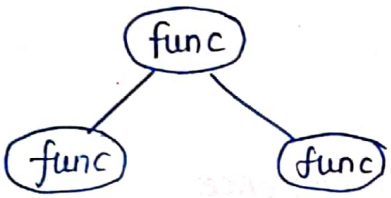
Dynamic Programming

-Pavan Patel

• Dynamic Programming :-

DP = Enhanced Recursion



function ~~calles~~ calls itself with smaller inputs.

→ parent of Dynamic Programming is recursion
→ DP asks for optimal solution
→ Choice / choose → Recursion can be applied
→

Recursive function	→ Memoize (Top-down)	→ ~~Top-down~~ Bottom-up

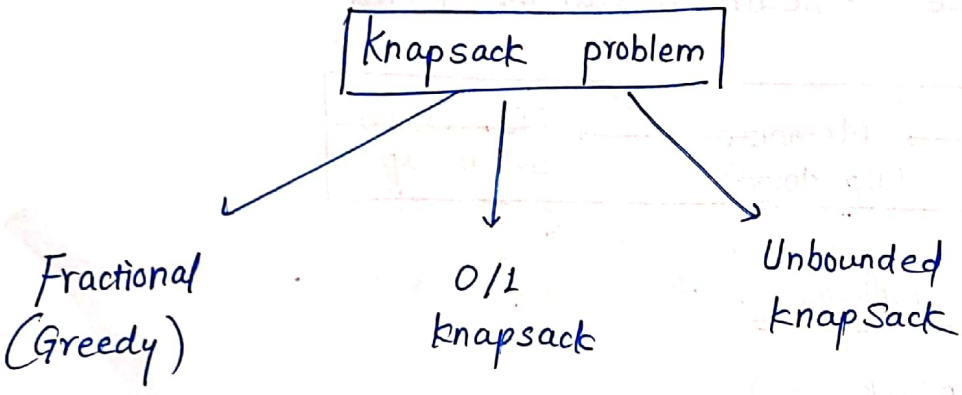
• Variations in DP problems :-

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

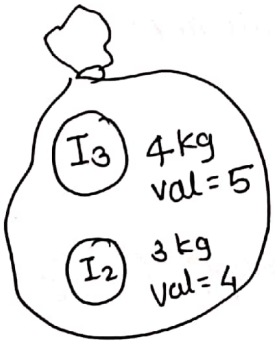
1] Subset Sum
2] Equal Sum partition
3] Count of Subset sum
4] Minimum subset sum difference
5] Target Sum
6] Number of subsets with given difference



problem

	I_1	I_2	I_3	I_4
wt[]:	1	3	4	5
val[]:	1	4	5	7

W = 7
(capacity)



⇒ 4+5
= 9 profit

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

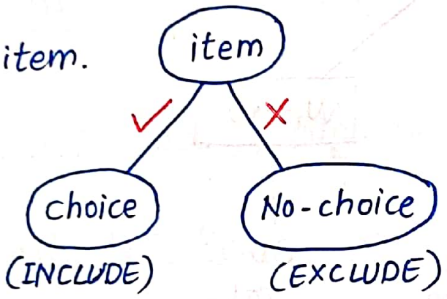
value[] : 1 4 5 7

W : 7 kg

output : maxProfit ?

so they are asking optimal profit

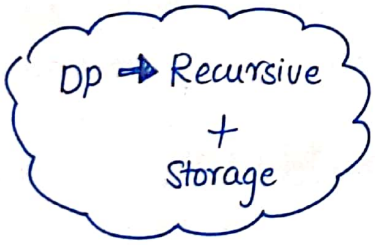
for every item.



choice

Hence this is a problem of DP.

DP : Recursive solution → Memoization (Top-down) → Bottom-up



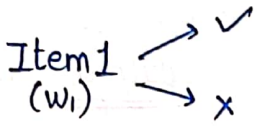
0/1 knapsack Recursion

Input :-

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

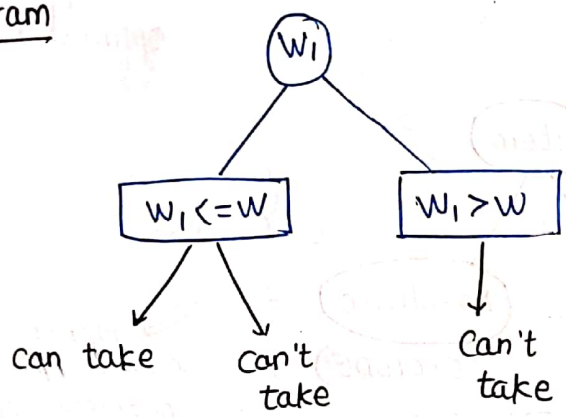
W = 7

Let suppose



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);
}
```

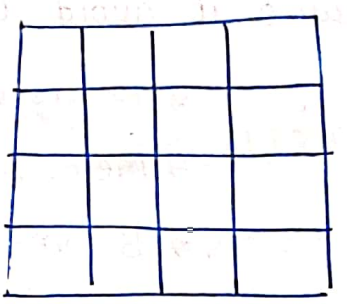
How to think about the base condition ?

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 →

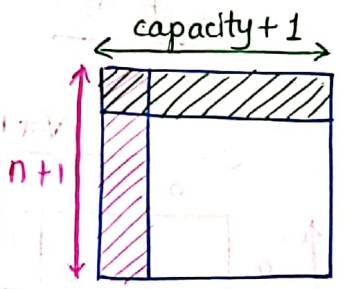
a) n
b) capacity



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));
```

(-1) → Initialization of table with "-1".

