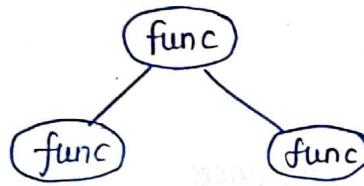


# Dynamic Programming

-Pavan Patel

## • Dynamic Programming :-

DP = Enhanced Recursion



function calls itself with smaller inputs.

- Parent of Dynamic Programming is recursion
- DP asks for optimal Solution
- Choice / choose → Recursion can be applied
- →



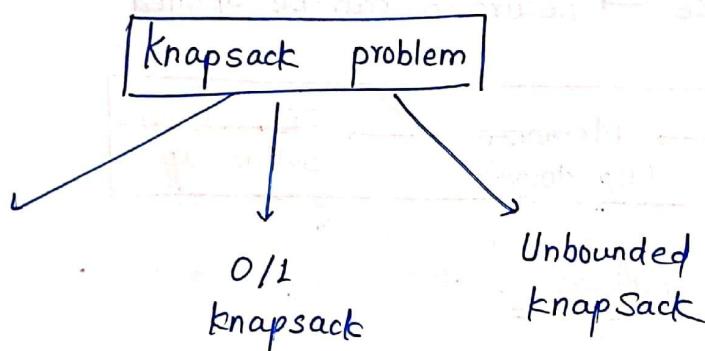
## • Variations in DP problems :-

- ① 0 - 1 knapsack (6)
- ② Unbounded knapsack (5)
- ③ Fibonacci (7)
- ④ Longest Common Subsequence (15)
- ⑤ Longest Increasing Subsequence (10)
- ⑥ Kadane's Algorithm (6)
- ⑦ Matrix Chain Multiplication (7)
- ⑧ DP on Trees (4)
- ⑨ DP on matrix (14)
- ⑩ Others (5)

Total problems/Variations = 79 problems

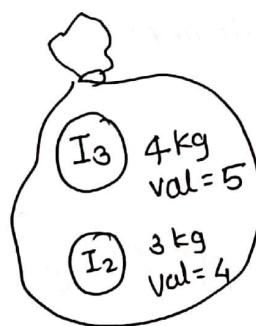
# 0-1 knapsack Problem

- 1] Subset Sum
- 2] Equal sum partition
- 3] Count of subset sum
- 4] Minimum subset sum difference
- 5] Target sum
- 6] Number of subsets with given difference



<u>problem</u>	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>
wt[] :	1	3	4	5
val[] :	1	4	5	7

$W = 7$   
(capacity)



$$\begin{aligned} &\Rightarrow 4 + 5 \\ &= \underline{\underline{9 \text{ profit}}} \end{aligned}$$



In fractional knapsack, fraction of item can be taken.  
 But in 0/1 knapsack, either we have to include that item or exclude the item  
 In unbounded knapsack, we can take multiple instances of item.

How to identify?

wt[] : 1 3 4 5

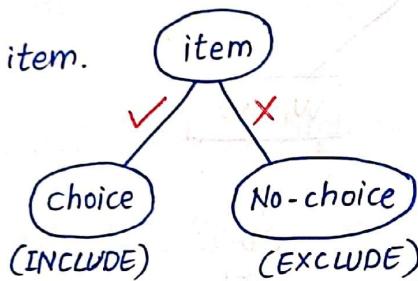
value[] : 1 4 5 7

w : 7 kg

output: maxProfit ?

so they are asking optimal profit

for every item.



{ choice

Hence this is a problem of DP.

DP: Recursive  $\rightarrow$  Memoization  $\rightarrow$  Bottom-up  
 Solution (Top-down)

DP  $\rightarrow$  Recursive + Storage

## • 0/1 knapsack Recursion

Input :-

$wt[] =$	1	3	4	5
$val[] =$	1	4	5	7
$W = 7$				

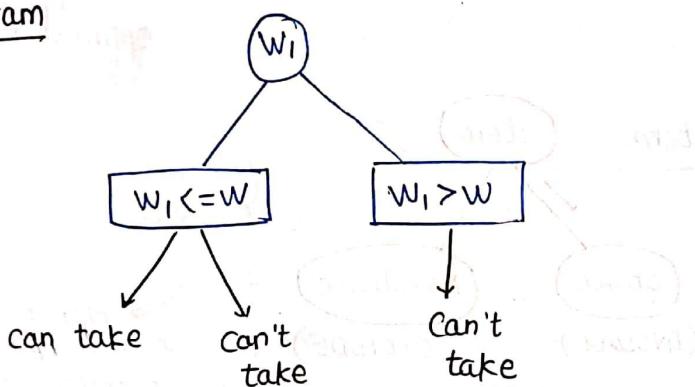
let suppose

Item 1  
( $w_1$ )



if the weight of the item > the capacity of the bag.

choice-Diagram



```

int knapsack(int wt[], int val[], int capacity, int n) {
    // Base Condition
    if (n == 0 || capacity == 0)
        return 0;
    if (wt[n-1] <= capacity)
        return max(val[n-1] + knapsack(wt, val, capacity - wt[n-1], n-1),
                  knapsack(wt, val, capacity, n-1));
    else if (wt[n-1] > capacity)
        return knapsack(wt, val, capacity, n-1);
}
  
```

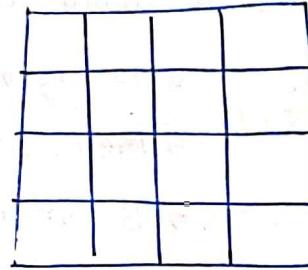
## How to think about the base condition?

① Base condition → Think of the smallest valid input

### 0/1 knapsack top-down Approach: Memoization

so in the recursive knapsack the variables that are changing →

- a)  $n$
- b) capacity



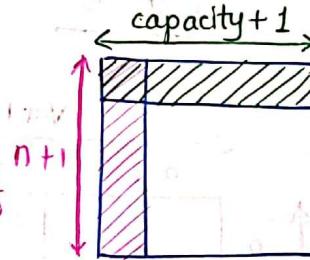
So for these variables only, we have to prepare the table.

knapsack (wt[], val[], capacity, n) {

int dp[n+1][capacity+1];

memset(dp, -1, sizeof(dp));

Initialization of table with "-1".



#### Code

```
int dp[102][1002]
memset(dp, -1, sizeof(t));
```

Let's take the constraints

$$n \leq 100$$

$$\text{capacity} \leq 1000$$

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

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

```
if (dp[n][capacity] != -1)
```

```
return dp[n][capacity];
```

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

~~```
dp[n][capacity] = max(knapsack(
```~~~~```
dp[n][capacity] = val[n-1] + max(knapsack(wt, val, capacity - wt[n-1],
```~~~~```
knapsack(wt, val, capacity, n-1);
```~~

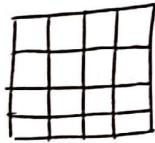
```
else if (wt[n-1] > capacity)
```

```
dp[n][capacity] = knapsack(wt, val, capacity, n-1);
```

```
return dp[n][capacity];
```

## Bottom-up Approach

In this only table is there

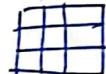


## Why bottom-up approach is best?

Cause it avoid the error of stackoverflow error.

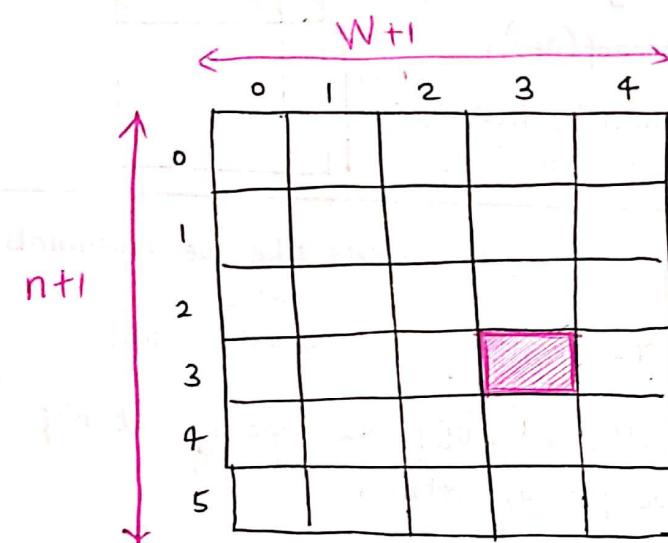
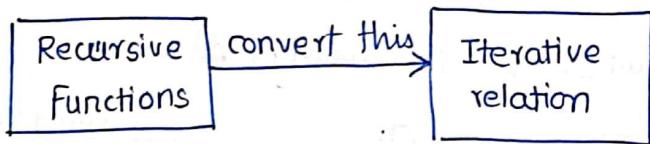
→ Recursive → R.C calls

→ Memoization → R.C + 

→ Bottom-up → 

## Step 1 :- Initialization

## Step 2 :-



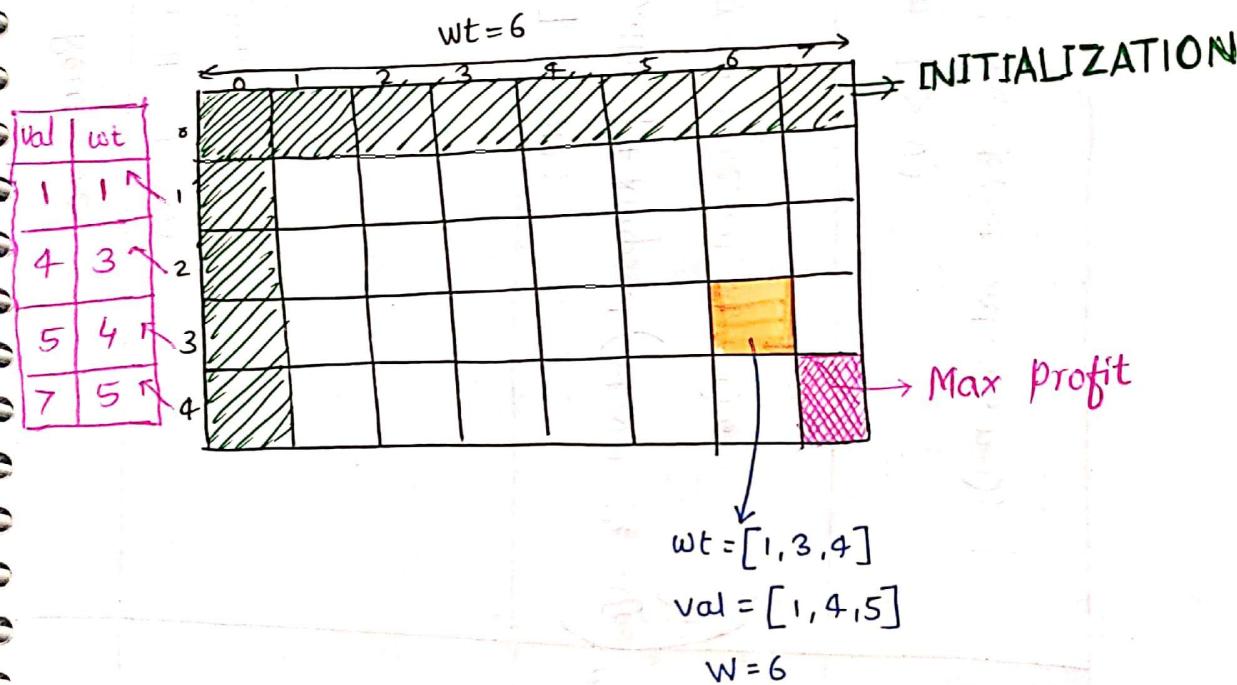
|   |   |   |   |   |   |       |
|---|---|---|---|---|---|-------|
|   | 0 | 1 | 2 | 3 | 4 | $n+1$ |
| 0 |   |   |   |   |   |       |
| 1 |   |   |   |   |   |       |
| 2 |   |   |   |   |   |       |
| 3 |   |   |   |   |   |       |
| 4 |   |   |   |   |   |       |
| 5 |   |   |   |   |   |       |

Example :

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

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

$$W = 7$$



### Why initialization is important?

While Converting the recursive calls to the iterative relation...

Then the base condition of recursive calls

changes into

Initialization in the iterative soln.

### Recursive

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

if (wt[n-1] <= w) {
    return max (val[n-1] + knapsack(wt, val, W - wt[n-1],
                                     n-1),
                knapsack(wt, val, W, n-1));
}

else if (wt[n-1] > w)
    return knapsack(wt, val, W, n-1);
```

### Iterative

```
for (i=0 ; i<n+1 ; i++) {
    for (j=0 ; j<w+1 ; j++) {
        if (i==0 || j==0) {
            dp[i][j] = 0;
        }
        else if (wt[n-1] <= w)
            dp[n][w] = max (val[n-1] + dp[n-1][W - wt[n-1]],
                             dp[n-1][w]);
        else if (wt[n-1] > w)
            dp[n][w] = dp[n-1][w];
```

Code :-

```
int knapsack (int wt[], int val[], int n, int w){  
    vector<vector<int>> dp(n+1, vector<int>(w+1));  
    for(int i=0; i<n+1; i++) {  
        for(int j=0; j<w+1; j++) {  
            if(i==0 || j==0) {  
                dp[i][j] = 0;  
            }  
        }  
    }  
  
    for(int i=1; i<n+1; i++) {  
        for(int j=1; j<w+1; j++) {  
            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];  
            }  
        }  
    }  
  
    return dp[n][w];  
}
```

## • Subset-Sum Problem

Given an array:-

$$\text{arr}[] = \{2, 3, 7, 8, 10\}$$

$$\text{Sum} = 11$$

If a subset which having the sum of its element is equal to the given sum then return true.

Here  $\Rightarrow$  subset =  $\{3, 8\} \rightarrow 3 + 8 = 11$  ]  $\sum = 11 \rightarrow$  return true.

We are having the choice to include the element or exclude the element in the subset.

$$\text{arr}[] = [2 \boxed{3} \boxed{7} \boxed{8} \boxed{10}] \quad \text{sum} = 11$$

If we compare this problem with the knapsack,

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



$dp[n+1][\text{sum}+1]$

|   |  | → sum |   |   |   |   |   |   |   |   |   |    |    |
|---|--|-------|---|---|---|---|---|---|---|---|---|----|----|
|   |  | 0     | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 0 |  | T     | F | F | F | F | F | F | F | F | F | F  | F  |
| 1 |  | T     |   |   |   |   |   |   |   |   |   |    |    |
| 2 |  | T     |   |   |   |   |   |   |   |   |   |    |    |
| 3 |  | T     |   |   |   |   |   |   |   |   |   |    |    |
| 4 |  | T     |   |   |   |   |   |   |   |   |   |    |    |
| 5 |  | T     |   |   |   |   |   |   |   |   |   |    |    |

↓  
n  
If size of array  $\geq 0$

still we can create with subset sum = 0

[Empty subset]

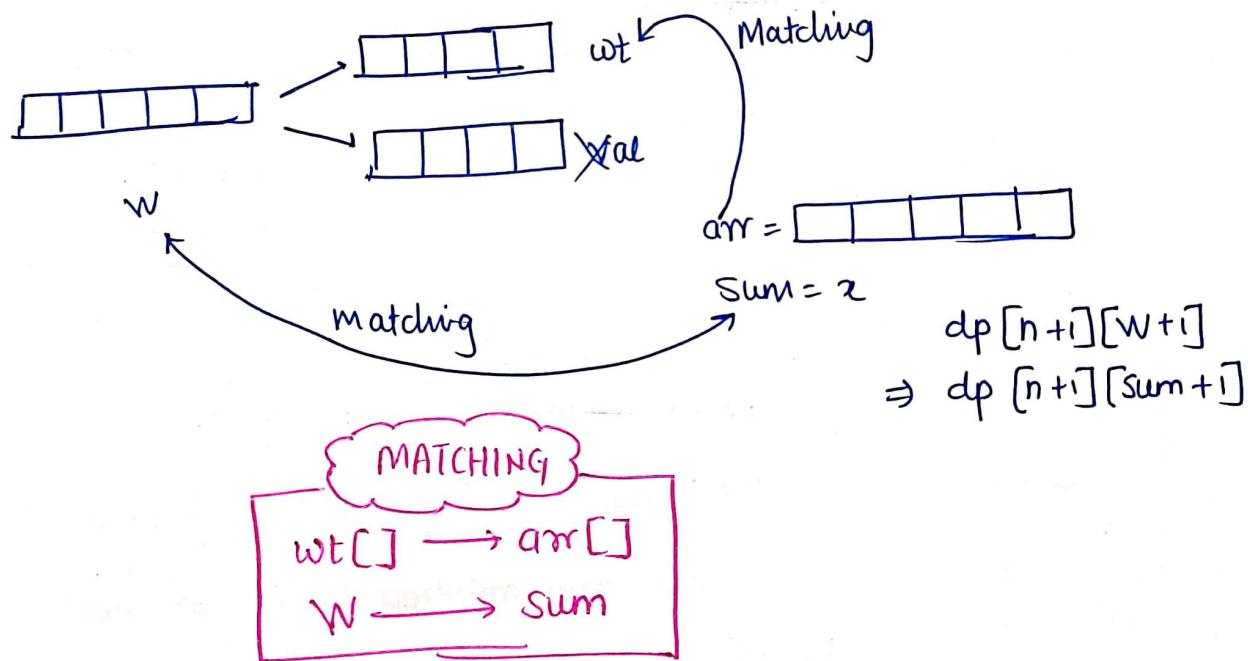
But if the array of size = 0 and sum  $> 0$

$\rightarrow$  Then we can't create any subset

$\rightarrow$  So Initialize with 'F'!

