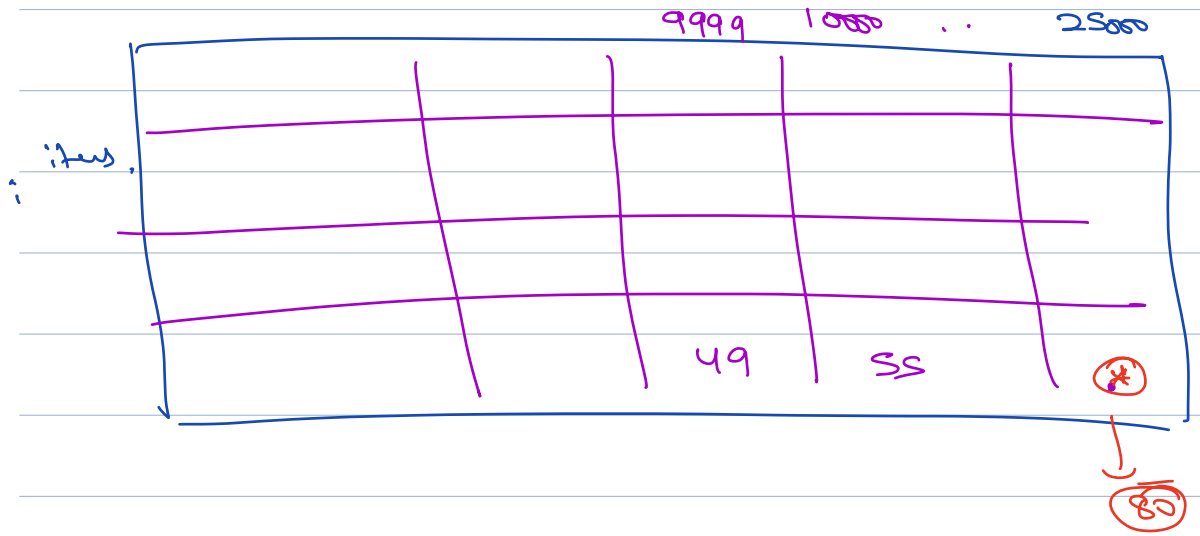
$\Rightarrow 25000$.

cap = 80.

dp[i][J] → min wt. req. to get J profit

if we choose first i items.

J → Profit.



wt 10, 20, 30

val 1000 500 50

dp[i][J] → min wt. req. to get J profit

if we choose first i items.

J → Profit.

wt[] → 4 1 5 4 3 7 4

val[] → 2 1 2 1 2 1 2

Profit

cap = 6.

val	wt	0	1	2	3	4	5
-	-	0	∞	∞	∞	∞	∞
2	4	0	∞	4	∞	∞	∞
1	1	0	1	4	5	∞	∞
2	5	0	1	4	5	9	10

```
private int knapsack_Unbounded_Memo(int cap, int[] val, int[] wts, int n, int[] dp) {
    if (dp[cap] != -1)
        return dp[cap];
   // Base case:
    if (cap == 0 || n == 0)
        return 0;


    return dp[cap] = Math.max(
            val[n - 1] + knapsack_Unbounded_Memo(cap - wts[n - 1], val, wts, n, dp),
                            knapsack_Unbounded_Memo(cap, val, wts, n - 1, dp)
    );
}
```