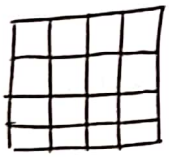


Code

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

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], 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];
}
```

Let's take the constraints
$n \leq 100$
$capacity \leq 1000$

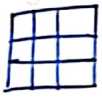
Bottom-up Approch

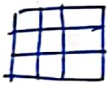
In this only table is there 

Why bottom-up approach is best?

Cause it avoid the error of stackoverflow error.

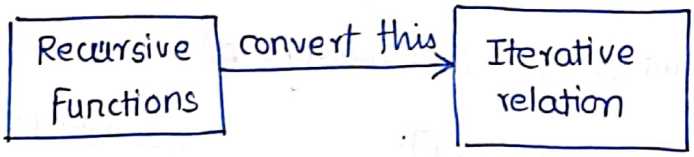
⇒ Recursive → R.C calls

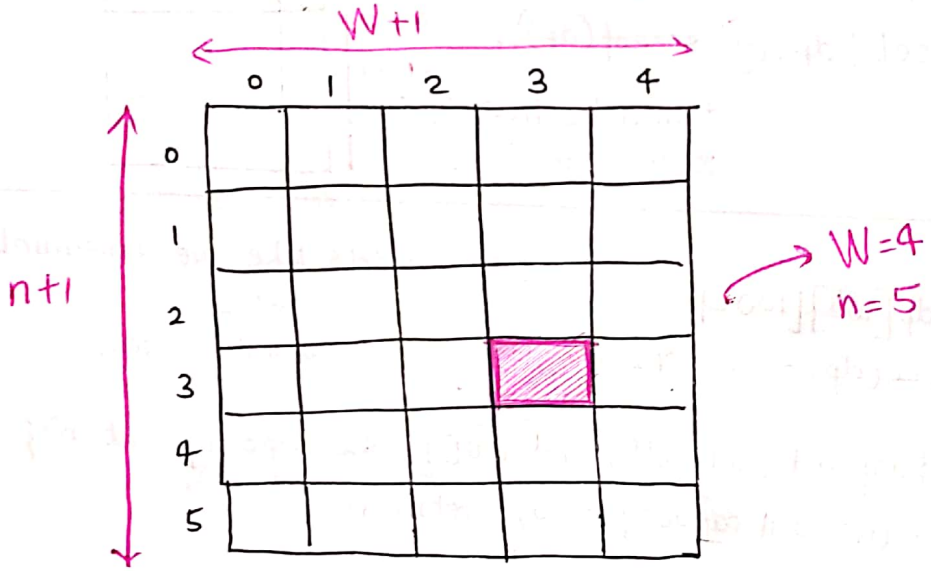
⇒ Memoization → R.C + 

⇒ Bottom-up → 

Step 1 :- Initialization

Step 2 :-





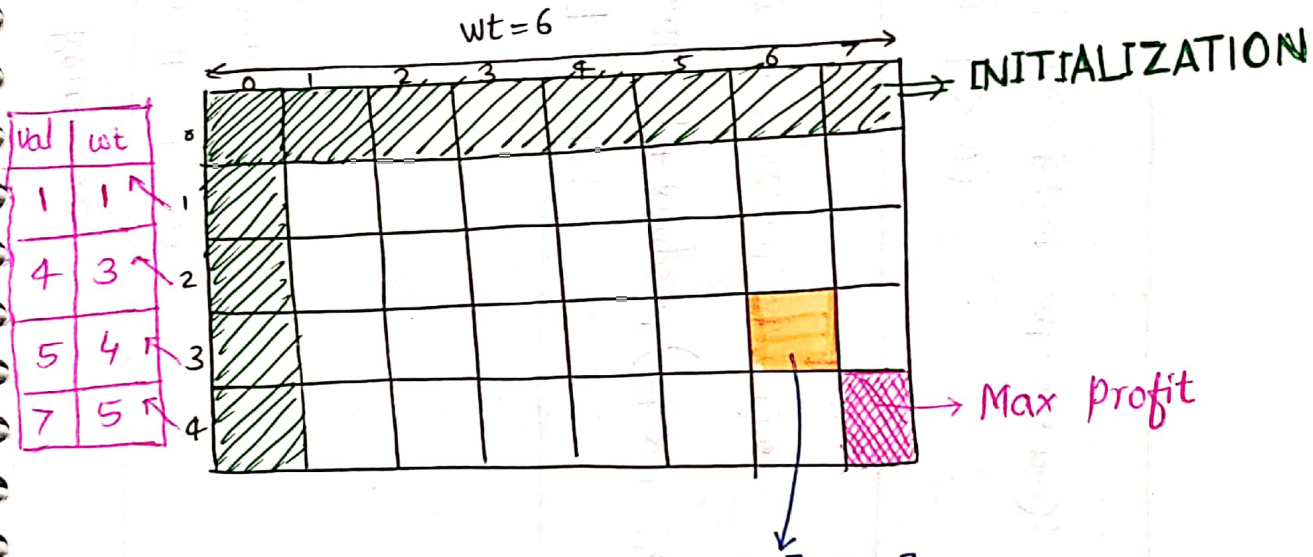
Example :

wt[] = | 1 | 3 | 4 | 5 |

val[] = | 1 | 4 | 5 | 7 |

n = 4

W = 7



wt = [1, 3, 4]

val = [1, 4, 5]

W = 6

• 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	Iterative

`if (n==0 || W==0)`
`  return 0;`
	`for (i=0 ; i<n+1 ; i++) {`
	`  for (j=0 ; j<W+1 ; j++) {`
	`  if (i==0 || j==0) {`
	`  dp[i][j] = 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) );`
	`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)` `  return knapsack (wt, val, W, n-1);`	`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:-

arr[] :- { 2, 3, 7, 8, 10 }

Sum = 11

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

Here ⇒ subset = { 3, 8 } → 3+8 = 11 → return true.
Sum = 11

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

arr[] = | 2 | 3 | 7 | 8 | 10 | sum = 11

If we compare this problem with the knapsack, then

dp[n+1] [W+1]

⇓

dp[n+1][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											

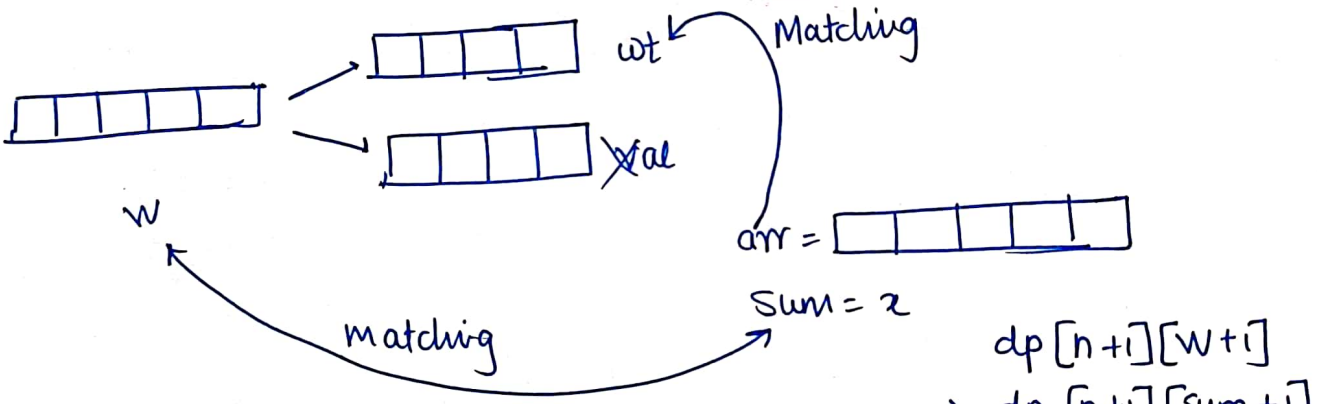
↓ n

If size of array >=0
still we can create with subset sum = 0
[Empty subset]

But if the array of size == 0 and sum > 0
→ Then we can't create any subset
→ 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;
        }
    }
}
```



dp [n+i][W+i]
⇒ dp [n+i] [Sum+i]

MATCHING

wt[] → arr[]