## Initialization

```
for(int i=0 ; i<n+1 ; i++) {  
    for(int j=0 ; j<sum+1 ; j++) {  
        if(i==0) {  
            dp[i][j] = false;  
        }  
        if(j==0) {  
            dp[i][j] = true;  
        }  
    }  
}
```



### Knapsack

```
if (wt[i-1] <= j) {
```

$$dp[i][j] = \max \left[ val[i-1] + dp[i-1][j - wt[i-1]], dp[i-1][j] \right]$$

```
else
```

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

### Subset-Sum

```
if (arr[i-1] <= j) {
```

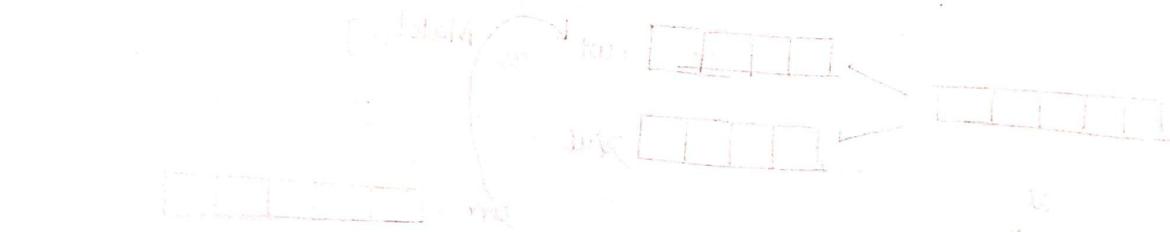
$$dp[i][j] = dp[i-1][j - arr[i-1]] + arr[i-1]$$

```
||
```

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

```
else
```

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



subset sum  
target sum  
subset sum  
target sum  
subset sum

## • Equal Sum partition Problem :-

Given an array, we have to find two different subset such that sum of their elements should be equal.

$$arr = [1, 5, 11, 5]$$

O/P  $\Rightarrow$  True

$$\begin{array}{ll} \{1, 5, 5\} & \{11\} \\ 1+5+5=11 & \downarrow 11 \\ \checkmark & \end{array}$$

### Flow

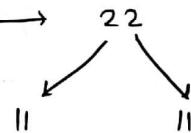
- ① problem statement
- ② Similar with subset sum
- ③ Odd/Even significance
- ④ Code variation

→ The basic condition is → if the sum of the array elements is even  $\rightarrow$  Then only we can divide the array  
Else we can't do the partition.

```
int sum=0;
for(int i=0 ; i<size ; i++){
    sum += arr[i];
}
if(sum%2==0)  $\rightarrow$  return false;
```

$$arr = [1, 5, 11, 5] \rightarrow sum = 22$$

Now for Equal partition  $\rightarrow$



so we have to find the subset whose sum is 11.  
so this problem now is converted into subset sum problem

```
if (sum/.2 == 0)
    return subsetSum (arr, sum/2);
```

```
for(int i=0; i<n; i++) {
    sum += arr[i];
}
if (sum/.2 != 0) return false;
if (sum/.2 == 0) return subsetSum (arr, sum/2);
```

## Count of Subsets with a given sum :-

Input:-

|                                      |
|--------------------------------------|
| $\text{arr}[] = [2, 3, 5, 6, 8, 10]$ |
| Sum = 10                             |

We have to count the number of Subsets with the given sum.

|   |   |   |   |   |    |
|---|---|---|---|---|----|
| 2 | 3 | 5 | 6 | 8 | 10 |
|---|---|---|---|---|----|

$$\begin{array}{l} \text{sum} = 10 \\ \quad \left. \begin{array}{l} \{2, 8\} \\ \{2, 3, 5\} \\ \{10\} \end{array} \right\} \end{array} \text{Answer :- } (3)$$

### Initialization

$$\text{arr}[] = \{2, 3, 5, 6, 8, 10\}$$

$$\text{sum} = 10$$

$$t[n+1][\text{sum}+1]$$

$$t[7][11]$$

|  |  | Sum            |                |                |                |                |                |                |                |                |                |    |
|--|--|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----|
|  |  | 0              | 1              | 2              | 3              | 4              | 5              | 6              | 7              | 8              | 9              | 10 |
|  |  | T <sub>0</sub> | F <sub>0</sub> |    |
|  |  | T <sub>1</sub> |                |                |                |                |                |                |                |                |                |    |
|  |  | T <sub>1</sub> |                |                |                |                |                |                |                |                |                |    |
|  |  | T <sub>1</sub> |                |                |                |                |                |                |                |                |                |    |
|  |  | T <sub>1</sub> |                |                |                |                |                |                |                |                |                |    |
|  |  | T <sub>1</sub> |                |                |                |                |                |                |                |                |                |    |
|  |  | T <sub>1</sub> |                |                |                |                |                |                |                |                |                |    |
|  |  | T <sub>1</sub> |                |                |                |                |                |                |                |                |                |    |

↓  
n

$\{\}$   $\rightarrow F \rightarrow <0$  Subsets

$\rightarrow T \rightarrow >0$  subsets

so  $\rightarrow F \rightarrow 0$   
 $T \rightarrow 1$

Subset  
Sum  
Code

$$\begin{aligned} \text{if } (\text{arr}[i-1] \leq j) \\ t[i][j] = \frac{t[i-1][j]}{\text{Exclude}} \quad \frac{t[i-1][j - \text{arr}[i-1]]}{\text{Include}} \\ \text{else} \\ t[i][j] = t[i-1][j] \end{aligned}$$

but Subset Sum ~~code~~ → return type was bool.

so if any of the parameter is true then it was giving true

but Here, the return type is "~~bool~~" and we have to count the # of subsets

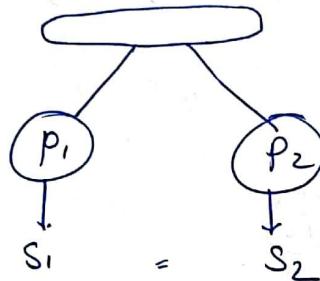
$$t[i][j] = t[i-1][j] \quad \# \text{ count of subsets} \\ \begin{array}{lll} \textcircled{1} & T & || \\ \textcircled{2} & F & || \\ \textcircled{3} & T & || \\ \textcircled{4} & F & || \end{array} \quad \begin{array}{l} F \longrightarrow 1+0=1 \\ T \longrightarrow 0+1=1 \\ T \longrightarrow 1+1=2 \\ F \longrightarrow 0+0=0 \end{array}$$

So In # of subsets with given sum  
we have to add both parameters  
i.e.,

$$t[i][j] = t[i-1][j] + t[i-1][j - \text{arr}[i-1]]$$

## Minimum Subset Sum difference

$\text{arr}[] = [1 \ 6 \ 11 \ 5]$   
output = 1



$S_1 + S_2 \Rightarrow \text{Even} \rightarrow \text{Equal Sum partition}$   
i.e.  $S_1 - S_2 = 0$

But in this question, the difference (absolute) of subset sum  
should be minimum

$$\text{abs}(S_1 - S_2) = \text{minimum}$$

In the above example,

the subsets having the minimum difference

$$\{1, 6, 5\} \quad \{11\}$$

$$1+6+5 = 11$$

$$12 \quad 11$$

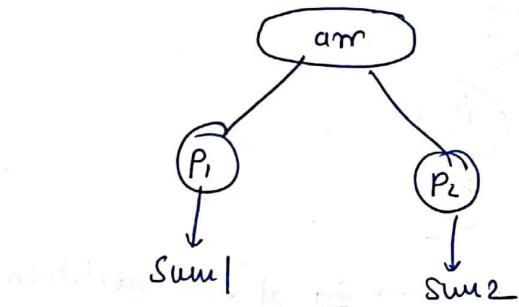
$$(12 - 11) \longrightarrow 1 \rightarrow \underline{\text{Answer}}$$

This problem is similar to Equal Sum partition

arr[] 

|   |   |    |   |
|---|---|----|---|
| 1 | 6 | 11 | 5 |
|---|---|----|---|

Now we have to find the range of sum of partitions



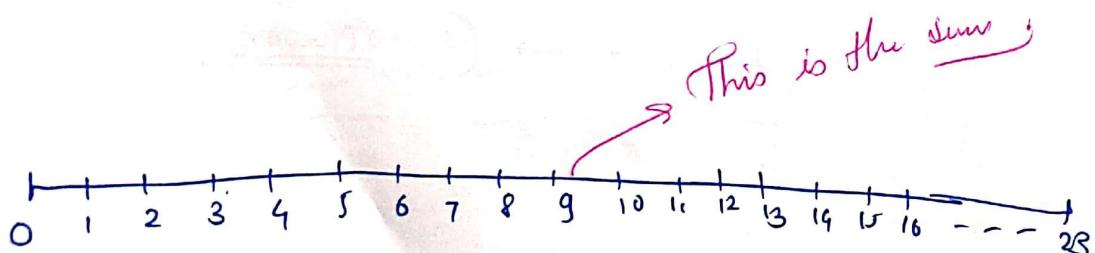
WHAT COULD BE THE RANGE ?

|   |   |   |    |
|---|---|---|----|
| 1 | 6 | 5 | 11 |
|---|---|---|----|

 Either Subset is empty  $\{\}$   $\rightarrow S_1 = 0$

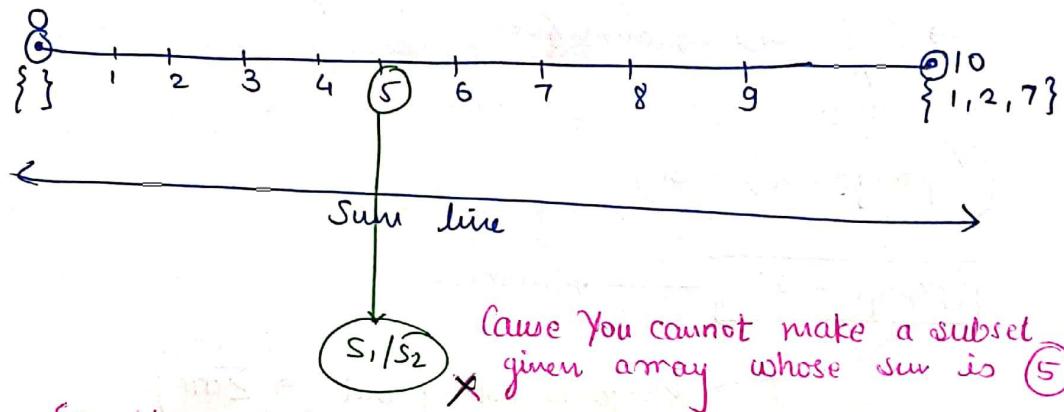
or Subset having all the elements of array  $\{1, 6, 5, 11\}$   $\rightarrow S_2 = 23$

Hence the range is  $[0, 23]$   
of sum

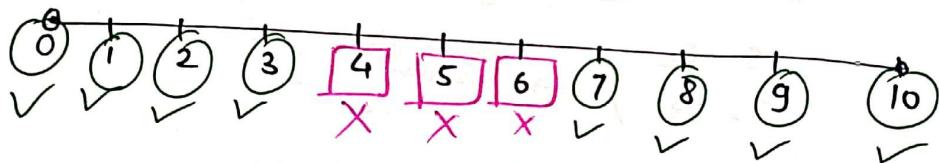


Ex:-

$$\text{arr} = \boxed{1 \ 2 \ 1 \ 7}$$



So Now find out How many entries on the number line satisfies the condition  
or  
Means can we make  $S_1$  &  $S_2$  from the array.



$$S_1/S_2 = \{0, 1, 2, 3, 7, 8, 9, 10\}$$

Think



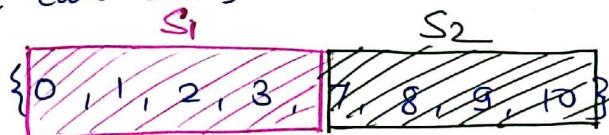
This array consists both  $S_1$  and  $S_2$ . if  $S_1=1$  then

$(\sum \text{arr}-1) \rightarrow$  is also present  $\rightarrow$  i.e.  $\rightarrow 9$

if  $S_1=2 \rightarrow (\sum \text{arr}-2) \rightarrow$  is also present  $\rightarrow 8$

$\therefore S_1=3 \rightarrow (\sum \text{arr}-3) \rightarrow 7$  is also present so

We can see,



So basically we have to find the only one partition, another partition will automatically derived.

$$(S_1 - S_2) \rightarrow \text{minimize}$$

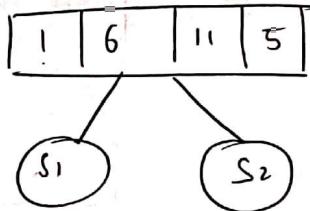
$$\rightarrow (S_2 - S_1) \rightarrow \text{minimize}$$

$$((\text{Range} - S_1) - S_1)$$

$$[(\text{Range} - 2S_1)] \rightarrow \text{Minimize}$$

Range is nothing but  $\sum \text{arr}$

Sum Up



$$(S_1 - S_2) \rightarrow \text{minimize}$$

$$[(\text{Range} - 2S_1)] \rightarrow \text{minimize}$$

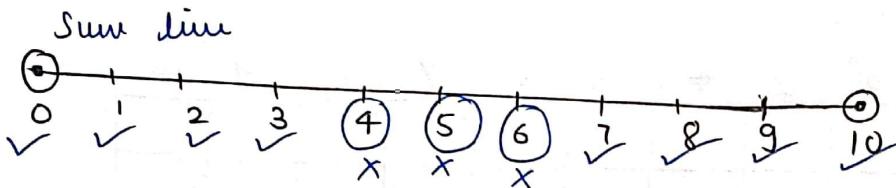
$$\sum \text{arr}$$

$S_1 \rightarrow \text{Smaller}$

$S_2 \rightarrow \text{Greater}$

Ex:-  $\text{arr}[] = [1 | 2 | 7]$

$n = 3$   
Range = 10



$dp[n+1][\text{Range}+1]$

|           |   |   |   |   |   |   |   |   |   |   |    |
|-----------|---|---|---|---|---|---|---|---|---|---|----|
|           | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1         |   |   |   |   |   |   |   |   |   |   |    |
| 2         |   |   |   |   |   |   |   |   |   |   |    |
| 3         |   |   |   |   |   |   |   |   |   |   |    |
| [1, 2, 7] |   |   |   |   |   |   |   |   |   |   |    |

This block will tell whether subset is present for sum=0

check for sum = 1

similarly

What does this block will tell?

size of array = 1  $\rightarrow \{1\}$   
Sum = 7

We have to find the last row of the dp table

so this problem will solved by the code  
It will check whether we are getting the sum as '7' or not



⑧ SubsetSum( int arr[], int Range ) {

|   |   |   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|---|---|----|
|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 0 |   |   |   |   |   |   |   |   |   |   |    |
| 1 |   |   |   |   |   |   |   |   |   |   |    |
| 2 |   |   |   |   |   |   |   |   |   |   |    |
| 3 | T | T | T | T | F | F | F | T | T | T | T  |

}

arr[] = 

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
|---|---|---|

 sum=0 → ✓

|   |   |   |
|---|---|---|
| 1 | 2 | 7 |
|---|---|---|

 sum=1 → ✓

|   |   |   |
|---|---|---|
| 1 | 2 | 7 |
|---|---|---|

 sum=2 → ✓

|   |   |   |
|---|---|---|
| 1 | 2 | 7 |
|---|---|---|

 sum=3 → ✓

sum=4 → ✗  
bit at sum shw  
sum to work with

sum=5 → ✗  
sum qd

sum=6 → ✗

|   |   |   |
|---|---|---|
| 1 | 2 | 7 |
|---|---|---|

 sum=7 → ✓

|   |   |   |
|---|---|---|
| 1 | 2 | 7 |
|---|---|---|

 sum=8 → ✓

|   |   |   |
|---|---|---|
| 1 | 2 | 7 |
|---|---|---|

 sum=9 → ✓

|   |   |   |
|---|---|---|
| 1 | 2 | 7 |
|---|---|---|

 sum=10 → ✓

We only need the last row of the dp table.

I'll add the last row in our vector until half our tree  
cause I want to store smaller values only  $(s_1)$

Range = 10  $\rightarrow \Sigma_{arr}$

|   |   |   |   |     |
|---|---|---|---|-----|
| 0 | 1 | 2 | 3 | ... |
|---|---|---|---|-----|

int mini = INT\_MAX;  
for (int i=0 ; i<vec.size() ; i++)

mini = min(mini, vec[i]);

{ mini = min(mini, Range - (2 \* vec[i])); }

return mini;

11.

12.

13.

14.  $\leftarrow$  Min + Max of current

15.  $\leftarrow$  Min + Max of current

16.  $\leftarrow$  Min + Max of current

17.  $\leftarrow$  Min + Max of current

18.  $\leftarrow$  Min + Max of current

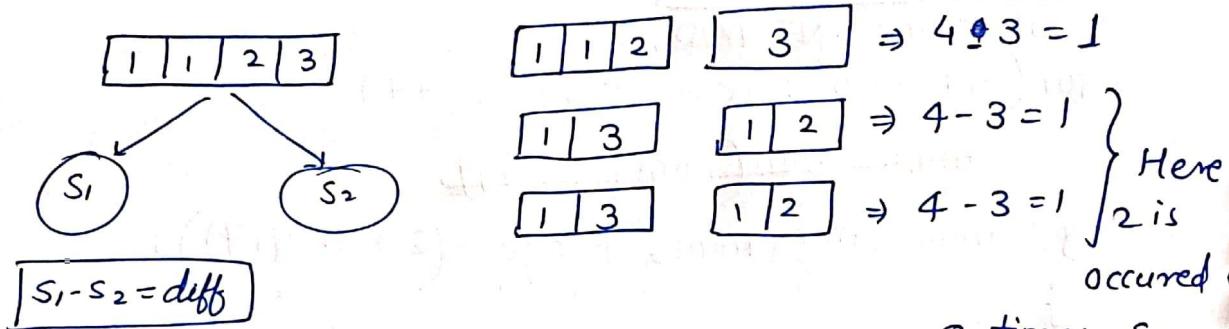
19.  $\leftarrow$  Min + Max of current

20.  $\leftarrow$  Min + Max of current

21.  $\leftarrow$  Min + Max of current

## Count the no. of Subsets with a given difference

$\text{arr}[] = [1 | 1 | 2 | 3]$   
 $\text{diff} = 1$



let's called diff as 'diff'

$$\boxed{\sum(S_1) - \sum(S_2) = \text{diff}} \quad \text{Eq } ①$$

$$\boxed{\sum(S_1) + \sum(S_2) = \sum(\text{arr})} \quad \text{Eq } ②$$

$$\boxed{2\sum(S_1) = \text{diff} + \sum(\text{arr})} \quad ③$$

$$\Rightarrow 2\sum(S_1) = \text{diff} + \sum(\text{arr})$$

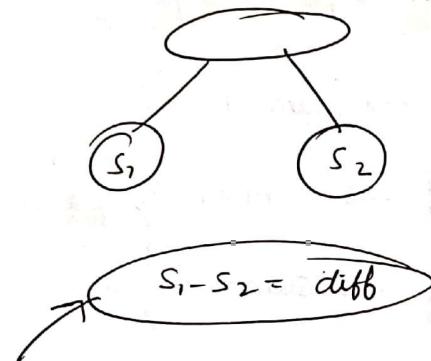
$$\boxed{\sum(S_1) = \frac{\text{diff} + \sum(\text{arr})}{2}}$$

$$\begin{aligned} &= \frac{1+7}{2} \\ &= 4 \quad \text{so} \quad \sum(S_1) = 4 \end{aligned}$$

When sum of subset<sub>1</sub> is 4 then only we have ~~that~~  
the given diff.

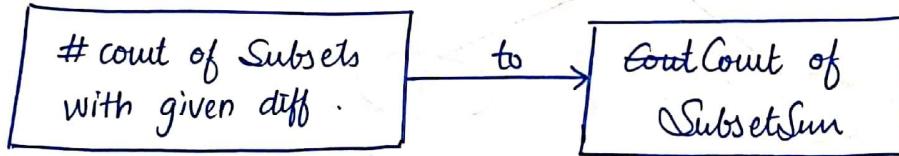
i.e.  $S_1 - S_2 = \text{diff}$

count?



we have to count

So we have reduced this ~~count~~



Code

```
int sum = diff + sum(arr)  
          ---  
          2
```

return count of SubsetSum(arr, sum)

```
int countofSubsetSum ( int [ ] arr, int sum )
```

//Initialization

//Traversing

```
if ( arr [ i - 1 ] <= j )
```

```
    t [ i ] [ j ] = t [ i - 1 ] [ j ] + t [ i - 1 ] [ j - arr [ i - 1 ] ] ;
```

else

```
    t [ i ] [ j ] = t [ i - 1 ] [ j ]
```

```
return t [ n ] [ sum ] ;
```

## Target Sum

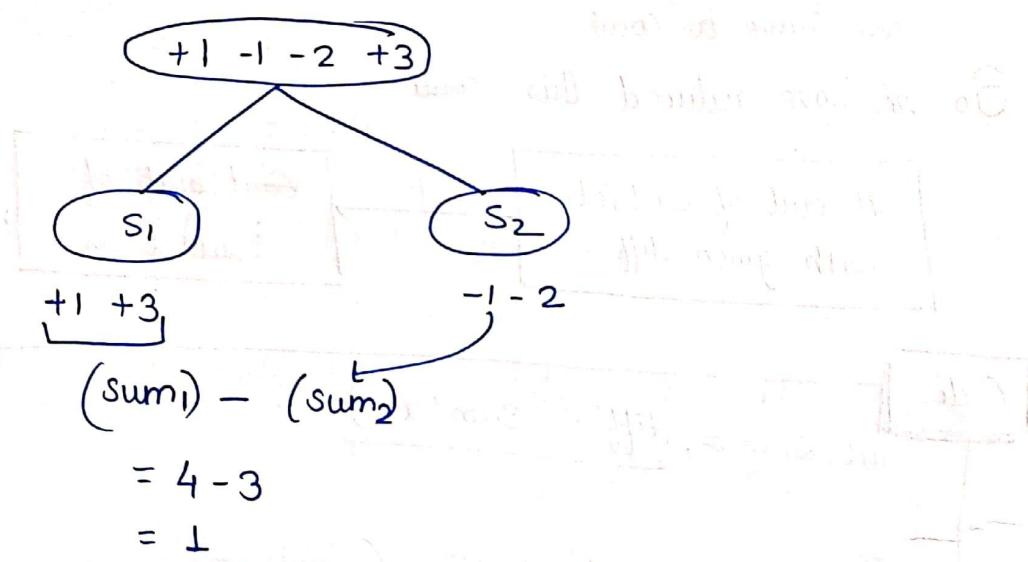
|       |                                                                                                                                      |   |   |   |   |
|-------|--------------------------------------------------------------------------------------------------------------------------------------|---|---|---|---|
| arr:  | <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>1</td><td>1</td><td>2</td><td>3</td></tr> </table> | 1 | 1 | 2 | 3 |
| 1     | 1                                                                                                                                    | 2 | 3 |   |   |
| Sum:  | 1                                                                                                                                    |   |   |   |   |
| O/p = | 3                                                                                                                                    |   |   |   |   |

↓  
+/-  
⇒ Allowed

like we can take any of the signs to any element

ex:-

|                        |           |   |
|------------------------|-----------|---|
| <del>+1 -1 -2 +3</del> | → sum = 1 | } |
| <del>+1 +1 +2 -3</del> | → sum = 1 |   |
| <del>-1 +1 -2 +3</del> | → sum = 1 |   |



So basically ⇒

$$\text{sum}_1 - \text{sum}_2 = \text{sum}$$

This is same as the count of subset sum diff.

# Unbounded

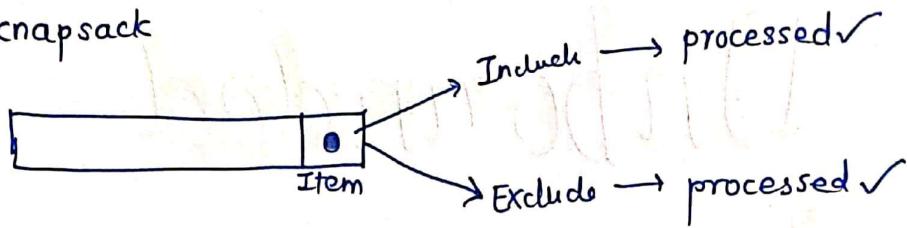
# Knapsack

- Related problems :-

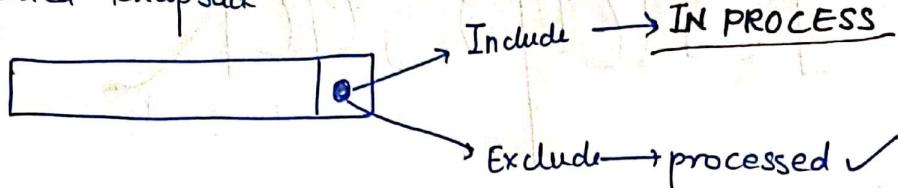
- ① Rod cutting
- ② Coin change - I
- ③ Coin change - II
- ④ Maximum Ribbon cut

- ⇒ In unbounded Knapsack, multiple occurrences of item is allowed.
- ⇒ In 0/1 Knapsack, If we include / exclude the item, then it is considered as processed item. means, we can't consider in further iteration.
- ⇒ In unbounded Knapsack, We can consider a single item, multiple items.
- Ex:-  
let's suppose I like Ice-cream, then I can take Ice-cream multiple times  
and  
If I don't want burger, then I'll not consider it even if it has been offered to me multiple times

In 0/1 knapsack



In unbounded knapsack



Code Variation:

0/1 knapsack code ⇒

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 |   |   |   |   |   |   |
| 0 |   |   |   |   |   |   |
| 0 |   |   |   |   |   |   |

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

In this, if we consider the element then we add the value of that element and we move forward i.e., (n-1) but in unbounded knapsack, we can take the element multiple times so we don't have to (n-1),

Variation

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

## • Rod Cutting Problem

You are given a rod of length ' $N$ ', you can cut it infinite no. of times. price array is given. such that you can; if you cut the rod with length ' $i$ ' then you can take the price of that piece from  $\text{price}[i]$ .

Ex:

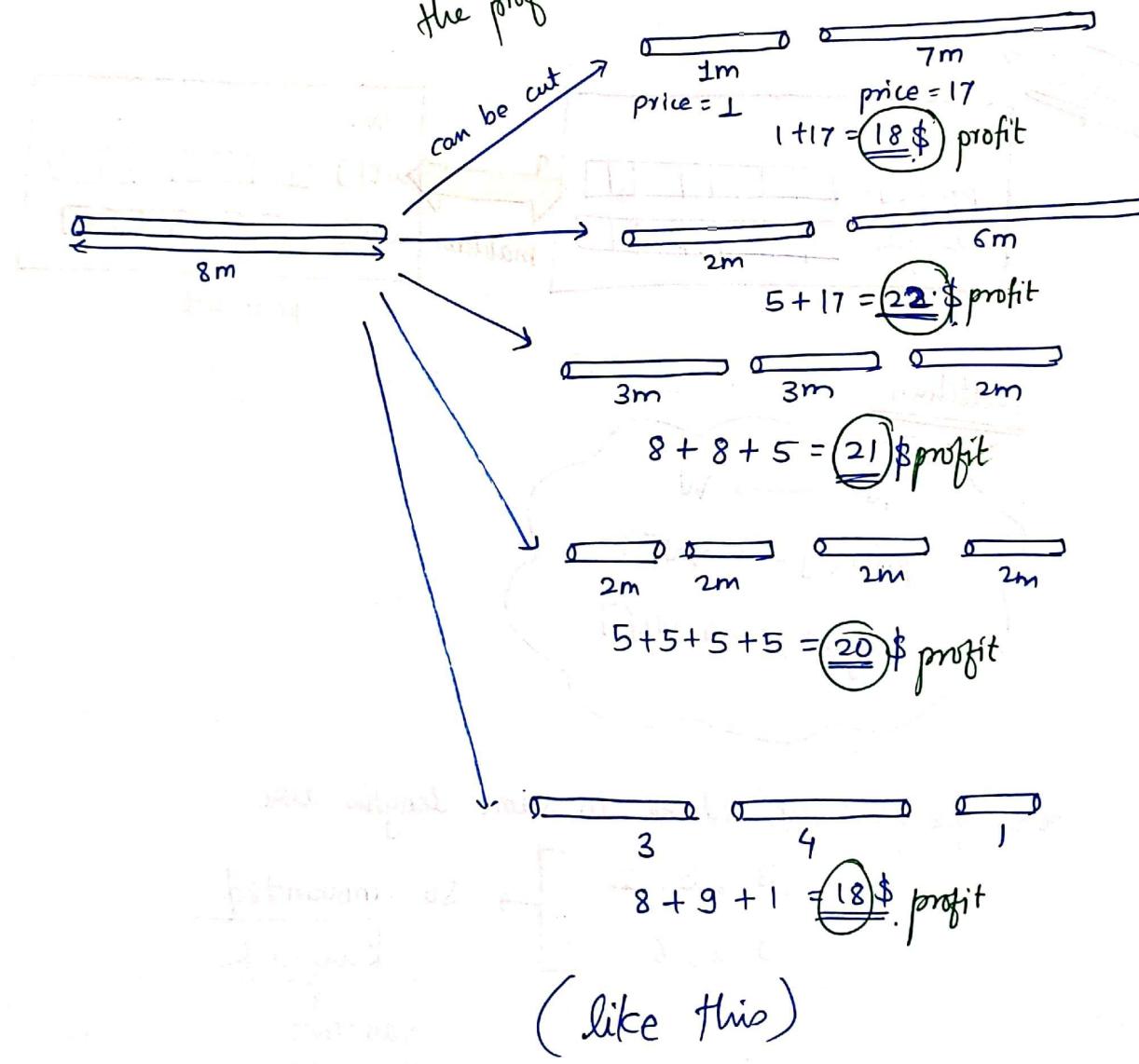
$$\text{length}[] = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline \end{array}$$

$$\text{price}[] = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 5 & 8 & 9 & 10 & 17 & 17 & 20 \\ \hline \end{array}$$

$$N = 8$$

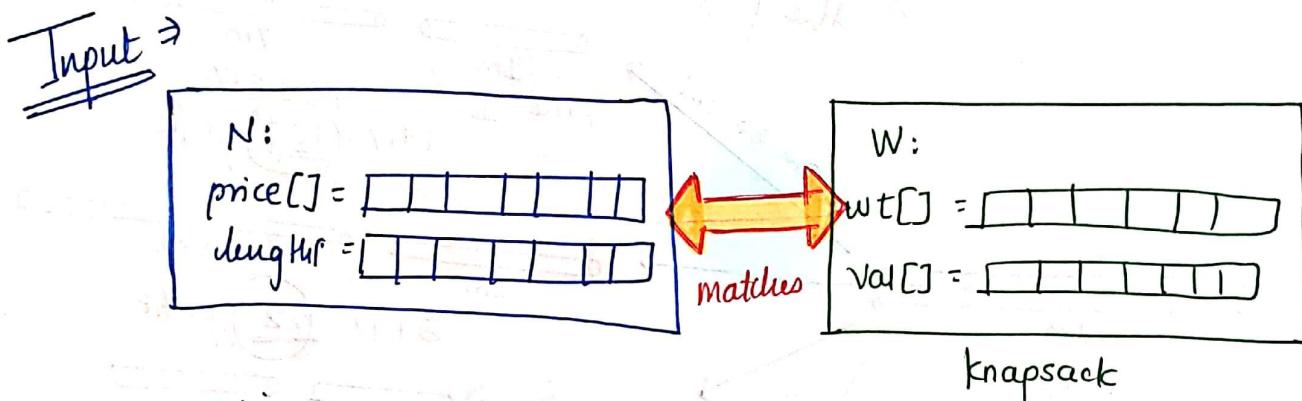
simulation:

We have to maximize the profit.

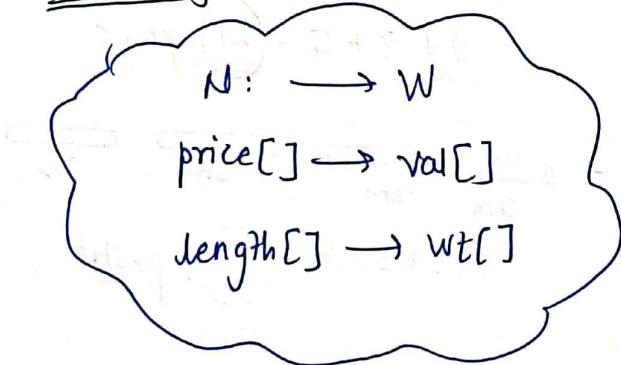


Stiññni ti affini duz nos voy. 'N' dñgnal te bor o neviq ero soj  
 voy ti ; nos voy don't duz neviq ai yoro sing . wmit te . on  
 to sing soft slot nos voy nent 'i' dñgnal affin bor soft duz  
 length  $\rightarrow$  1 to N . Dñgnal mñt sing dont  
 price  
 $N \rightarrow$  length of rod

Sometimes the length array is not given in the question then  
 we have to make the length array by pushing 1 to N  
 elements in the length array



matching



Here, we can cut rod ~~the~~ in same lengths like

3 , 3 , 2      1 , 1 , 6

]  $\rightarrow$  So unbounded  
knapsack  
Variation

Code

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

## Coin Change - I

coin [] = 

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
|---|---|---|

  
Sum = 5

Infinite Supply of coins is there.

$$\begin{aligned}2+3 &= 5 \\2+2+1 &= 5 \\1+1+3 &= 5 \\1+1+1+2 &= 5 \\1+1+1+1+1 &= 5\end{aligned}$$

5 ways

Now we have to count the no. of ways in which we can obtain the given sum

→ Knapsack pattern

But in knapsack, two arrays (wt & val) were given then whenever an array is given then consider it as wt array  
only

### MATCHING

coins [] → wt []  
Sum → W

→ This is unbounded knapsack cause repetition is allowed.