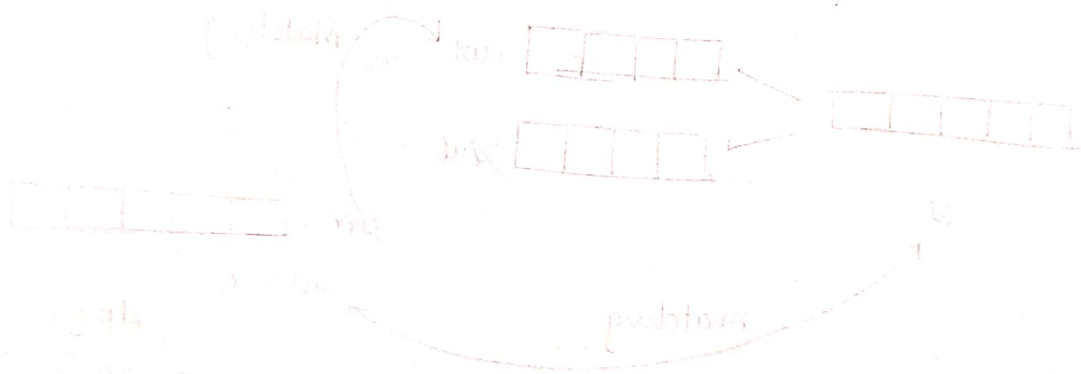
W → Sum

Knapsack

```
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]
```

Subset-Sum

```
if(arr[i-1] <= j){
dp[i][j] = dp[i][j - arr[i-1]]
           ||
           dp[i-1][j];
else
  dp[i][j] = dp[i-1][j];
```



• Equal Sum Partition Problem :-

Given an array, we have to find (divide) two diff. subset such that sum of their elements should be Equal

Flow
1. problem statement
2. Similar with subset sum
3. Odd/Even significance
4. Code Variation

arr = [1, 5, 11, 5]

o/p ⇒ True

{1, 5, 5} {11} → 11

1+5+5 = 11 ✓

→ The basic condition is → if the sum of the array elements is even ⟶ 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) ⟶ return false;
```

arr = [1, 5, 11, 5] ⟶ sum = 22

Now for Equal partition ⟶ 22 → 11, 11

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:-

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

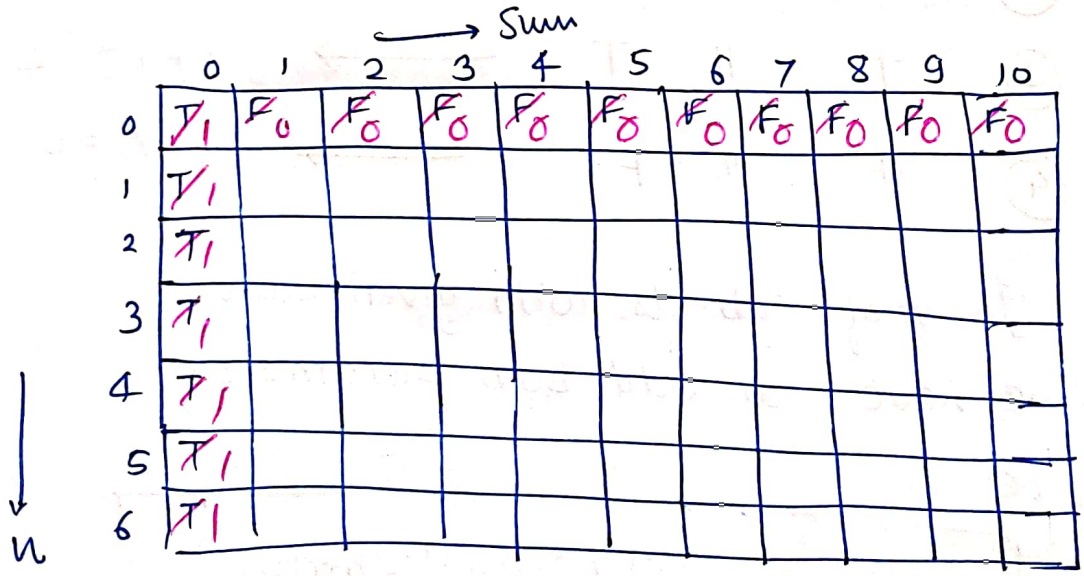
Sum = 10 {2, 8}
{2, 3, 5}
{10}
} Answer :- (3)

Initialization

arr[] = {2, 3, 5, 6, 8, 10}
Sum = 10

t[n+1][sum+1]

t[7][11]



{} → F → <0 Subsets
→ T → >0 subsets

SO → F → 0
T → 1

Subset Sum code

```
if (arr[i-1] <= j)
  t[i][j] = t[i-1][j] || t[i-1][j - arr[i-1]]
                Exclude          Include
else
  t[i][j] = t[i-1][j]
```

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 ~~count~~ "int" and we have to count the # of subsets

count of subsets

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

① T || F → 1+0 = 1

② F || T → 0+1 = 1

③ T || T → 1+1 = 2

④ F || F → 0+0 = 0

So In # of subsets with given sum we have to add both parameters

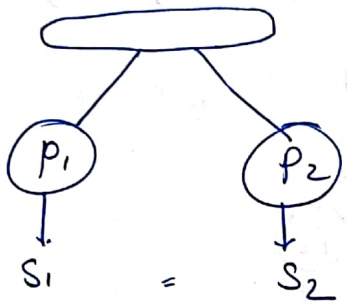
i.e..

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

Minimum Subset Sum difference

arr[] =	1	6	11	5

output = 1



$S_1 + S_2 \Rightarrow$ Even ⟶ Equal Sum partition

i.e. $S_1 - S_2 = 0$

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

$abs(S_1 - S_2) = minimum$

In the above example,

the subsets having the minimum difference

{1, 6, 5} {11}

1+6+5 11

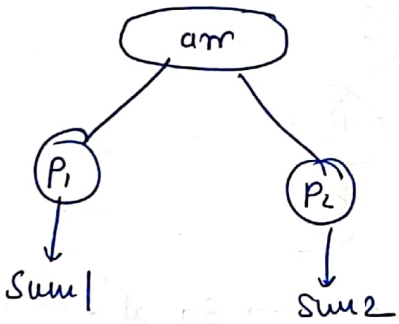
12 11

(12 - 11) ⟶ (1) → Answer

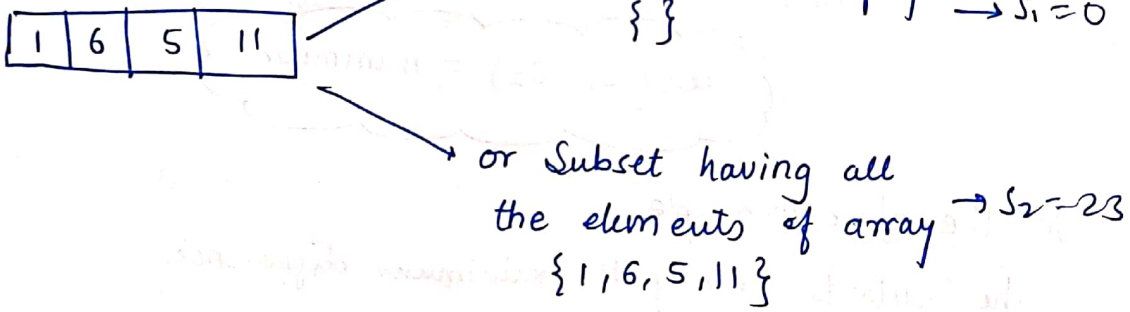
This problem is similar to Equal Sum partition

arr[] | 1 | 6 | 11 | 5 |

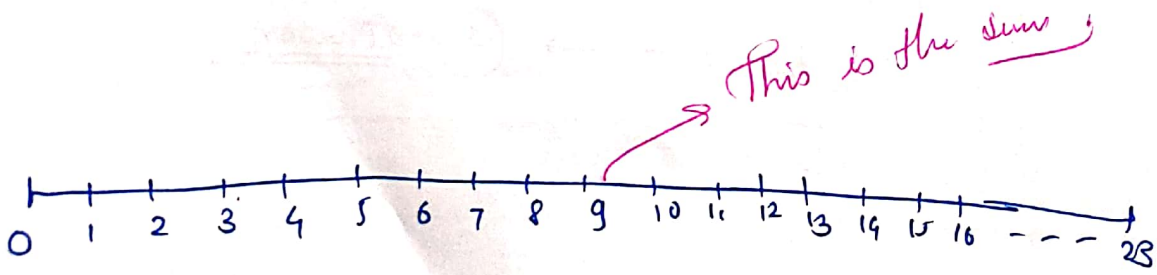
Now we have to find the range of ~~po~~ sum of partitions



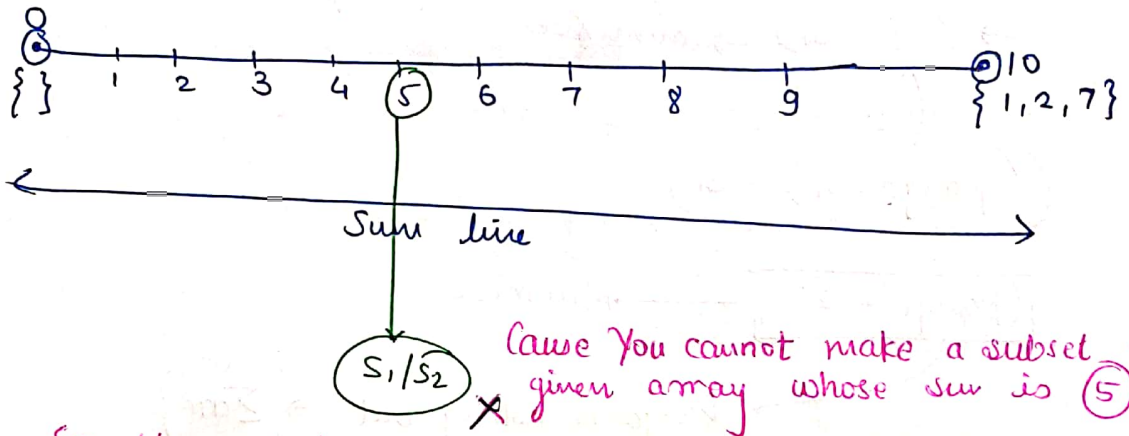
WHAT COULD BE THE RANGE ?



Hence the range is [0, 23]
↓
of Sum

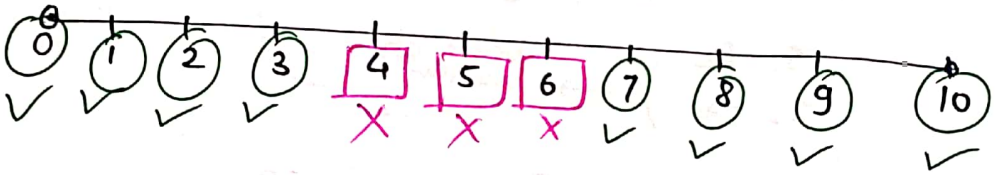


Ex :- arr = | 1 | 2 | 7 |



Cause You cannot make a subset of the given array whose sum is (5)

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



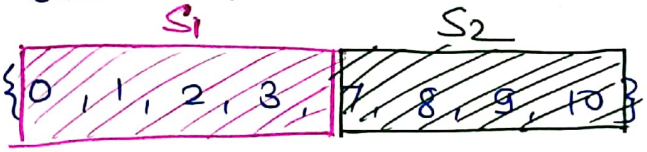
$S_1/S_2 = \{0, 1, 2, 3, 7, 8, 9, 10\}$

Think ?

This array consists both S_1 and S_2. if $S_1 = 1$ then $(\Sigma arr - 1)$ → is also present → i.e. → 9

if $S_1 = 2$ → $(\Sigma arr - 2)$ → is also present → (8)

$S_1 = 3$ → $(\Sigma arr - 3)$ → (7) is also present so we can see,



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

$(S_1 - S_2) \rightarrow$ minimize

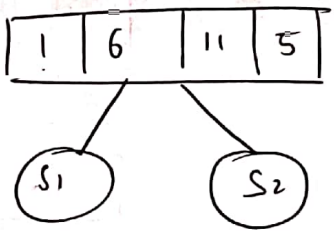
$\rightarrow (S_2 - S_1) \rightarrow$ minimize

$((Range - S_1) - S_1)$

$[Range - 2S_1] \longrightarrow$ Minimize

Range is nothing but $\Rightarrow$ Σarr

Sum Up



$S_1 - S_2$
$S_2 - S_1$) $\rightarrow$ minimize

$[Range - 2S_1] \rightarrow$ minimize

Σarr

$S_1 \rightarrow$ Smaller

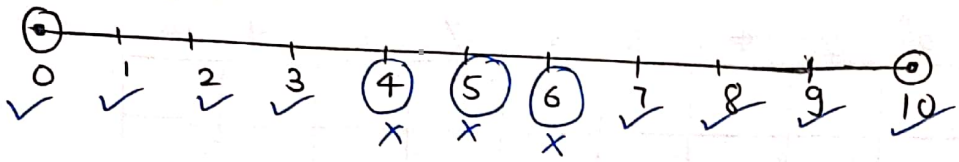
$S_2 \rightarrow$ Greater

Ex:- arr[] = | 1 | 2 | 7 |

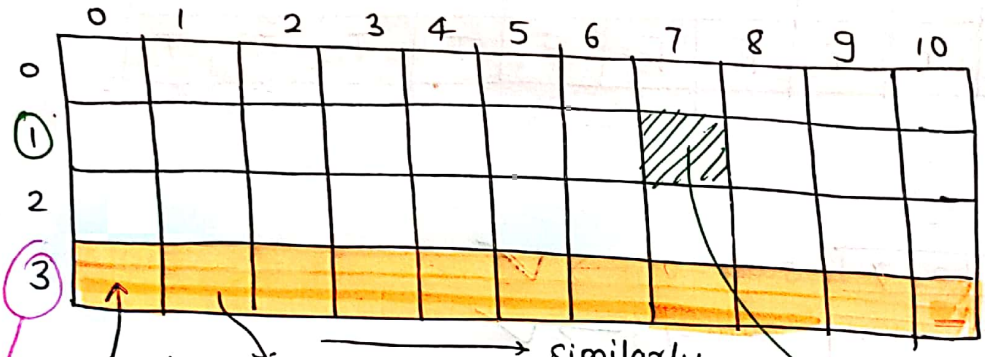
n = 3

Range = 10

Sum line



dp [n+1] [Range+1]



arr[]

[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?

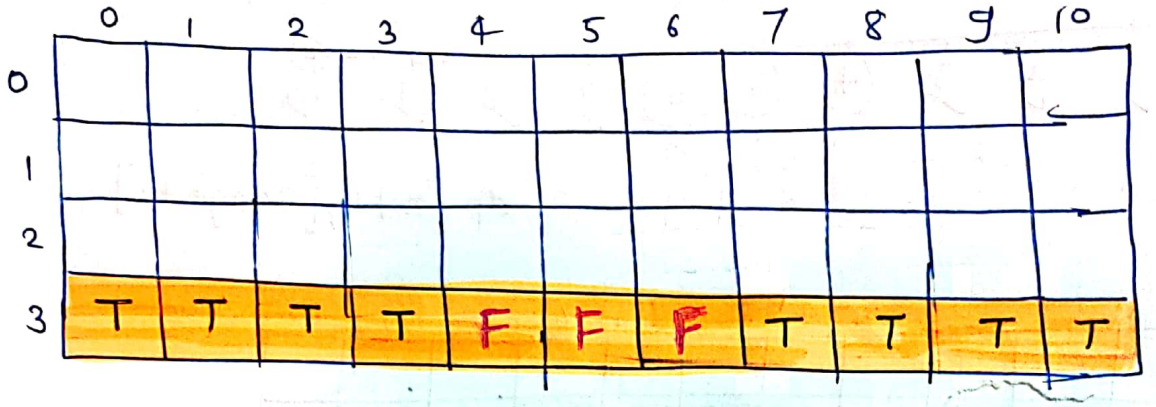
size of array = 1 → {1}

Sum = 7

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

We have to find the last row of the dp table

bool SubsetSum(int arr[], int Range){



}

arr[] = 1 2 7

sum = 0 → ✓

sum = 1 → ✓

sum = 2 → ✓

sum = 3 → ✓

sum = 4 → ✗

sum = 5 → ✗

sum = 6 → ✗

sum = 7 → ✓

sum = 8 → ✓

sum = 9 → ✓

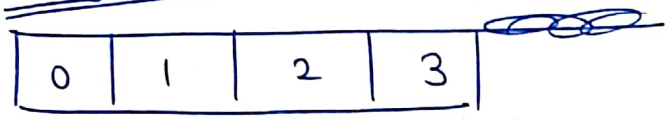
sum = 10 → ✓

We only need the last row of the dp table.

I'll add the last row in a vector till half

↳ cause I want to store smaller values only (Si)

Range = 10 → Σarr

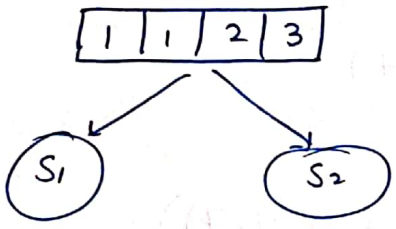