## Subset Sum

→ This question is related to SubsetSum problem

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 5 |
|---|---|---|---|

$$\text{sum} = 8$$

{1, 2, 5}

True ✓  
(or)  
False

## Subset sum

if ( $\text{arr}[i-1] \leq j$ )

$$t[i][j] = t[i-1][j] \text{ || } t[i-1][j - \text{arr}[i]];$$

else

$$t[i][j] = t[i-1][j];$$

## Count of Subset Sum

if ( $\text{arr}[i-1] \leq j$ )

$$t[i][j] = t[i-1][j] + t[i-1][j - \text{arr}[i-1]];$$

else

$$t[i][j] = t[i-1][j];$$

## Code

## Max no. of ways (coin change)

if ( $\text{coins}[i-1] \leq j$ ) {

$$t[i][j] = t[i-1][j] + t[i-1][j - \text{coins}[i-1]];$$

} else {

$$t[i][j] = t[i-1][j];$$

}

## Coin Change - II

find the minimum number of coins

$$\text{coins}[] = [1 \ 2 \ 3]$$

$$\text{Sum} = 5$$

$$\# \text{coins} = 2 \leftarrow 2+3 = 5$$

$$\# \text{coins} = 5 \leftarrow 1+1+1+1+1 = 5$$

$$\# \text{coins} = 4 \leftarrow 1+1+1+2 = 5$$

$$\# \text{coins} = 3 \leftarrow 1+1+3 = 5$$

$$\# \text{coins} = 3 \leftarrow 2+2+1 = 5$$

$\} \Rightarrow$  Hence minimum 2 coins are required.

$$\text{dp}[\text{coins.size()} + 1][\text{sum} + 1]$$

$$\text{dp}[4][6]$$

|  |  | 0 | 1 | 2 | 3 | 4 | 5 | Sum           |
|--|--|---|---|---|---|---|---|---------------|
|  |  | 0 | 0 | 0 | 0 | 0 | 0 | (INT_MAX - 1) |
|  |  | 1 | 0 | 0 | 0 | 0 | 0 |               |
|  |  | 2 | 0 | 0 | 0 | 0 | 0 |               |
|  |  | 3 | 0 | 0 | 0 | 0 | 0 |               |

u ↓      Assign 0

coin[] : {}      Sum = 1      so we can't make this situation

TWIST  
is there

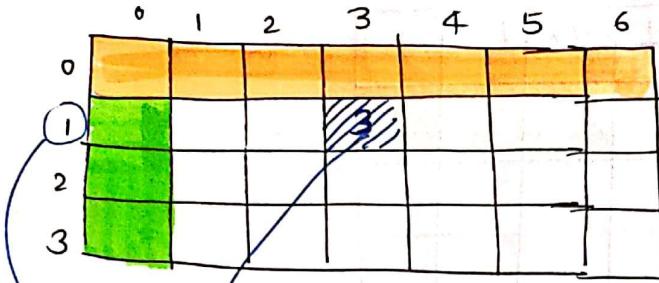
means coins array is empty  
so we require infinite number  
of coins which sums up to  
a sum value

but we can't take infinity in that

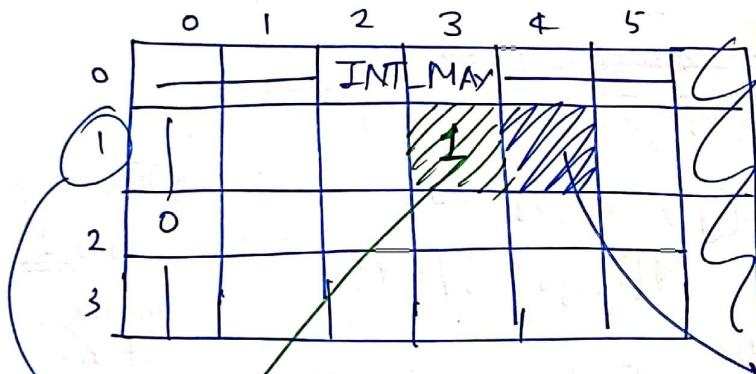
so store  $\boxed{\text{INT\_MAX} - 1}$

Why -1?

let's talk after this



$\text{arr} = [1]$   
 $\text{sum} = 3 \rightarrow \text{no. of min wins}$



If  $\text{arr} = [3 | 5 | 2]$

but length = 1

$\boxed{3}$   
 $\text{sum} = 3 \rightarrow \text{so 1 coin required}$

if  $\text{arr} = [3 | 5 | 2]$   
 $\text{sum} = 4$

size = 1  
 $\text{arr} = \boxed{3}$

$\text{sum} = 4$

min. no. of coins of denomination "3"  
 such that we get sum = 4

(NOT POSSIBLE)

so  $(\text{INT\_MAX} - 1)$

|   | 0 | 1           | 2 | (j)<br>3 | 4 | 5 |
|---|---|-------------|---|----------|---|---|
| i | 0 | INT_MAX - L |   |          |   |   |
|   | 0 |             |   |          |   |   |
|   | 1 |             |   |          |   |   |
|   | 2 |             |   |          |   |   |
|   | 3 |             |   |          |   |   |

if ( $j \leq arr[i] = 0$ ) → put 1

if ( $j \leq arr[i] \neq 0$ ) put INT\_MAX - 1

~~for (int i=0 ; i<n+1 ; i++)~~

```
for (int j=1 ; j<sum+1 ; j++)
    if ( $j \leq arr[0] == 0$ )
         $t[i][j] = j / arr[0]$ 
    else
         $t[i][j] = INT\_MAX - 1$ 
```

→ This is for row 1  
means if  $i=1$

Now the real code  $i=2$  and  $j=1$

```
for (int i=2 ; i<n+1 ; i++) {
    for (int j=1 ; j<sum+1 ; j++) {
        if ( $arr[i-1] \leq j$ ) {
             $t[i][j] = \min(t[i][j - coins[i-1]], t[i-1][j])$ 
        } else {
             $t[i][j] = t[i-1][j];$ 
        }
    }
}
```

→ New code

```

for(int i=2 ; i<n+1 ; i++) {
    for(int j=1 ; j<sum+1 ; j++) {
        if(coins[i-1] <= j)
            dp[i][j] = min(dp[i][j - coins[i-1]] + 1,
                            dp[i-1][j])
        else
            dp[i][j] = dp[i-1][j]
    }
}

```

For this only, we have taken  
INT\_MAX - 1

Why INT\_MAX - 1?

cause 1 is get added then

$$INT\_MAX - 1 + 1 = INT\_MAX$$



But what if we take only INT\_MAX instead  
of INT\_MAX - 1

If INT\_MAX

and If we add +1

INT\_MAX + 1

and this value ~~is~~ can't be  
able to store in integer  
so that's why we have  
to take INT\_MAX - 1.

# Longest Common Subsequence

## • patterns •

- ① Longest Common Substring
- ② Print longest Common Subsequence
- ③ Shortest Common supersequence
- ④ Print shortest common supersequence
- ⑤ minimum number of deletions and insertions  $a \rightarrow b$
- ⑥ Longest repeating subsequence
- ⑦ Length of longest subsequence of 'a' which is a substring in 'b'.
- ⑧ Subsequence Pattern Matching
- ⑨ Count how many times  $\textcircled{a}$  appear as subsequence in  $\textcircled{b}$ .
- ⑩ Longest palindromic Subsequence.
- ⑪ Longest palindromic Substring
- ⑫ Count of palindromic substring
- ⑬ Minimum no. of deletions in a string to make it a palindrome.
- ⑭ Minimum no. of Insertions in a string to make it a palindrome.

## • Longest Common Subsequence

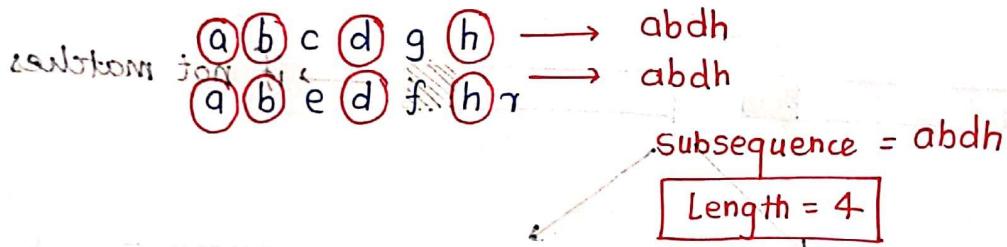
morphic solution

Recursive :-

Two strings will be given.

string  $x = "abcdgh"$  size( $n$ ) = 6

string  $y = "abedfhr"$  size( $m$ ) = 7



### Recursive Solution

- ① Base Condition
- ② Choice Diagram

### # Base Condition :-

if the length of both string is zero

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

subsequence with empty string

length 0

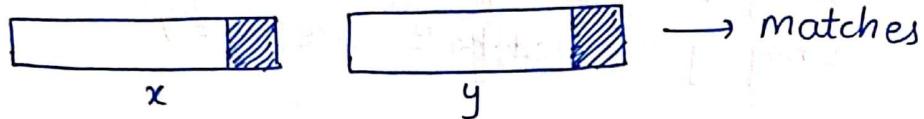
with ref used via empty

initially subsequence

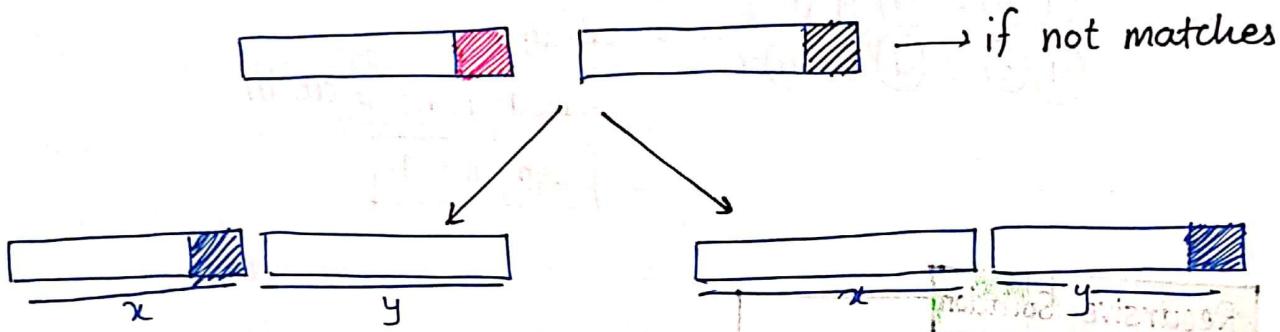
|     |              |
|-----|--------------|
| 1-n | ith char ref |
| 1-m |              |

## Choice Diagram

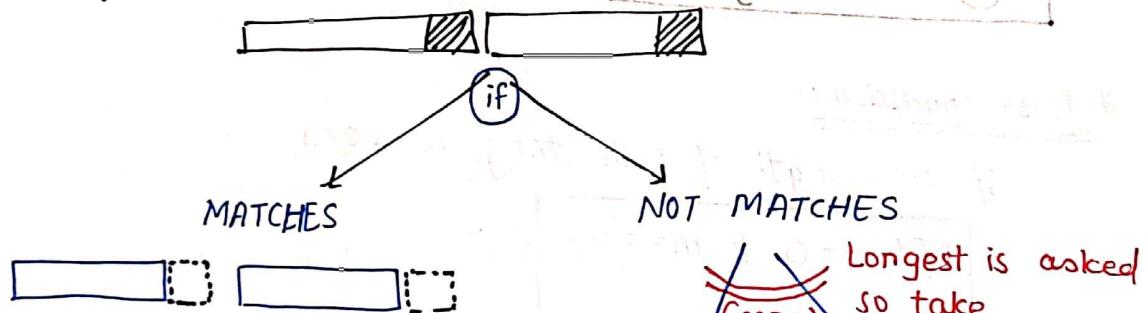
$\text{if}(x[n-1] == y[m-1]) \rightarrow \text{if the last character matches}$



else



## choice Diagram:



Remove the matching part  
count++;

Means now call for the  
recursive function  
for length  $\boxed{n-1 \\ m-1}$

~~Longest is asked  
(max)~~  
so take max



### Code

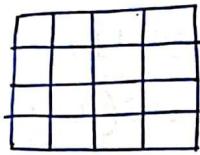
```
int lcs(string x, string y, int n, int m){  
    if(n==0 || m==0) return 0;  
    if(x[n-1] == y[m-1]) {  
        return 1 + lcs(x, y, n-1, m-1);  
    }  
    else{  
        return max(lcs(x, y, n-1, m),  
                   lcs(x, y, n, m-1));  
    }  
}
```

## LCS Memoization :-

Why do we need this?

To store the result of Subproblems Hence memoization is used.

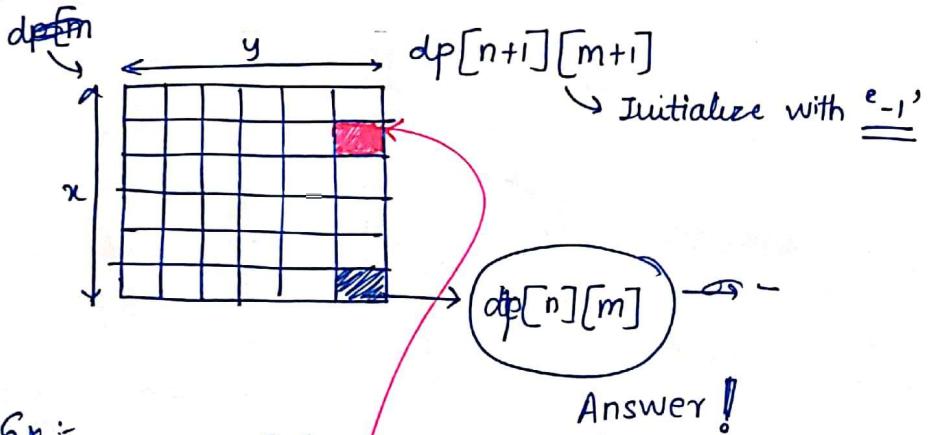
Recursive call +



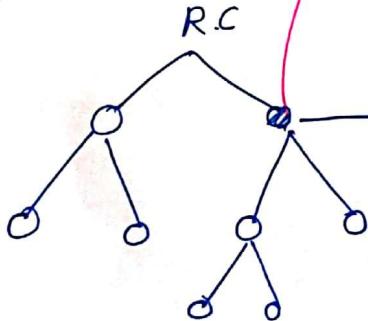
This table will be formed only by using the variables of the problem that are changing

→ In ~~LCS~~ LCS, lengths of strings are changing

$m, n \rightarrow$  ARE CHANGING



Ex:-



Here, we will check whether this function is already called or not  
if it has called  
HOW TO CHECK?

check whether the value is present in the table or not.

## Code

qu-motto

```
// globally declaration of the table  
static int dp[100][100];  
  
int LCS(string x, string y, int n, int m){  
    if (dp[n][m] != -1) {  
        return dp[n][m];  
    }  
  
    if (x[n-1] == y[m-1]) {  
        dp[n][m] = 1 + LCS(x, y, n-1, m-1);  
    }  
    else {  
        dp[n][m] = max(LCS(x, y, n, m-1),  
                        LCS(x, y, n-1, m));  
    }  
    return dp[n][m];
```

## Bottom - Up

base condition       $\xrightarrow{\text{Initialization in bottom-up}}$   
 In recursion

$\text{if}(m == 0 \text{ || } n == 0) \rightarrow \text{zero Initialization}$

~~dp[0]~~

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 |   |   |   |
| 0 |   |   |   |
| 0 |   |   |   |
| 0 |   |   |   |

$x : a b c f \rightarrow m = 4$

$y : a b c d a f \rightarrow n = 6$

$\text{dp}[5][7] \rightarrow \text{dp}[m+1][n+1]$

$\xrightarrow{n(y.\text{length}())}$

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 |   |   |   |   |   |   |
| 0 |   |   |   |   |   |   |
| 0 |   |   |   |   |   |   |
| 0 |   |   |   |   |   |   |

↓

$m$   
 $(x.\text{length}())$

This is  
the answer

// Initialization

$\text{dp}[m+1][n+1];$

```
for (int i=0 ; i<m+1 ; i++)
    for (int j=0 ; j<n+1 ; j++)
        if (i==0 || j==0)
            dp[i][j] = 0;
```

```

for(int i=1 ; i<m+1 ; i++) {
    for(int j=1 ; j<n+1 ; j++) {
        if (x[i-1] == y[j-1]) {
            dp[i][j] = 1 + dp[i-1][j-1]
        }
        else {
            dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
        }
    }
}
return dp[m][n];

```

## Longest Common Substring

Two strings are given:-

$$S_1 = "abcde"$$

$$S_2 = "abfce"$$

In subsequence, ~~not~~ subsequence can be found by skipping some characters between the string.

Ex

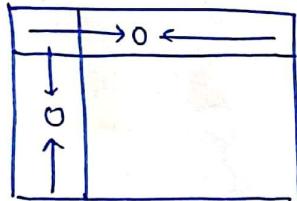
$S_1 = \underline{\underline{abcde}}$  → abce → abce  
 $S_2 = \underline{\underline{abfce}}$  → abce → abce  
 ↓  
 skipped

But in substring, the answer should be continuous.

$S_1 = [\underline{\underline{ab@de}}] \rightarrow e$  ] These are the substrings  
 $S_2 = [\underline{\underline{abf@e}}] \rightarrow ab$

Longest Common Substring = ab

// Initialization



```
for(int i=0 ; i<m+1 ; i++)
  for(int j=0 ; j<n+1 ; j++)
    if(i==0 || j==0)
      dp[i][j]=0
```

// main code

```
for(int i=1 ; i<m+1 ; i++)
  for(int j=1 ; j<n+1 ; j++)
    if(S1[i-1] == S2[j-1])
      dp[i][j] = dp[i-1][j-1] + 1;
    else
      dp[i][j] = 0;
```

## • Print Longest Common Subsequence

string  $s_1 = "acbcf"$ ;

string  $s_2 = "abcdaf"$ ;

Longest Common Subsequence = "abcf"

Task is to print the LCS of two strings.

How exactly LCS works?