```
int mini = INT_MAX;
for(int i=0 ; i<vec.size() ; i++)
    mini = min(mini, vec[i]);
    mini = min(mini, Range -(2*vec[i]));
}
return mini;
```

Count the no. of Subset with a given difference

arr[ ] = | 1 | 1 | 2 | 3 |

diff = 1



$S_1 - S_2 = diff$

1 1 2	3	$\Rightarrow 4 - 3 = 1$
1 3	1 2	$\Rightarrow 4 - 3 = 1$
1 3	1 2	$\Rightarrow 4 - 3 = 1$

} Here 2 is occured 2 times so we can take any of the one

let's called diff as 'diff'

$$sum(S_1) - sum(S_2) = diff \quad \text{——— Eq}^n (I)$$

$$sum(S_1) + sum(S_2) = sum(arr) \quad \text{——— Eq}^n (II)$$

$$2\,sum(S_1) = diff + sum(arr)$$

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

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

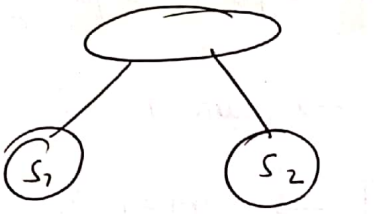
$$= \frac{1+7}{2}$$

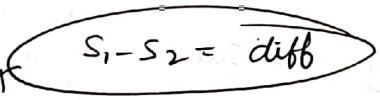
$$= 4 \longrightarrow \text{so} \quad sum(S_1) = 4$$

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

i.e. $S_1 - S_2 = diff$

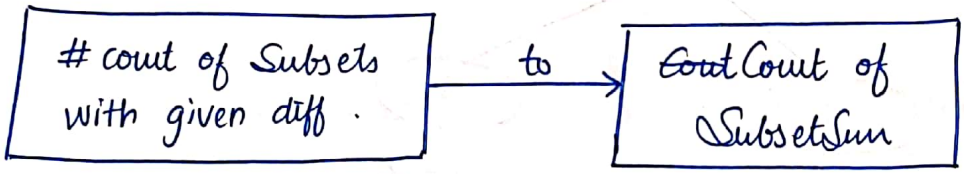
count ?





we have to count

So we have reduced this ~~count~~



Code

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

return countofSubsetSum(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: | 1 | 1 | 2 | 3 |

Sum: 1

o/p = 3

+/-

⇒ Allowed

like we can take any of the sign to any element

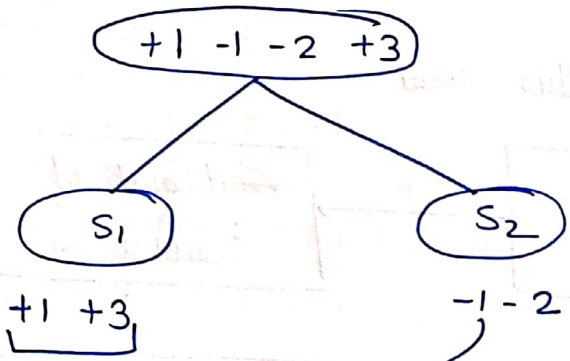
ex:-

+1 -1 -2 +3 ⟶ sum = 1

+1 +1 +2 -3 ⟶ sum = 1

-1 +1 -2 +3 ⟶ sum = 1

} 3



$(sum_1) - (sum_2)$

$= 4 - 3$

$= 1$

So basically ⇒ sum1 - sum2 = 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.

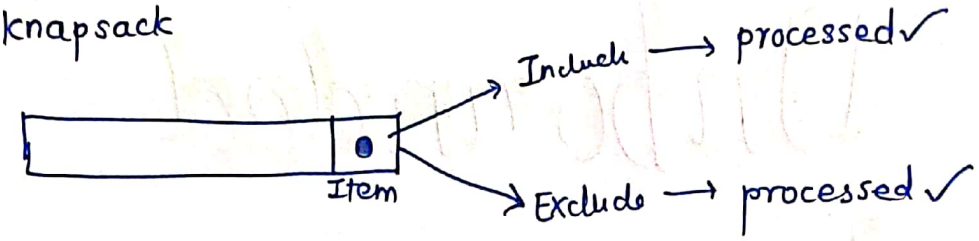
→ Ex:-

let's suppose I like Ice-cream, then I can take Ice-cream multiple times

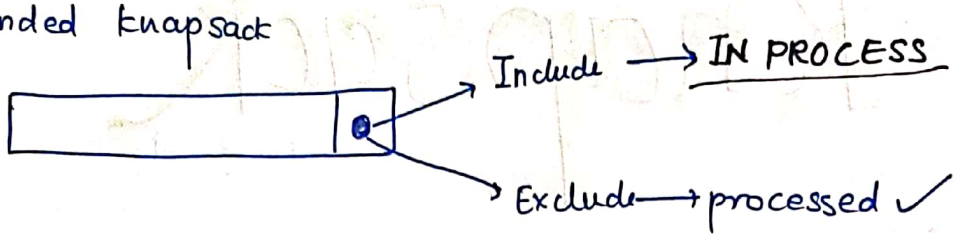
and

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]
```