$s_1 = "acbcf" \rightarrow m = 5$

$s_2 = "abcdaf" \rightarrow n = 6$

$dp[m+1][n+1]$

$\xrightarrow{a} ab \xrightarrow{c} abc$

|        |   | $\xrightarrow{n}$ (size of $s_2$ ) |   |   |   |   |   |   |
|--------|---|------------------------------------|---|---|---|---|---|---|
|        |   | 0                                  | 1 | 2 | 3 | 4 | 5 | 6 |
| $\phi$ | 0 | 0                                  | 0 | 0 | 0 | 0 | 0 | 0 |
|        | 1 | 0                                  | 1 | 1 | 1 | 1 | 1 | 1 |
| a      | 2 | 0                                  | 1 | 1 | 2 | 2 | 2 | 2 |
| b      | 3 | 0                                  | 1 | 2 | 2 | 2 | 2 | 2 |
| c      | 4 | 0                                  | 1 | 2 | 3 | 3 | 3 | 3 |
| f      | 5 | 0                                  | 1 | 2 | 3 | 3 | 3 | 4 |

$m$   
(size of  $s_1$ )

At  $i=0, j=0 \rightarrow s_1[i] = s_2[j] \Rightarrow a == a$

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

At  $i=0, j=1 \rightarrow s_1[i] \neq s_2[j]$

so  $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$

|             | $\emptyset$ | a | b | c | d | a | f |
|-------------|-------------|---|---|---|---|---|---|
| $\emptyset$ | 0           | 0 | 0 | 0 | 0 | 0 | 0 |
| a           | 0           | 1 | 1 | 1 | 1 | 1 | 1 |
| c           | 0           | 1 | 1 | 2 | 2 | 2 | 2 |
| b           | 0           | 1 | 2 | 2 | 2 | 2 | 2 |
| c           | 0           | 1 | 2 | 3 | 3 | 3 | 3 |
| f           | 0           | 1 | 2 | 3 | 3 | 3 | 4 |

Not matched

Matched

Here f,f are same so we will go diagonally

"fcba"

Reverse()

"abcf"

This is our answer

Now a & c are not equal  
so find max(3,2)  
and move to that cell, 3 is max  
so move left

This process we have to follow

Summary

if Equal  $(i,j)$

$i--$   
 $j--$

if Not Equal  $(i,j)$

$\max(i-1, j)$   
 $i, j-1$

## Steps!

- ① prepare a table for LCS
- ② Now we have to start from the last cell

so int i=m, j=n;

```
while (i>0 && j>0){  
    if (s1[i-1] == s2[j-1]) {  
        ans += s1[i-1];  
        i--; j--;  
    }  
    else {  
        if (dp[i][j-1] > dp[i-1][j]) {  
            j--;  
        }  
        else  
            i--;  
    }  
}  
reverse(ans.begin(), ans.end());  
return ans;
```

## Shortest Common Supersequence

string a: "geek"

string b: "eke"



merge

merge them in such a way that

geelce

both a & b strings are present

This is the shortest length

Ex:-2 :-

a: "AGGTAB"

b: "GXTXAYB"

AGGTAB GXTXAYB

AGGTABGXTXAYB → Supersequence

Find the shortest Supersequence (length)

#

a: "AGGTAB"

b: "GXTXAYB"

AGGTAB is present  
GXTXAYB → present  
GXTXAYB → present

This is the shortest supersequence

length = 9

a: AGGTAB  
b: GXTXAYB

3G are there

G G

only 1 has taken

G A T B  
G A T B → Write once

AGGXTXAYB



GTAB

Longest Common Subsequence



What would be the worst case of making supersequence :-

a: AGGTAB

b: GXTXAYB

$\overbrace{\text{AGGTAB}}^m + \overbrace{\text{GXTXAYB}}^n$

LCS = "GTAB"

LCS = "GTAB"

AGGTABGXTXAYB

worst case

One LCS can be removed

IDEA

Length of ~~shorter~~  
Supersequence

$$= m+n$$

$$= m+n - \text{length of LCS}$$

Shortest length of  
Supersequence.

```

int LCS(string a, string b, int m, int n) {
    int dp[m+1][n+1];
    for(int i=0; i<m+1; i++) {
        for(int j=0; j<n+1; j++) {
            if(i==0) dp[i][j] = 0;
            if(j==0) dp[i][j] = 0;
        }
    }

    for(int i=1; i<m+1; i++) {
        for(int j=1; j<n+1; j++) {
            if(a[i-1] == b[j-1]) {
                dp[i][j] = 1 + dp[i-1][j-1];
            } else {
                dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
            }
        }
    }

    return dp[m][n];
}

int main() {
    string a, b;
    cin >> a >> b;
    cout << a.length() + b.length() - LCS(a, b, a.length(), b.length());
}

```

- Minimum no. of deletion & insertion required to convert string  $a$  to  $b$

Input:-  
 $a$  : "heap"  
 $b$  : "pea"

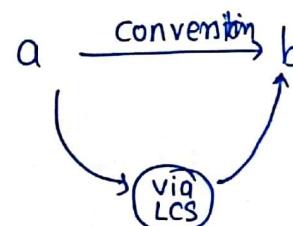
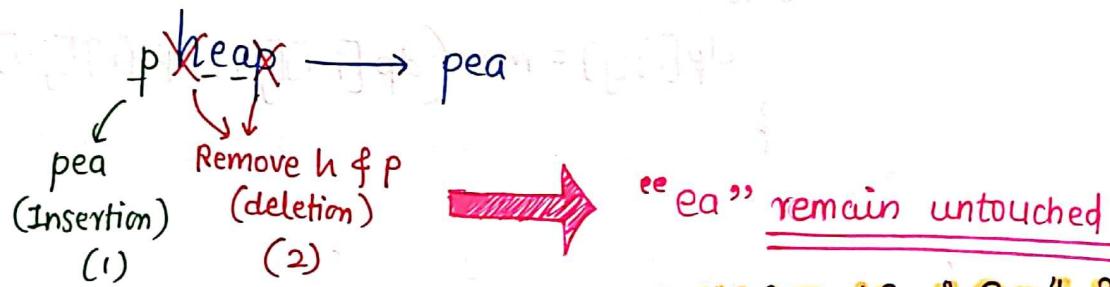
Output:  
 Insertion: 1  
 Deletion: 2

$a \xrightarrow{\text{convert}} b$   
 heap                    pea

How to think if we should apply LCS or not?

Two strings are given & optimal answer is required  
 then it is a variation of LCS problem.

→ This is an variation of LCS problem



WHAT IS "ea"?

Longest Common Subsequence

We will do not jump directly  
on conversion

~~heap~~ → pea

2 deletion from 'a' to convert to LCS

1 insertion in 'ea' to convert to 'b'

ea

points formed of common characters & suitable for minimum

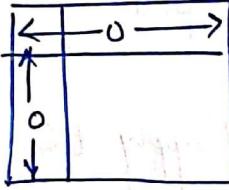
d of p

Code :-

LCS (string a, string b, int m, int n)

dp[m+1][n+1];

// Initialization



// Main code

```
for(int i=1 ; i<m+1; i++) {  
    for(int j=1 ; j<n+1; j++) {  
        if(a[i-1] == b[j-1]) { dp[i][j] = 1 + dp[i-1][j-1]; }  
        else {  
            dp[i][j] = max(dp[i-1][j], dp[i][j-1]);  
        }  
    }  
}
```

return dp[m][n];

}

main () {

// Input a, & b strings

```
cout << "Deletions" << a.length() - LCS(a,b,m,n);  
cout << "Min. Insertions" << b.length() - LCS(a,b,m,n);  
cout << endl;
```

= }

## Longest Palindromic Subsequence

One string is given, you have to return the longest palindromic subsequence.

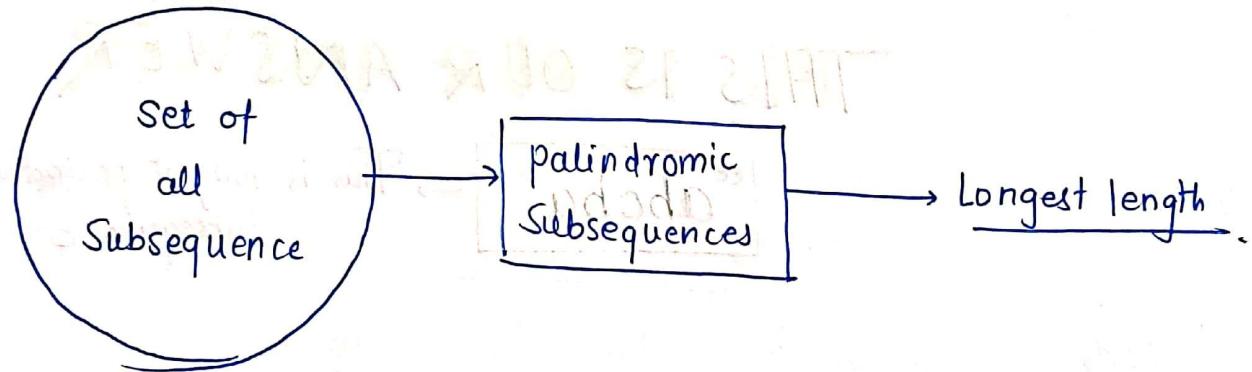
S: "agbcba"

In this, abcba

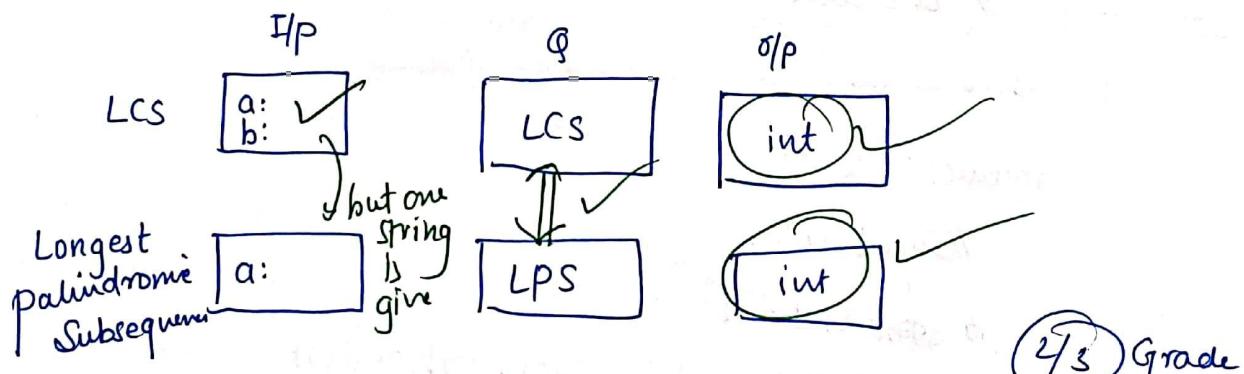
O/P = 5

This is Longest palindromic subsequence.

- find all subsequences of string "S"
- then find out how many subsequences are palindrome
- return the max. length of all palindromic subsequence.

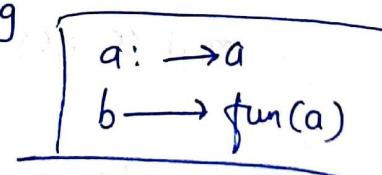


## Matching Algorithm :-



we can think of like this another string "b" is a function of a string

So LCS can apply



longest palindromic subsequence

String a = "abca gbc ba"

→ lets reverse this and store it in b

a = "a g b c b a"

b = "a b c b g a"

"abcba" → This is LCS

AND GUESS WHAT???

THIS IS OUR ANSWER

"abcba"

→ This is longest palindromic Subsequence

### Code

```
LCS (a, b, m, n) {
```

```
    // LCS code
```

```
}
```

```
main() {
```

```
    // Input "a";
```

```
    #include <iostream>
```

```
    string str1 = a;
```

```
    reverse(str1.begin(), str1.end());
```

```
    b = str1;
```

```
    cout << LCS(a, b, m, n) << endl;
```

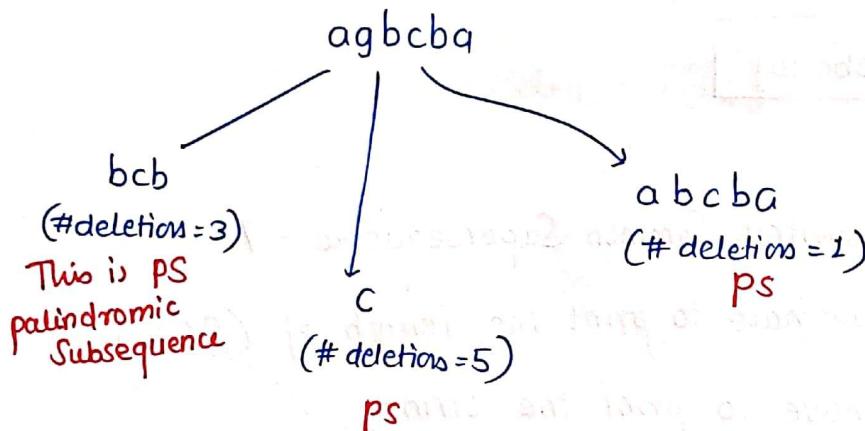
```
}
```

• Minimum no. of deletions required to make a string palindrome

A string requires minimum deletions to become a palindrome.

I/P:  $S = "agbcba"$

O/P = 1



Think ?

As we are deleting the maximum characters, we are getting shortest palindromic subsequence,

Hence,

Length of palindromic Subsequence  $\propto \frac{1}{\text{number of deletions}}$

Simply

Find the length of longest palindromic Subsequence.

return  $(\text{length of String}) - (\text{length of longest palindromic Subsequence})$

points to solution of longest common subsequence problem

## print Shortest Common SuperSequence

Ip: a: "acbcf"  
b: "abcdaf"

O/P: "acbcdaf"

→ length of Shortest Common SuperSequence = 7

previously we have to print the length of (SCS)

Now we have to print the string.

|   |  | a | b | c | d | a   | f   |
|---|--|---|---|---|---|-----|-----|
|   |  | 0 | 0 | 0 | 0 | 0   | 0   |
| a |  | 0 | 1 | 1 | 1 | 1   | 1   |
| b |  | 0 | 1 | 1 | 2 | 2   | 2   |
| c |  | 0 | 1 | 2 | 3 | (3) | (3) |
| f |  | 0 | 1 | 2 | 3 | 3   | 4   |

match ✓

Here a f c are f ✓

not equal then

move to max ( $dp[i-1][j]$ ,  $dp[i][j-1]$ )

but add the character of lower cell

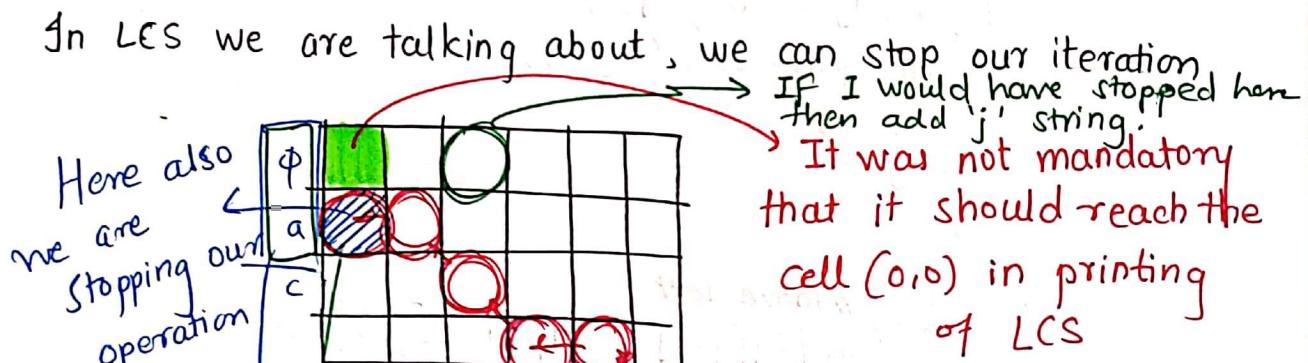
Here, After taking 'f' it will move to left

but before moving add the character  $s_2[j-1]$  to ans and  
then Let's code this portion ⇒

```

string ans = "";
int i = a.length();
int j = b.length();
while(i > 0 && j > 0) {
    if(a[i-1] == b[j-1]) {
        ans.push_back(a[i-1]);
        i--;
        j--;
    }
    else {
        if(dp[i][j-1] > dp[i-1][j]) {
            ans.push_back(b[j-1]);
            j--;
        }
        else if(dp[i][j-1] < dp[i-1][j]) {
            ans.push_back(a[i-1]);
            i--;
        }
    }
}

```



If we stop here, means ( $j=0$ )  
then we have to add the remaining ~~of~~ 'i' string

Let's take an example

a: "ac"

b: " "

Then LCS = " "

but in SCS = "ac"

cause in SCS we take aggregation

so

```

while ( i > 0 ) {
    ans.push_back ( a [ i - 1 ] );
}
while ( j > 0 ) {
    ans.push_back ( b [ j - 1 ] );
}

```

# Important

## Final Code

- ① prepare a table for LCS
- ② Now we have to start from last cell

so  $i = a.length() , j = b.length()$

- ③ string ans = "";

```

while ( i > 0 && j > 0 ) {
    if ( a [ i - 1 ] == b [ j - 1 ] ) {
        ans.push_back ( a [ i - 1 ] );
        i--;
        j--;
    } else {
        // move left
        if ( dp [ i ] [ j - 1 ] > dp [ i - 1 ] [ j ] ) {
            ans.push_back ( b [ j - 1 ] );
            j--;
        } else if ( dp [ i ] [ j - 1 ] < dp [ i - 1 ] [ j ] ) {
            ans.push_back ( a [ i - 1 ] );
            i--;
        }
    }
}

```

}

```
while (i > 0) {  
    ans.push_back(a[i - 1]);  
    i--;  
}
```

```
while (j > 0) {  
    ans.push_back(b[j - 1]);  
    j--;  
}
```

↳ Last element of array is not included in answer

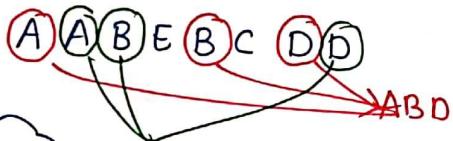
→ If we want to include last element of array in answer  
→ Then we have to change condition of loop  
→ Instead of  $i > 0$  we can write  $i \geq 0$   
→ Instead of  $j > 0$  we can write  $j \geq 0$

## Longest Repeating Subsequence

String  $S = "AABEBCDD"$

Subsequence → order (should be maintain)  
discontinuous (✓)

Let's take a subsequence = "ABD"



"You can't reuse the characters once it's taken"

"ABD" occurs = 2 times

"Ac" → 2 times

Answer is "ABD"

means the length is 3.

Output the length of longest Repeating Subsequence

Given string is

copy  $S_1: "AABEBCDD"$ :

$S_2: "AABEBCDD"$ ; → find LCS

AAB EBC DD  
↓  
This will never come in longest Repeating Subsequence  
Why?

AABBDD  
ABD      ABD  
↓      ↓  
OUR Answer

Index  
 $E \rightarrow 3$  (in both string)

$C \rightarrow 5$  (in both strings)

If (letters come at the same index then we never consider them)

Then why AA is taken?

?

A A B E B C D D  
A A B E B C D D

WH?

$S_1 \Rightarrow A \rightarrow \begin{matrix} 0 \\ 1 \end{matrix}$   
 $S_2 \Rightarrow A \rightarrow \begin{matrix} 0 \\ 1 \end{matrix}$

Take the cross

Now we can take, Right??

YES

so

Now code it  $\Rightarrow$

LCS

if( $a[i-1] == b[j-1]$  ff  $i \neq j$ ) {

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

}

else {

$dp[i][j] = \max(dp[i][j-1],$   
 $dp[i-1][j]);$

}

cause we don't want  
index same

## Sequence Pattern Matching

Input  $\Rightarrow$

$a = "AXY"$   
 $b = "ADXCPY"$

output

T/F

Subsequence

order preserved ✓

is, "a" is a subsequence of string

'b'.

(ADXC PY)  $\rightarrow$  AXY  $\rightarrow$  True

So simply we can find the length of LCS in string a & b  
if length of LCS == length of A  $\rightarrow$  return true  
else return  $\rightarrow$  false.

But how length will decide?

There is a possibility of getting the LCS different.

a: AXY (3)

b: ADXCPY (6)

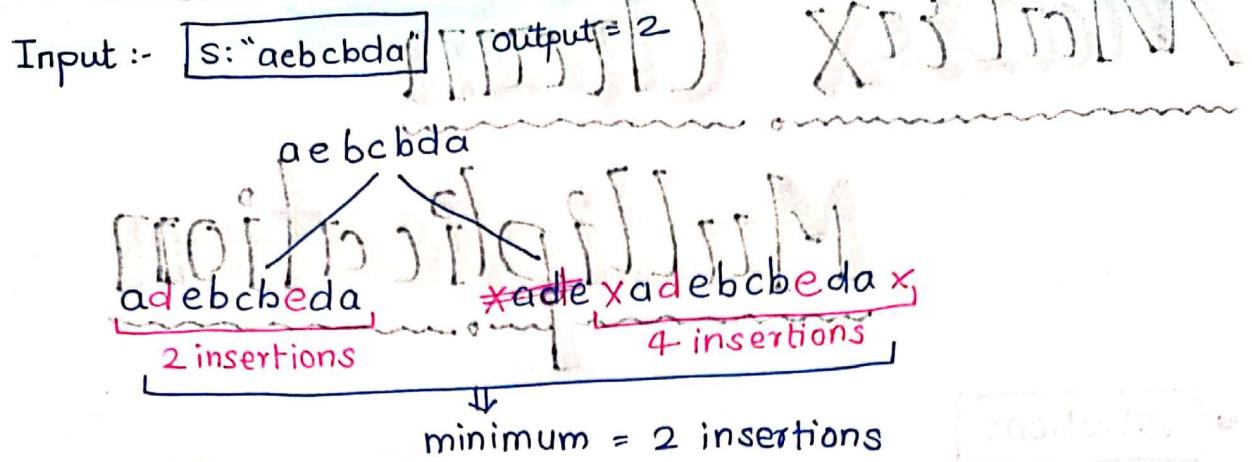
Then LCS ( $0 \rightarrow \min(3, 6)$ )  $\rightarrow$  length range

$[0 - 3]$   $\Rightarrow$  Hence length will be sufficient

Code

```
for(int i=1 ; i<m+1 ; i++) {  
    for(int j=1 ; j<n+1 ; j++) {  
        if(a[i-1] == b[j-1]) {  
            dp[i][j] = 1 + dp[i-1][j-1];  
        }  
        else {  
            dp[i][j] = max(dp[i-1][j], dp[i][j-1]);  
        }  
    }  
    int LCS = dp[m][n];  
    if(LCS == m)  $\rightarrow$  return true;  
    else  $\rightarrow$  return false;
```

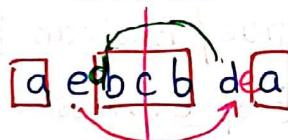
- Minimum no. of deletions Insertion in a string to make it palindrom



$s = "aebcbda"$

↓  
chang to LPS  
Longest palindromic Subsequence

↓  
 $abcba \rightarrow e f d$  are removed



LPS में transform करने के लिए  
one 'e' can added  
one 'd' can added

So eventually ,  $\# \text{deletions} = \# \text{insertions}$  WOW!!

⇒ same code as the min. no. of deletions

↳ Matrix Chain Multiplication

# Matrix Chain

## Multiplication

### • Variations

- ① Matrix Chain Multiplication (MCM) recursive
- ② MCM memoization
- ③ MCM bottom-up
- ④ Printing MCM
- ⑤ Evaluate expression to true/ Boolean parenthesization
- ⑥ Minimum/maximum value of expression
- ⑦ palindromic Partitioning
- ⑧ Scramble String
- ⑨ Egg Dropping problem

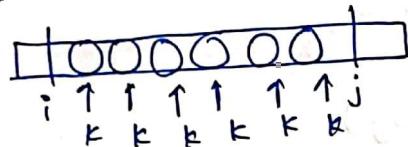
## Matrix Chain Multiplication

↳ Interview

### MCM format

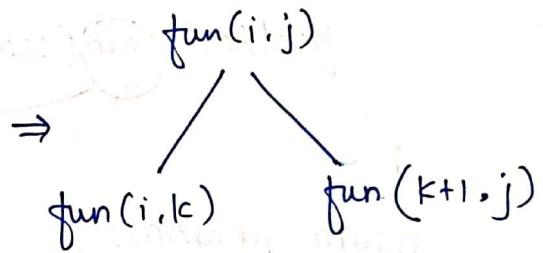
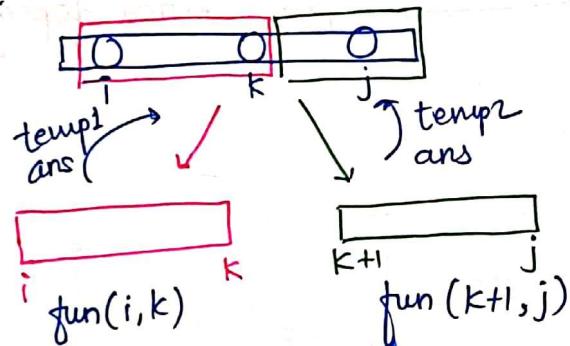
- ① string or array will be given.
- ② You will get the feeling of breaking the array/string while solving the problem.

Ex:-



k will move i to j

Ex:-



③

जो भी temp ans मिलेगा उस पर एक function लगेगा फू वे will get our answer.

## #format:-

```
int solve(int arr[], int i, int j){  
    if(i > j) → This may be different for other questions  
        return 0;  
  
    for(int k=i ; k < j ; k++) {  
        //Calculate temporary answer  
        tempAns = solve(arr, i, k)  
        + Solve(arr, k+1, j);  
  
        finalAns = fun(tempAns);  
    }  
    return finalAns;  
}
```

## MCM

### problem statement

Array is given

$$\text{arr[]} = \{40, 20, 30, 10, 30\}$$

matrices are given

$$\begin{array}{cccc}
 A_1 & A_2 & A_3 & A_4 \\
 \downarrow & \downarrow & \downarrow & \downarrow \\
 \text{dimension } (2 \times 5) & (20 \times 20) & (30 \times 10) & \dots \quad (\text{Any dimension can be there})
 \end{array}$$

we have to multiply the matrices such that number of multiplications should be minimum.

$$\begin{bmatrix} & & \end{bmatrix}_{2 \times 3} \times \begin{bmatrix} & & \\ & & \end{bmatrix}_{3 \times 6} \Rightarrow \begin{array}{l} \text{2nd matrix } 2 \times 3 \\ \text{1st matrix } 3 \times 6 \end{array} \Rightarrow \begin{array}{l} \text{order } a \times b \cdot b \times d \\ \underline{\text{order } a \times d} \end{array}$$

↓

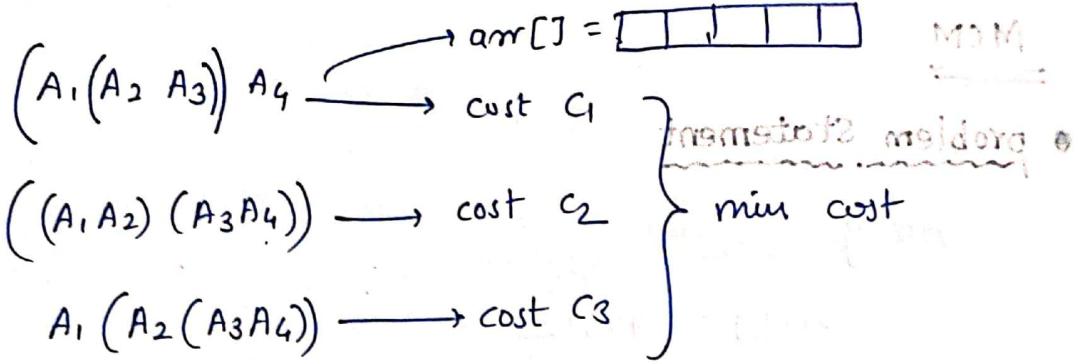
$$\begin{bmatrix} & \end{bmatrix}_{2 \times 6}$$

Min. cost =

$$\begin{array}{c}
 2 \times 3 \quad 3 \times 6 \\
 + \quad \downarrow \quad \downarrow \\
 2 * 3 * 6
 \end{array}$$

$$= 6 * 6$$

$$= 36$$



Ex:-

$$\text{mat. A} = 10 * 30$$

$$\text{mat. B} = 30 * 5$$

$$\text{mat. C} = 5 * 60$$

$$\underline{(A \cdot B) C} =$$

$$AB = 10 \times 30 = 30 \times 5$$

$$= 10 \times 30 \times 5$$

$$AB = 150 \times 10 = 1500$$

$$(AB)C = 10 \times 5 = 5 \times 60$$

$$= 1500 + 10 \times 5 \times 60$$

$$= 1500 + 3000$$

$$\underline{\underline{(AB)C} = 4500}$$

$$\underline{\underline{A(Bc)}}$$

$$BC = \frac{30 \times 5}{30 \times 5 \times 60} = \frac{5 \times 60}{300 \times 30} = \underline{\underline{9000}}$$

$$A(Bc) = \frac{10 \times 30}{10 \times 30 \times 60} = \frac{30 \times 60}{300 \times 30} = \underline{\underline{9000}}$$

$$= 18000 + 9000$$

$$\underline{\underline{A(Bc) = 27000}}$$

minimum cost return करना चाहे।

100  
50  
3000

$$arr[] = \{ \underset{0}{40}, \underset{1}{20}, \underset{2}{30}, \underset{3}{10}, \underset{4}{30} \} \quad n=5$$

then  $(n-1)$  matrices will be given

$$A_1 \rightarrow 40 \times 20$$

$$A_2 \rightarrow 20 \times 30$$

$$A_3 \rightarrow 30 \times 10$$

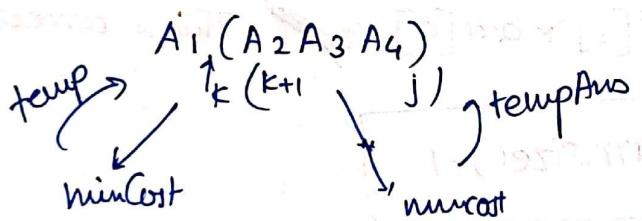
$$A_4 \rightarrow 10 \times 30$$

$$\Rightarrow A_i \rightarrow arr[i-1] * arr[i]$$

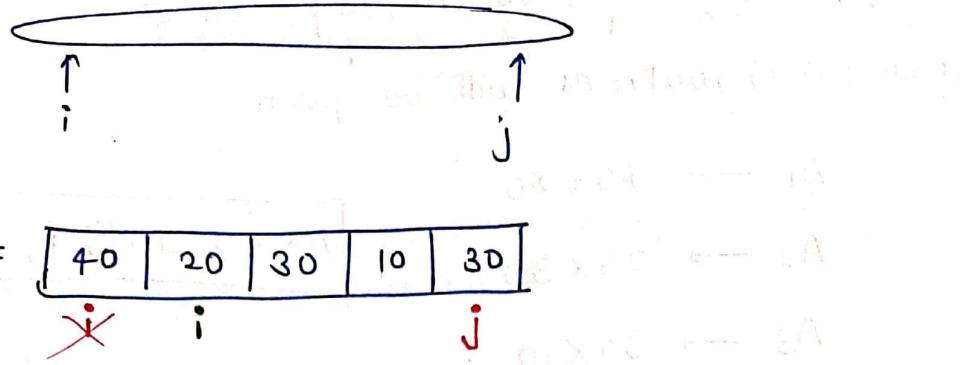
How to identify? whether this problem is based on the format given by aditya verma!

Cause here we have to put brackets

$$\begin{aligned} & (A_1(A_2 A_3)) A_4 \\ & (A_1 A_2)(A_3 A_4) \\ & A_1(A_2(A_3 A_4)) \end{aligned} \quad \left. \begin{array}{l} \text{so we are putting the brackets} \\ \text{in such a way that each matrix} \\ \text{is multiplied with its next matrix} \end{array} \right\}$$



$\Rightarrow$  Select minimum cost



matrix

$$A_i = arr[i-1] * arr[i]$$

$$\begin{aligned} A_0 &= arr[0-1] * arr[0] \\ &= arr[-1] * arr[0] \quad \rightarrow (-1) \text{ index} \end{aligned}$$

is not valid

Hence we can't take "i" at position 0  
so we have to take i from  $i=1$

And Now check 'j'

$$\begin{aligned} A_j &\rightarrow arr[j-1] * arr[j] \\ &\rightarrow arr[4-1] * arr[4] \\ &\rightarrow arr[3] * arr[4] \quad \checkmark \text{ This is correct.} \end{aligned}$$

So

$$i=1, j=arr.size() - 1$$

```
int solve( int arr[], int i, int j ) {
```

// Base Condition

```
if( i >= j ) {
```

```
    return 0;
```

```
}
```

if  $i = j$

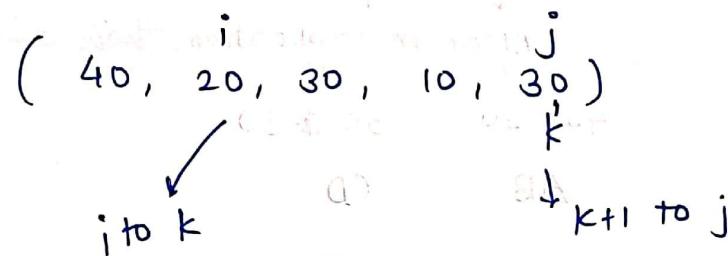
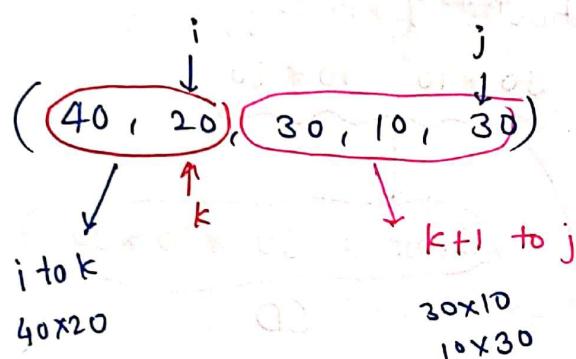


means only 1 element

~~arr~~  $A_i = arr[i-1] * arr[i]$  in array

$A_1 = arr[-1] * arr[0]$

This is invalid so  
array should be at least of  
size 2



40x20  
20x30  
30x10  
10x30

(EMPTY SET) !

so  $(k = j - 1)$

तक चलाना  
पडेगा

so  $k = i + 1 \text{ to } k = j - 1$

```
for( int k=i ; k<j-1 ; k++ ) {
```

```
    solve( arr, i, k );
```

```
    solve( arr, k+1, j );
```

40, 20, 30, 10, 30  
i      k      j

fun( i to k )

40 \* 20    20 \* 30

AB

fun( k+1 to j )

30 \* 10    10 \* 30

CD

$$\text{minCost} = 40 * 20 * 30$$

$$\text{minCost} = 30 * 10 * 30$$

After multiplication, their dimension

$$40 * 30 \quad 30 * 30$$

AB

CD

Where is 40 ?

$(i-1)$  ⚡

This 30?

→ at k

cost → This cost we have to calculate manually

Where is this 30 ?

$(j)$  ⚡

⇒

So formula =  $\text{arr}[i-1] * \text{arr}[k] * \text{arr}[j]$   
Extra Cost

## Final Code

(MEM) optimization

```
int solve (int arr[], int i, int j){  
    if(i >= j) return 0;  
    int mini = INT_MAX;  
    for(int k=i ; k<=j-1 ; k++) {  
        int tempAns = (solve(arr, i, k)  
                      +  
                      solve(arr, k+1, j))  
                     - (arr[i-1] * arr[k] * arr[j]);  
        mini = min(mini, tempAns);  
    }  
    return mini;  
}
```

## Memoization (MCM)

i, j → constraints

// Initialize dp[size of array + 1][size of array + 1] → -1.

// check if

(dp[i][j] != -1)

return dp[i][j];

int static dp[1001][1001];

int Solve(int arr[], int i, int j) {

if (i >= j)

return 0;

if (dp[i][j] != -1)

return dp[i][j];

int mini = INT\_MAX;

for (int k = i ; k < j ; k++) {

int tempAns = solve(i, k, arr, i, k) +

solve(arr, k+1, j) +

(arr[i-1] \* arr[k] \* arr[j]);

mini = min(tempAns, mini);

}

dp[i][j] = mini;

return dp[i][j];

{

int main() {

memset(dp, -1, sizeof(dp));

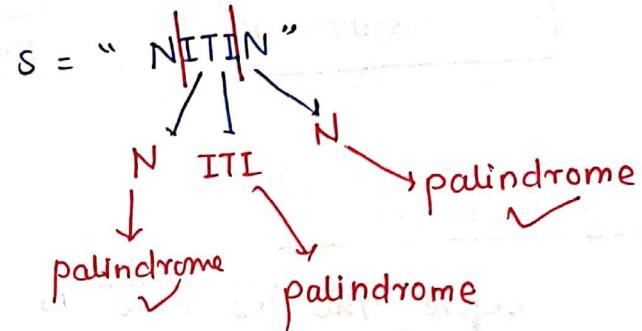
}

## • Palindrome Partitioning

Given a string  
make partition such that the string in each partition should  
be palindrome.

$$s = \text{"pavap"}$$

pavap



We have to find the minimum partitions.