Rod Cutting problem

You are given a rod of length 'N', you can cut ~~infin~~ 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 price[i].

Ex:-

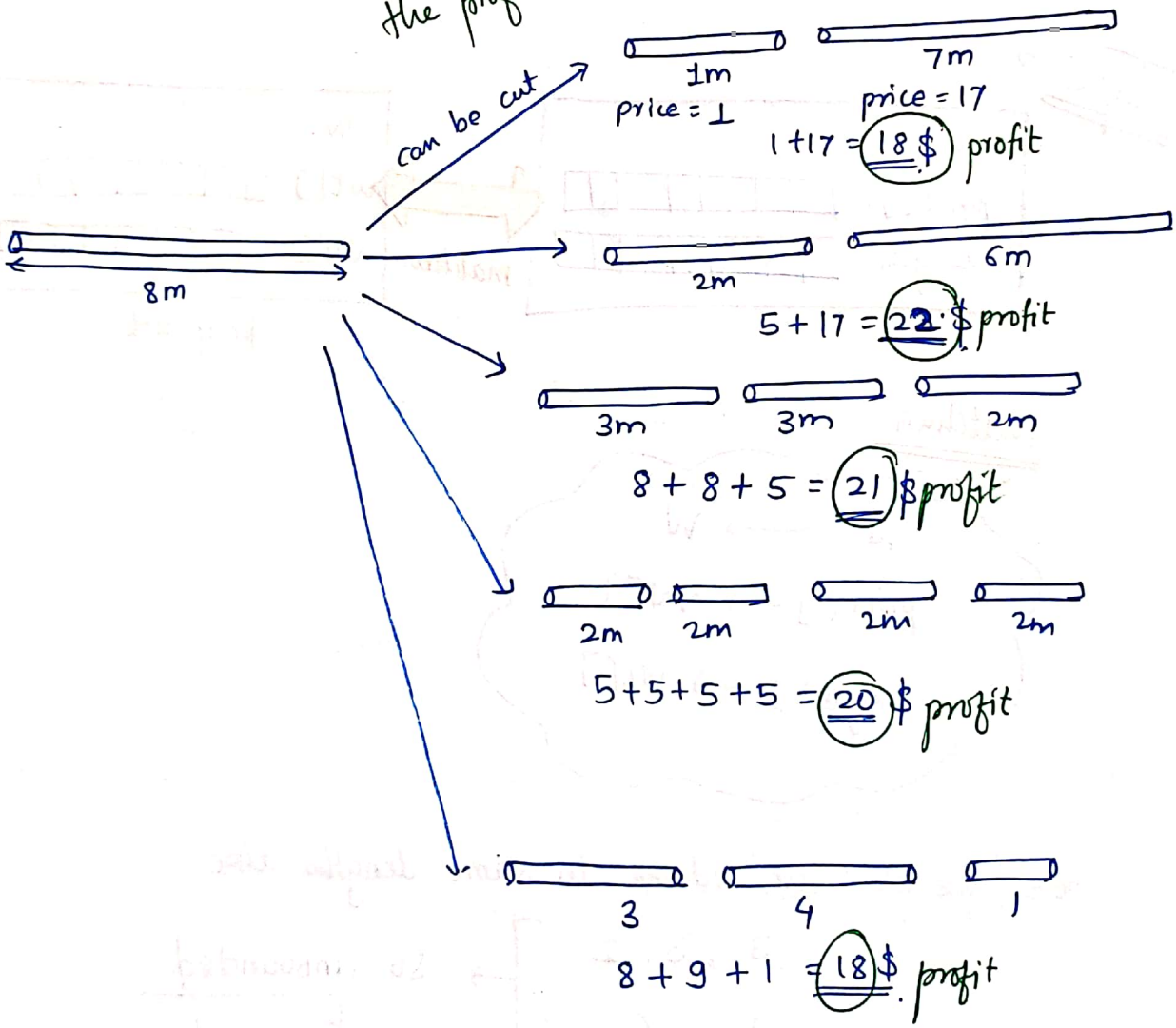
length[] = | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

price[] = | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 |

N = 8

Simulation:-

We have to maximize the profit.



(like this)

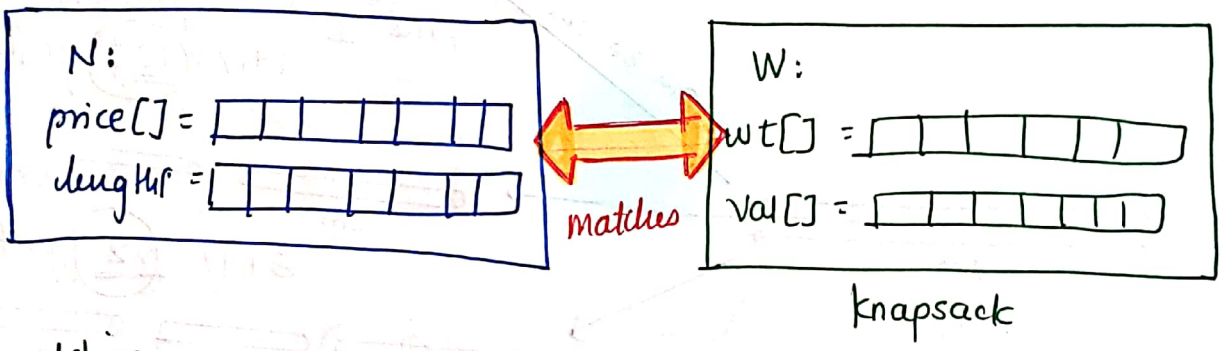
length ⟶ 1 to N

price

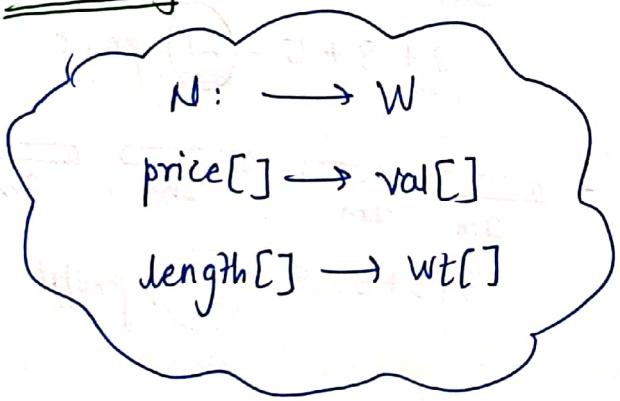
N → length of rod

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

Input ⇒



matching



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

3, 3, 2

1, 1, 6

→ 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.