① In worst case, we can make  $(n-1)$  partitions

n|t|t|n  
→ 4 partitions

Steps:-

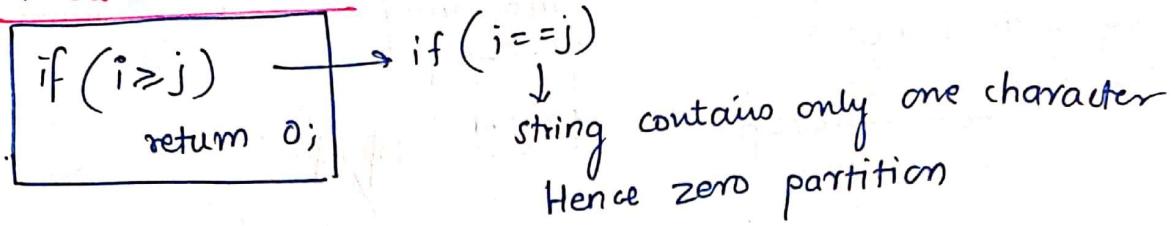
- 1.) find i & j
- 2.) Base Condition
- 3.) find 'k' loop
- 4.) Apply fun(tempAns);

~~int solve(string s, int i, int j) {~~

① // find i & j

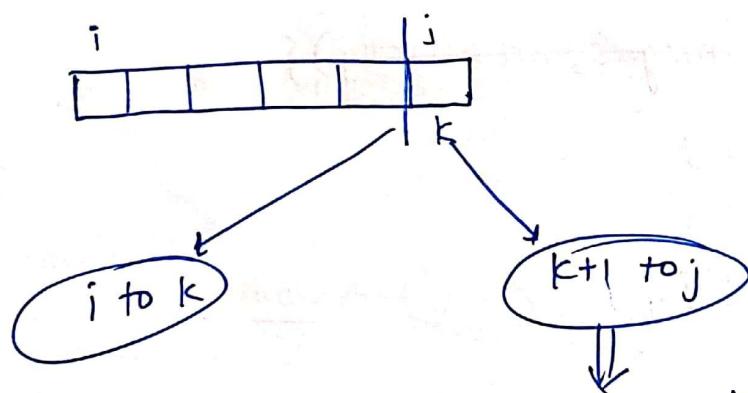
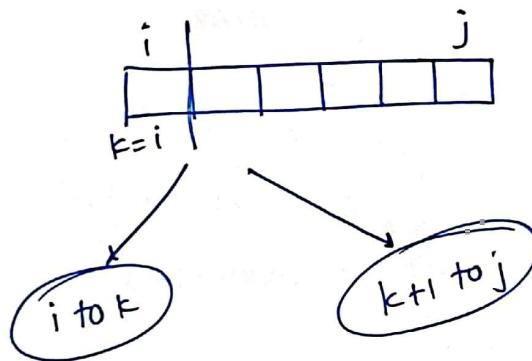
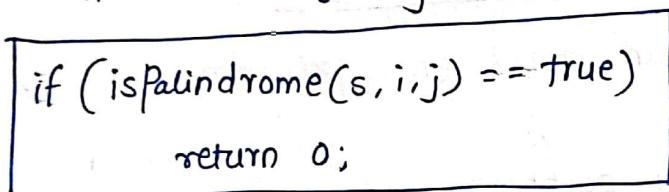
i n i t i n j  
i=0                       $\circlearrowleft i=0 \rightarrow \text{works}$   
j=n-1

② find base condition



③ find 'k' loop

before this we have to check, after partitioning if we are getting the palindromic string or not



THIS IS NOT  
POSSIBLE

so  $k$  should be till  $j-1$  primitive bin partitioning

```
for (int k=i ; k<=j-1 ; k++) {
```

```
    int tempAns = solve(s, i, k) +  
        solve(s, k+1, j)
```

+

$\downarrow \leftarrow i$

```
    mini = min(mini, tempAns);
```

```
}
```

```
return mini;
```

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| n | i | t | l | i | n |
|---|---|---|---|---|---|

"n"

"itin"

cost  $c_1$

cost  $c_2$

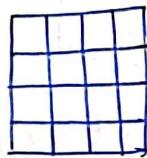
Along with this we  
are doing one partition

so add 1

## • Palindromic Partitioning memoization

s = "nitik"

memoization → R.C. +



In the recursive solution, i and j are changing



$0 \leq i, j \leq 1000$

so  $dp[100][100]$ ;

```
int static dp[100][100];
int solve(string s, int i, int j) {
    if(i <= j) return 0;
    if(ispalindrome(s, i, j)) return 0;
    if(dp[i][j] != -1)
        return dp[i][j];
    int mini = INT_MAX;
    for(int k=i ; k<=j-1 ; k++) {
        int tempAns = solve(s, i, k) +
                      solve(s, k+1, j) +
                      1;
        mini = min(mini, tempAns);
    }
    dp[i][j] = mini;
    return dp[i][j];
}
int main() {
    memset(dp, -1, sizeof(dp));
}
```

## Optimized memoization

In the for loop, we are calling two recursive calls

but what if we check the left part →  $\text{solve}(s, i, k)$

means we'll check if the left part has the result then we can store it.

else we'll call the ~~solve(s, i, k)~~  $\text{solve}(s, i, k)$

and store the result of it in left and left part stored in dp.

```
if (dp[i][k] != -1)
    left = dp[i][k];
```

```
else
    left = solve(s, i, k);
    dp[i][k] = left
```

```
if (dp[k+1][j] != -1)
```

```
    right = dp[k+1][j];
```

```
else
    right = solve(s, k+1, j);
    dp[k+1][j] = right
```

```
int tempAns = { 1 + left + right; }
```

This is  
the most  
optimized.

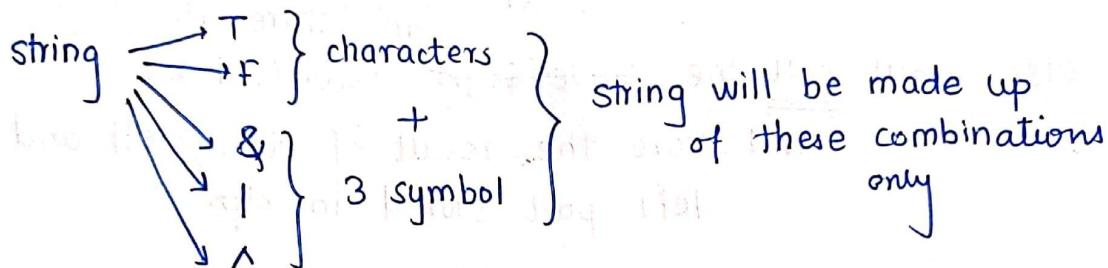


## Evaluate Expression to True Boolean Parenthesization

string  $s = "T \text{ or } F \text{ and } T"$



$s = "T \mid F \& T"$



Now task is to find the number of ways such that we can insert brackets and resulting expression would be true.

$s = "T \mid F \& T"$

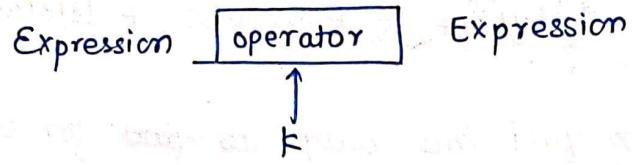
$$\begin{array}{ccc} & \downarrow & \\ (T \mid F) \& T & T \mid (F \& T) \\ = T \& T & = T \mid F \& T \\ = T & & = T \end{array}$$



$$\begin{array}{c} (T) \mid (F \& T \wedge F) \\ \uparrow k \\ (T \mid F) \& (T \wedge F) \end{array} \quad \left. \begin{array}{l} \{ \\ \} \\ \{ \end{array} \right\} \begin{array}{l} k \text{ is moving} \\ \text{with} \\ k = k + 2 \end{array}$$

$$(T) | (F \& T \wedge F)$$

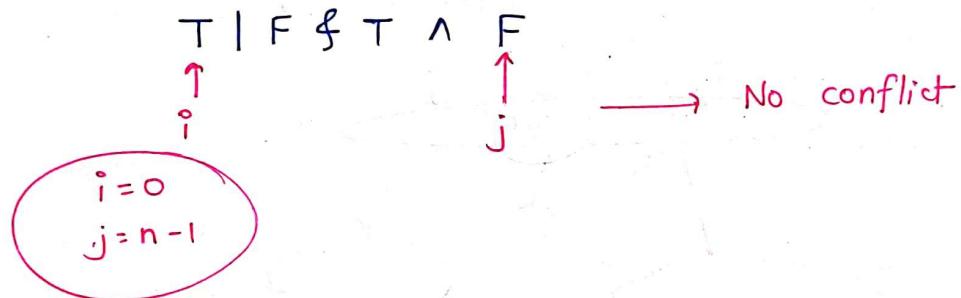
↑  
k



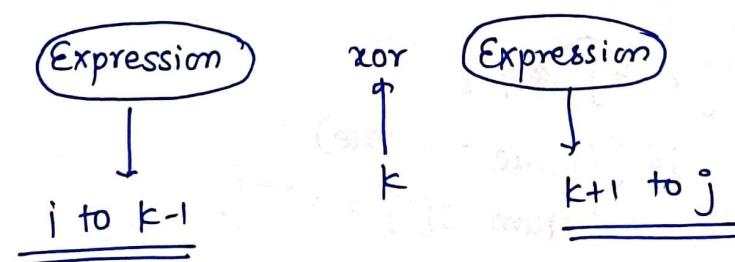
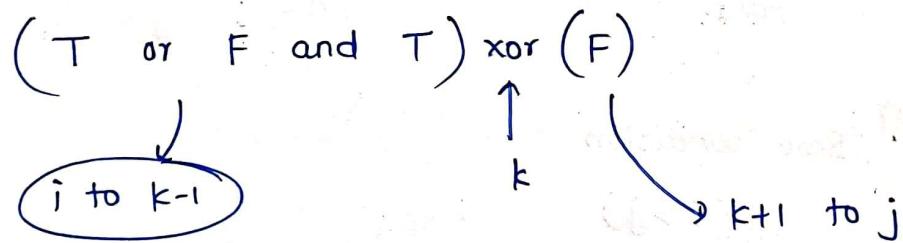
means, k will always be operator

4 steps :-

① find i & j :-



② Base Condition :-



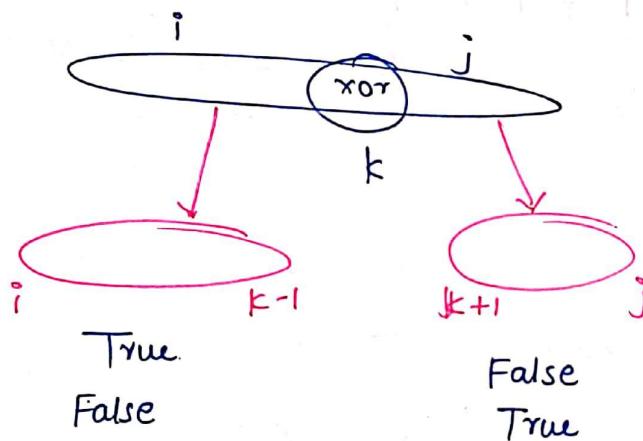
$\boxed{\text{exp1}} \quad \boxed{\text{xor}} \quad \text{exp2}$

No. of ways of 'True'  $\Rightarrow$

$$(\text{leftTrue} * \text{RightFalse} + \text{LeftFalse} * \text{RightTrue})$$

Means we have to find the ways ~~to find~~ for exp1 true as well as false.

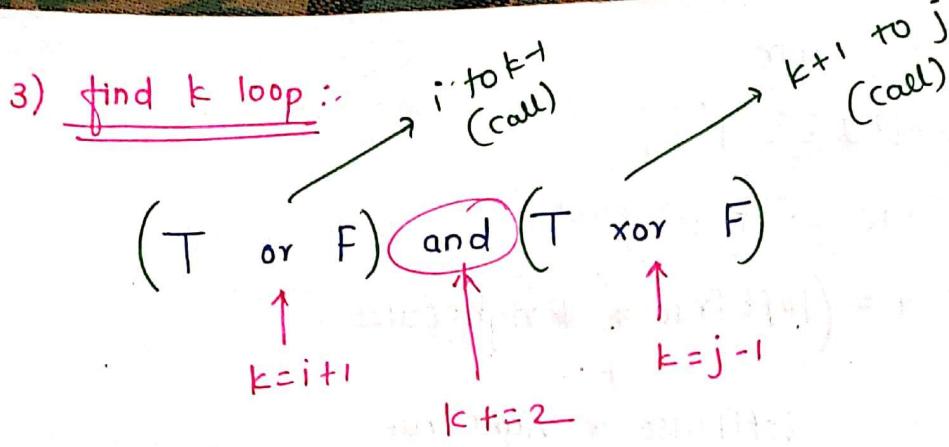
$\boxed{\text{exp1}} \quad \boxed{\text{xor}} \quad \text{exp2}$



// Base condition

```
if(i > j)
    return false
if(i == j) {
    if(isTrue == true)
        return s[i] == 'T';
    else
        return s[i] == 'F';
```

}



```

for(int k=i+1; k<=j-1; k+=2){
    int leftTrue = Solve(s, i, k-1, T);
    int leftFalse = Solve(s, i, k-1, F);
    int rightTrue = Solve(s, k+1, j, T);
    int rightFalse = Solve(s, k+1, j, F); } → TempAns.
    
```

// for and operator

```
if (s[k] == '&') {
```

// No. of ways for True

```
if (isTrue == true) {
```

```
ans += leftTrue * rightTrue;
```

} ← TempAns & continue

```
else {
```

```
ans += (leftTrue * rightFalse)
```

```
+ rightTrue * leftFalse
```

```
+ leftFalse * rightTrue);
```

For True

LT RT

For False

LT RF

LF RT

LF RF

}

```

// for or operator
else if (s[k] == '|') {
    if (isTrue == true) {
        ans += (leftTrue * rightFalse
                +
                leftFalse * rightTrue
                +
                leftTrue * rightTrue);
    }
    else {
        ans += leftFalse * rightFalse;
    }
}

// for xor operator
else if (s[k] == '^') {
    if (isTrue == true) {
        ans += (leftTrue * rightFalse
                +
                leftFalse * rightTrue);
    }
    else {
        ans += (leftTrue * rightTrue
                +
                leftFalse * rightFalse);
    }
}

return ans;

```

### • memoized Version

Whenever we are creating a DP, ~~1D~~-table then the ~~num~~ dimension of the table depends on the number of variables changing in the function call

Important

In the recursive code :-

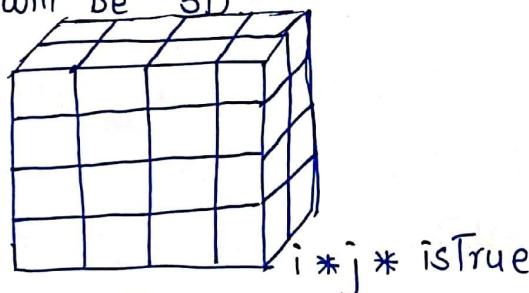
```
for( int k = i+1 ; k <= j-1 ; k++ ) {  
    int leftTrue = Solve(s, i, k-1, true);  
    int rightFalse = Solve(s, i, k-1, false);  
    int rightTrue = solve(s, k+1, j, true);  
    int rightFalse = solve(s, k+1, j, false);  
}
```

# No. of variables = 4 ✓  
Solve (s, i, j, isTrue)

It's not changing  
changing

(3 variables are changing)

So matrix will be 3D



i \* j \* isTrue

let's suppose constraints are :-  $0 \leq s \leq 1000$

```
int dp[100][100][2]
```

But we have another better option:-

Create a map

| map          | value |
|--------------|-------|
| "i j isTrue" |       |
| "50 90 F"    | 9     |
| "5 40 T"     | 4     |

} like this

key = i + " " + j + " " + isTrue

//create a global map

unordered\_map<string, int> mp;

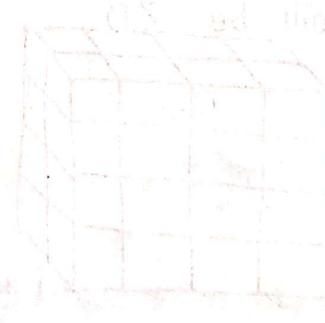
int main()

//clear the map

mp.clear();

Solve(), → call the function

}



[1][2][3][4] [5][6][7][8]

### Code :-

```
int Solve(string s, int i, int j, bool isTrue) {  
    if(i > j) return true;  
    if(i == j) {  
        if(isTrue == true)  
            return s[i] == 'T';  
        else  
            return s[i] == 'F';  
    }  
  
    // New code (map)  
  
    string temp = to_string(i);  
    temp.push_back(" ");  
    temp.append(to_string(j));  
    temp.push_back(" ");  
    temp.push_back(to_string(isTrue));  
  
    if(mp.find(temp) != mp.end())  
        return mp[temp];  
  
    int ans=0;  
  
    for(int k=i+1; k<=j-1; k+=2){  
        int leftTrue = solve(s, i, k-1, true);  
        int leftFalse = solve(s, i, k-1, false);  
        int rightTrue = solve(s, k+1, j, true);  
        int rightFalse = solve(s, k+1, j, false);  
    }  
}
```

key creation  
i + " + j + " + isTrue

// For '&' operator

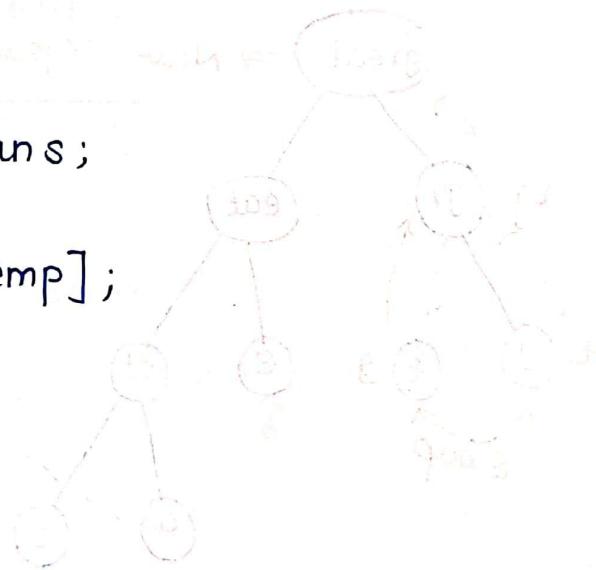
```
if (s[k] == '&') {  
    if (isTrue == true) {  
        ans += leftTrue * rightTrue;  
    }  
    else {  
        ans += ((leftTrue * rightFalse) +  
                (leftFalse * rightTrue) +  
                (rightFalse * leftFalse));  
    }  
}
```