2+3 = 5
2+2+1 = 5
1+1+3 = 5
1+1+1+2 = 5
1+1+1+1+1 = 5

} 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

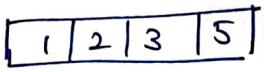
MATCHING

coins[] ⟶ wt[]
Sum ⟶ W

→ This is unbounded knapsack cause repetation is allowed.

Subset Sum

→ This question is related to SubsetSum problem



Sum = 8

{1, 2, 5} → True ✓ (or) → False

Subset sum

```
if (arr[i-1] <= j)
    t[i][j] = t[i-1][j] || t[i-1][j - arr[i]];
else
    t[i][j] = t[i-1][j];
```

Count of Subset Sum

```
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];
```

Code

Max no. of ways (coin change)

```
if( coins[i-1] <= j) {
    t[i][j] = t[i-1][j] + t[i-1][j - coins[i-1]];
}else{
    t[i][j] = t[i-1][j];
}
```

Coin Change - II

find the minimum number of coins

coins[] = | 1 | 2 | 3 |

Sum = 5

#coins = 2 ⟵ 2+3 = 5

#coins = 5 ⟵ 1+1+1+1+1 = 5

#coins = 4 ⟵ 1+1+1+2 = 5

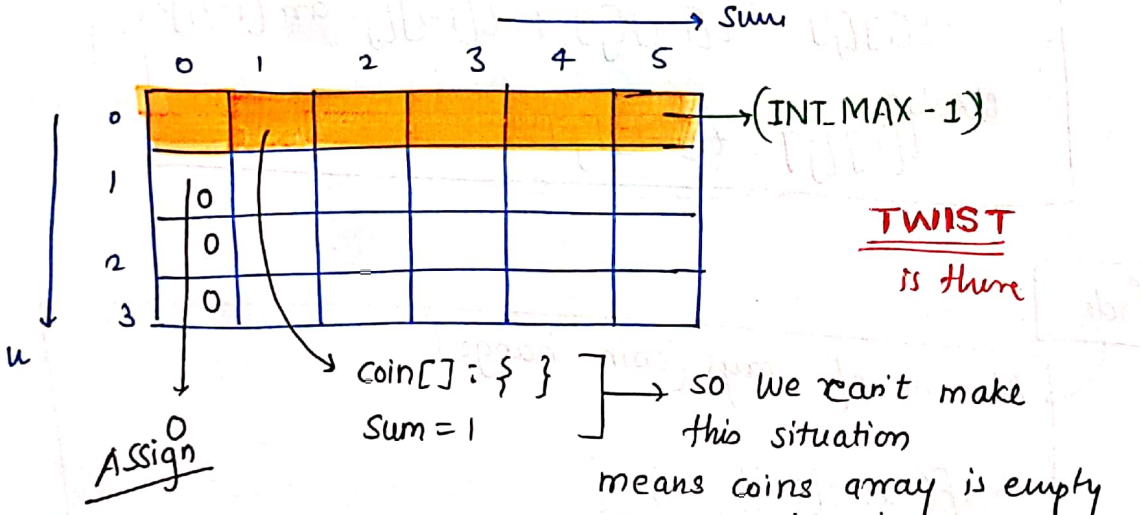
#coins = 3 ⟵ 1+1+3 = 5

#coins = 3 ⟵ 2+2+1 = 5

⟹ Hence minimum 2 coins are required.

dp[coins.size() + 1][sum + 1]

dp[4][6]



TWIST is there

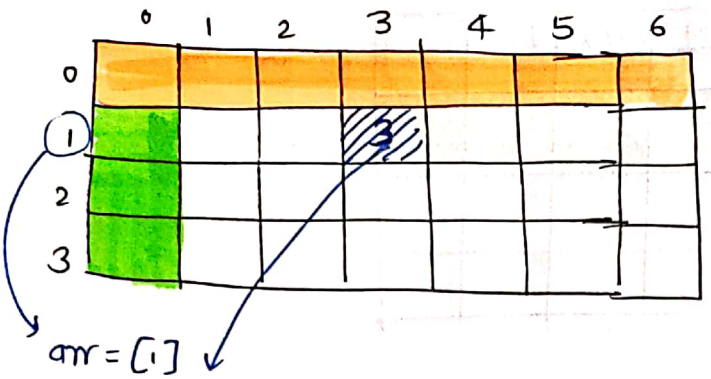
so we can't make this situation means coins array is empty so we require infinite number of coins which sums up to a sum value

but we can't take infinity in that

So store | INT_MAX - 1 |

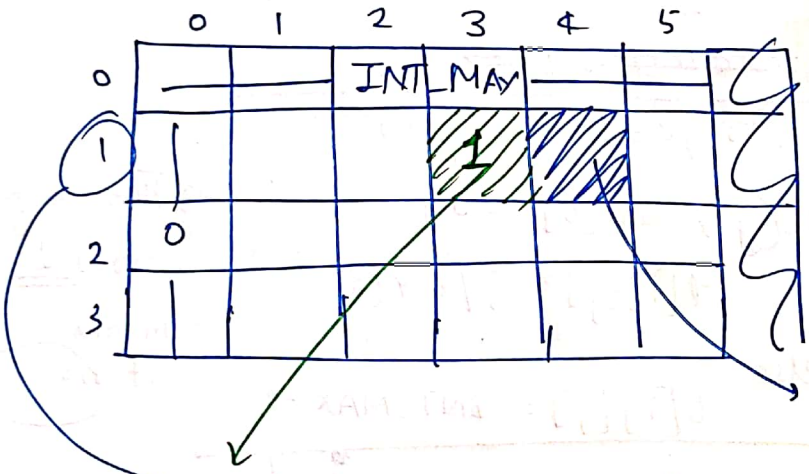
↳ Why -1 ?