// for '|'

```
operator  
else if (s[k] == '|') {  
    if (isTrue == true) {  
        ans += ((leftTrue * rightFalse) +  
                (leftFalse * rightTrue) +  
                (leftTrue * rightTrue));  
    }  
    else {  
        ans += (leftFalse * rightFalse);  
    }  
}
```

// For '^' operator

```
else if ( s[k] == '^' ) {  
    if (isTrue == true) {  
        ans += ((leftTrue * rightFalse)  
                +  
                (leftFalse * rightTrue));  
    }  
    else {  
        ans += ((leftFalse * rightFalse)  
                +  
                (leftTrue * rightTrue));  
    }  
}  
  
mp[temp] = ans;  
  
return mp[temp];  
}
```



int toAsmit (string str) recursive int ab and sw  
return function to return done bits

## Scrambled String

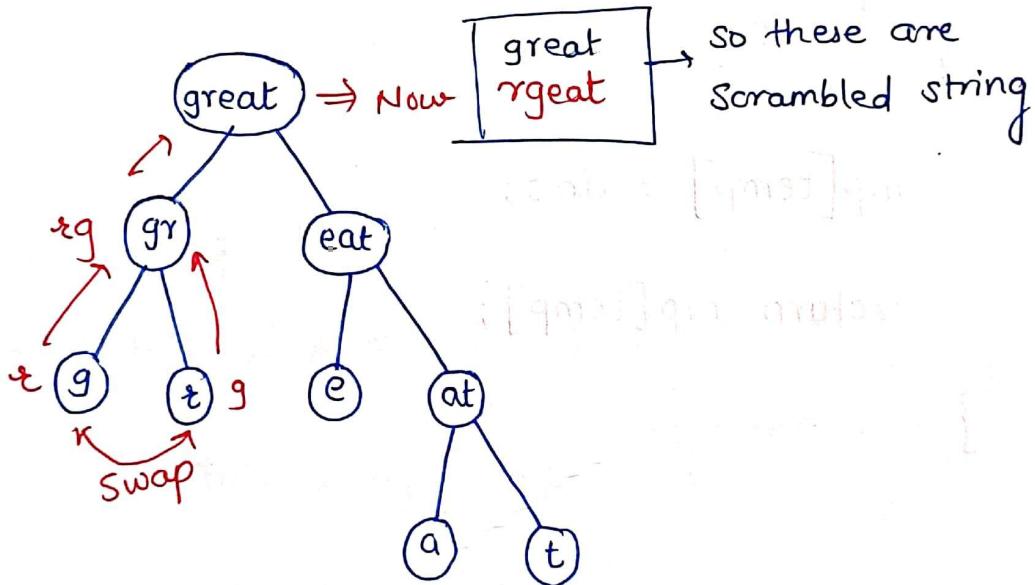
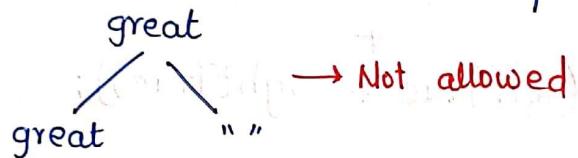
### # problem Statement:-

Given two strings 'a' and 'b', check whether they are Scrambled string or not.

I/p = a: "great" o/p = True  
b: "rgeat"

### What is Scrambled string?

- ① Create a binary tree.
- ② You cannot make child of binary tree empty



You can break down in any way.  
we can do the swapping (zero or more) times of the child ~~not~~ nodes of non-leaf nodes

great

gr eat

gre at

grea t

great

Not allowed

so length  $\Rightarrow$

$i=1$   
to  
 $i=n-1$

so as we are breaking at each length  
this is MCM problem

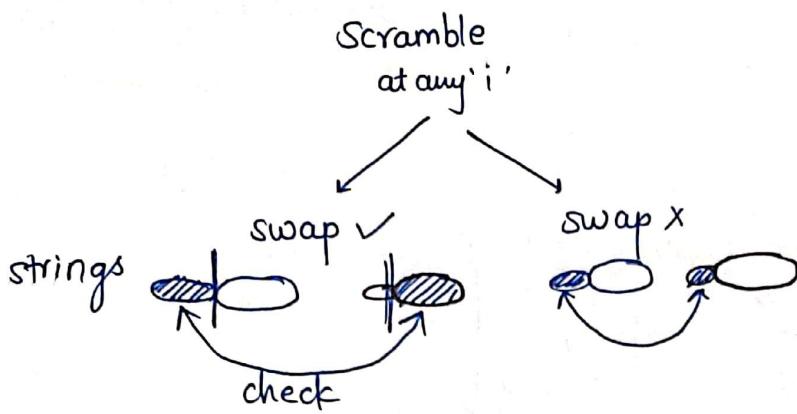
Approach :-

Scramble  
या तो swap होगा

या तो swap नहीं होगा

swap X  
swap ✓  
swap ✓

i = 2  
swap ✓  
swap X  
swap X



$a \rightarrow b$   
 $a(\text{last}) \rightarrow b(\text{first})$

$a(\text{first}) \rightarrow b(\text{first})$   
 $a(\text{last}) \rightarrow b(\text{last})$

Case-I

Swap ✓

$i=2$   
 $\begin{matrix} 0 & 1 & 2 & 3 & 4 \\ \text{g} & \text{r} & \text{e} & \text{a} & \text{t} \end{matrix}$

$\begin{matrix} 0 & 1 & 2 & 3 & 4 \\ \text{e} & \text{a} & \text{t} & \text{g} & \text{r} \end{matrix}$

bool Solve(a,b)

if(Solve(a.substr(0,i) , b.substr(n-i, i))) {  
 if solve(a.substr(i,n-i) , b.substr(0, n-i)) )  
 }

Condition-I

Case-II

$\begin{matrix} \text{g} & \text{r} & \text{e} & \text{a} & \text{t} \end{matrix}$   
 $\begin{matrix} \text{g} & \text{r} & \text{e} & \text{a} & \text{t} \end{matrix}$

$\begin{matrix} \text{g} & \text{r} & \text{e} & \text{a} & \text{t} \end{matrix}$   
 $\begin{matrix} \text{g} & \text{r} & \text{e} & \text{a} & \text{t} \end{matrix}$

Condition-II

if(solve( a.substr(0,i) , b.substr(0,i)) == true  
 if solve( a.substr(i,n-i) , b.substr( i,n-i) == true )

$\begin{matrix} \text{g} & \text{r} & \text{e} & \text{a} & \text{t} \end{matrix}$   
 $\begin{matrix} \text{g} & \text{r} & \text{e} & \text{a} & \text{t} \end{matrix}$

$\begin{matrix} \text{g} & \text{r} & \text{e} & \text{a} & \text{t} \end{matrix}$   
 $\begin{matrix} \text{g} & \text{r} & \text{e} & \text{a} & \text{t} \end{matrix}$

if (condition -I == true || condition -II == true)  
    return true

//Base Condition

great      ngreat w  
a              b

if (length diff)  
    return false

if (a & b are empty) → return true;

if (a.compare(b) == a) → return true; → This is Equal  
strings  
(a==b) ↗

if (a.length() <= 1)  
    return false;

Final Code

```
int main(){  
    string a, b;  
    cin >> a >> b;  
    → if(a.length() != b.length()) → return false;  
    → Both empty → return false  
    → Equal → return true  
    Solve(a,b);  
}
```

```
bool Solve(string a, string b) {
    if (a.compare(b) == 0) return true;
    if (a.length <= 1) return false;

    int n = a.length;
    bool flag = false;
    for (int i=1 ; i<n-1 ; i++) {
        if (conditionI || conditionII) {
            flag = true;
            break;
        }
    }
    return flag;
}
```

## Scrambled String memoized

Before → check map → if present → return value  
Calculate Subtree

After → Store value in map

```
bool solve(string a, string b){
```

```
    if(a.compare(b) == 0){  
        return true;  
    } if(a.length() <= 1){  
        return false;  
    }
```

global map से लें

unordered\_map<string, bool> mp

Changing variables → a & b

key → string = "a" + " " + "b"

string key = a; → "a"  
key.push\_back(' '); → "a\_"  
key.append(b); → "a\_b"

```
if(mp.find(key) != mp.end()) {  
    return mp[key];  
}
```

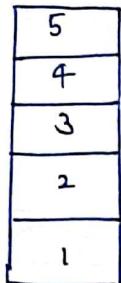
```
int n = a.length();  
bool flag = false;  
for(int i=1; i<n; i++) {  
    if(conditionI || conditionII){  
        flag = true;  
        break;  
    }  
    mp[key] = flag;  
}
```

return mp[key] = flag;

}

## Egg Dropping Problem

Given no. of eggs 'e' and number of floors 'f'



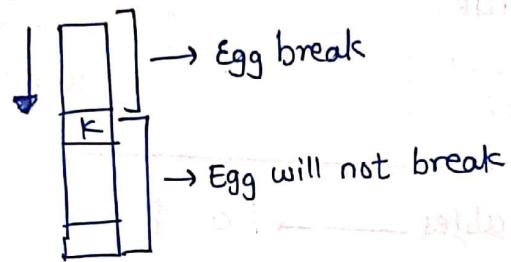
Example

|         |
|---------|
| $e = 3$ |
| $f = 5$ |

 $\rightarrow \text{o/p} = 3$

Find the critical floor such that egg will break

Building



We will move from top to bottom

then we have to find the minimum number of attempts required to identify the critical floor.

we have to use the eggs very wisely → minimize the number of attempts.

000 → minimum no. of attempts = 3  
wisely

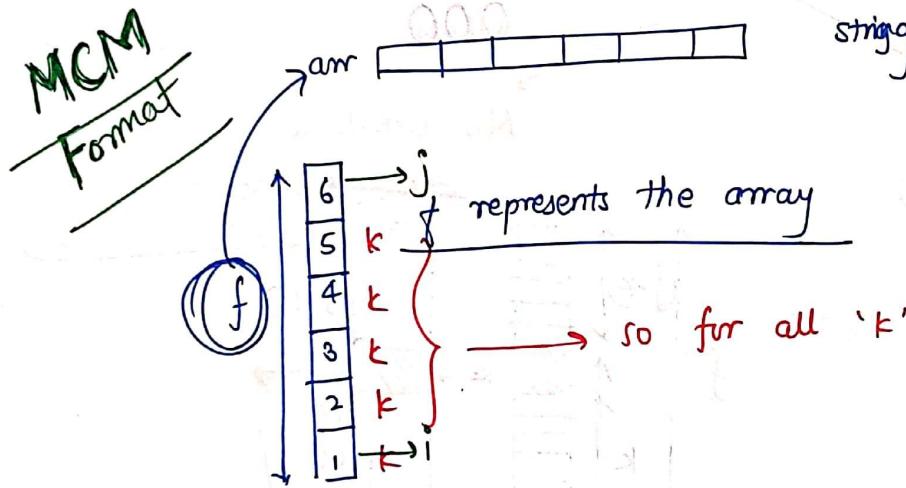
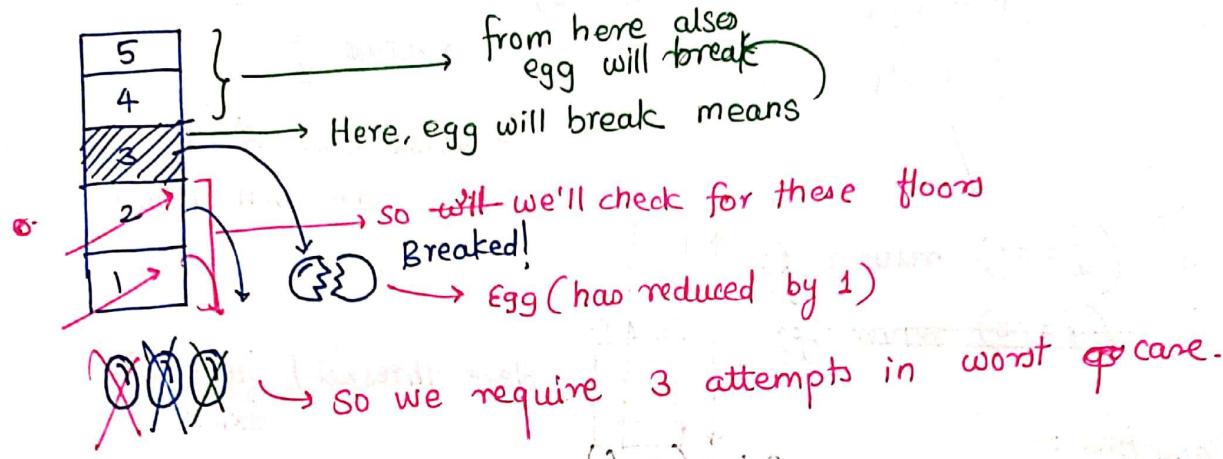
Best Technique/Strategy

So we can start from bottom floor

cause even if we drop the egg from bottom it will not break and we can use that egg again.

If  $e=1$  and 'f' floors are given  
then in worst case, 'f' attempts required

$$e=3 \quad f=5$$



- i) find i & j
- ii) Base Condition
- iii) k loop
- iv) ans  $\leftarrow$  temp ans

Now the main question is → from where we can start our dropping exercise

⇒ 'means' find the k loop

→ इसे जाएं तो loop लगा क्या देखेंगे!

for(k=1, k<=f ; k++)

## // Base Condition

think of the smallest valid input

$$\begin{array}{l} \text{I/P} \\ e \rightarrow 0/1 \\ f \rightarrow 0/1 \\ \downarrow \end{array}$$

if ( $e == 0$ )  $\rightarrow$  we never find the ans  
if ( $e == 1$ )  $\rightarrow$  return  $f$

↓ worst case में last floor  
तक जाना पड़ेगा।

$(f == 1)$  return  $f$ ;  
 ~~$(f == 0)$  return  $f$ ;  $f$~~

Solve( $e, f$ )

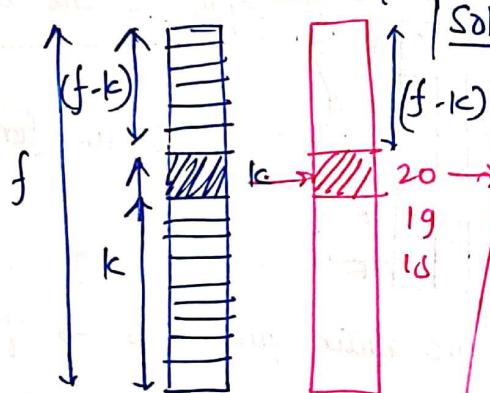
Here threshold floor will exist

TempAns :-

~~000~~  
Break ✓  
Solve( $e-1, k-1$ )

~~000~~

No break ✗



Suppose  $f=20$  पर break नहीं हो सकता  
so means 19, 18 में break होता करेगा

and this is for Sure

and अपने को first floor चाहिए कि जहाँ से  
egg break हो जाए।  
So ↑ तक जाना पड़ेगा।

int Solve ( int e, int f) {

// Base Condition

if (f == 0 || f == 1) return f;

if (e == 1) return f;

**TASK**

minimum no. of attempts  
in worst case

// k loop

int mn = INT\_MAX;

for (int k=1 ; k <= f ; k++) {

int temp = 1 + max (Solve (e-1, k-1),  
Solve (e, f-k));

in each iteration  
we are taking one attempt

why "max()"

cause we have to find in  
worst case

mn = min (mn, temp);

}

minimum no. of attempts.

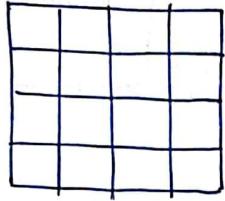
Don't  
get  
Confused :)

return mn;

}

## Egg Dropping Problem Memoization

matrix dimension: number of changing variable.



exf

as e and f are changing

// global declarations of table

int static ~~int~~ dp[101][101]  
e f

int solve(int e, int f, int dp[][]){

    if (f == 0 || f == 1) return f;

    if (e == 1) return f;

(: boundary)

    if (dp[e][f] != -1) {

        return dp[e][f];

}

    int mn = INT\_MAX;

    for (int k=1; k <= f; k++) {

        int temp = 1 + max(solve(e-1, k-1),  
                          solve(e, f-k));

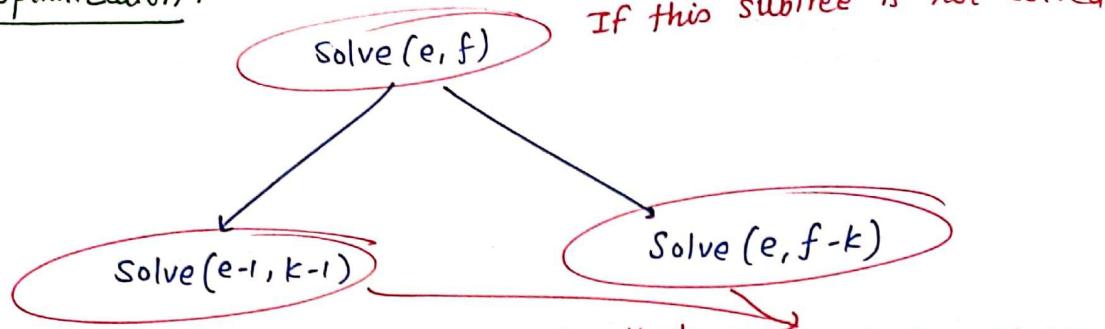
        mn = min(mn, temp);

}

    return dp[e][f] = mn;

{

//optimization :-



If this subtree is not solved

Then we were assuming that These both subtrees are not solved.

BUT

This is not necessarily true.

It can happen that  
one of them has been solved.

As it is (upper code)

```
for (int k=1 ; k<=f ; k++) {
    if (dp[e-1][k-1] != -1)
        int low = dp[e-1][k-1];
    else
        low = solve(e-1, k-1);
        dp[e-1][k-1] = low;

    if (dp[e][f-k] != -1)
        int high = dp[e][f-k];
    } else {
        high = solve(e, f-k);
        dp[e][f-k] = high;
    }

    int temp = 1 + max(low, high);
    mn = min(mn, temp);

}
return dp[e][f] = mn;
```