Let's talk after this



arr = [1]

Sum = 3 → no. of min coins



if arr = [3 | 5 | 2]

Sum = 4

size = 1

arr = [3]

Sum = 4

min. no. of coins of denomination "3" such that we get sum = 4 (NOT POSSIBLE)

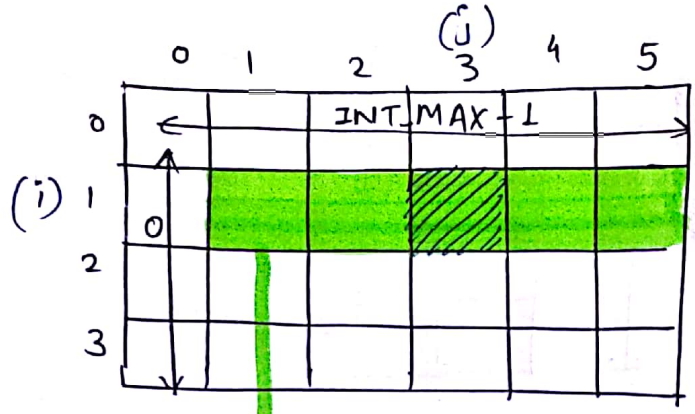
So (INT_MAX - 1)

if arr = [3 | 5 | 2]

but length = 1

[3]

Sum = 3 → so 1 coin required



if (j % arr[i] == 0) → put 1

if (j % arr[i] != 0) put INT_MAX - 1

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

```
for(int j=1 ; j<sum+1 ; j++)
    if(j % 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 ; j++){
    for(int j=1 ; j<sum+1 ; j++){
        if(coins[i-1] <= j){
            t[i][j] = min( t[i][j - coins[i-1]] , t[i-1][j] )
        }else{
            t[i][j] = t[i-1][j];
        }
    }
}
```

If coin is taken then we have to add 1 here

↳ New code

```
for(int i=2 ; i<n+1 ; j++){
    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

SAFE!

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 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 $a \longrightarrow b$
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 (a) appear as subsequence in (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

Recursive :-

Two strings will be given.

string x = "abcdgh" size(n) = 6
string y = "abedfhr" size(m) = 7

a b c d g h ⟶ abdh
a b e d f h r ⟶ abdh

Subsequence = abdh
Length = 4

Recursive Solution

1. Base Condition
2. Choice Diagram

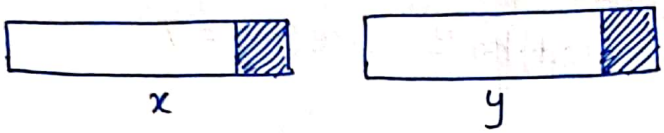
Base Condition :-

if the length of both string is zero

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

Choice Diagram

if(x[n-1] == y[m-1]) ⟶ if the last character matches



⟶ matches

else

⟶ if not matches

Choice Diagram:

if

MATCHES

Remove the matching part

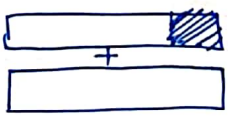
count++;

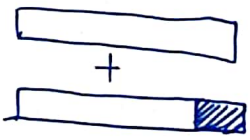
Means now call the recursive function for length n-1, m-1

NOT MATCHES

Longest is asked so take max

(max)





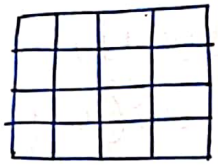
Code

```
int LCS (string x, string y, int n, int m){
    if(n==0 || m == 0) return 0;
    if(x[n-1] == y[n-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?

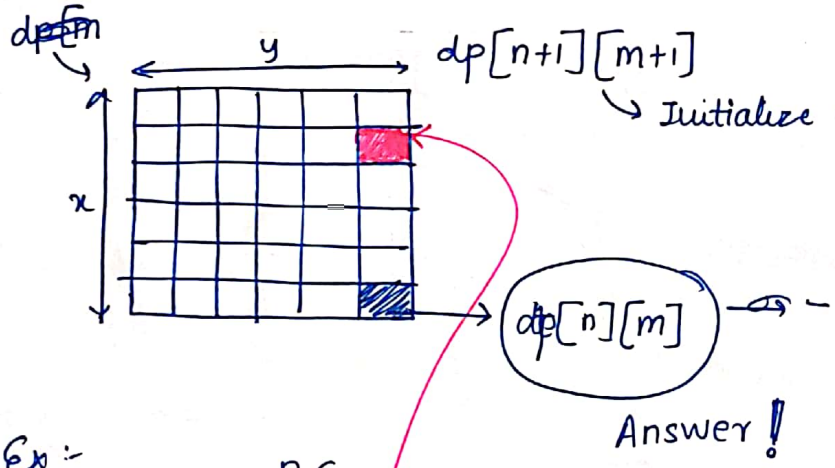
To store the result of subproblems Hence memoization is used.

Recursive call + 

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

→ In LCS, lengths of strings are changing

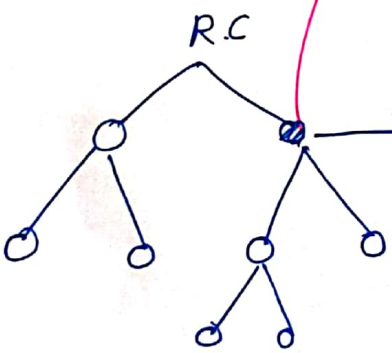
m , n → ARE CHANGING



→ Initialize with '-1'

Answer !

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

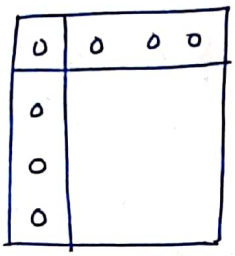
```
// globally declaration of the table
    static int dp[1001][1001];

int LCS(string x, string y, int n, int m){
    if (dp[m][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 In recursion ⟶ Initialization in bottom-up

if(m==0 || n==0) ⟶ zero Initialize

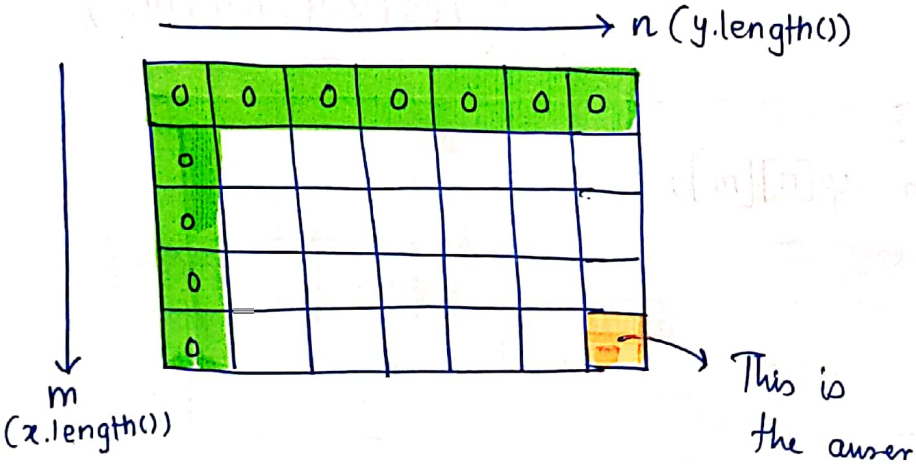
~~dp[i]~~



x : a b c f ⟶ m=4

y : a b c d af ⟶ n=6

dp[5][7] ⟶ dp[m+1][n+1]



// Initialization

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, ~~str~~ subsequence can be found by skipping some characters between the string.

Ex S_1 = "a b c d e" → (skipped d) → abce

S_2 = "a b f c e" → (skipped f) → abce → abce

But in substring, answer should be continuous.

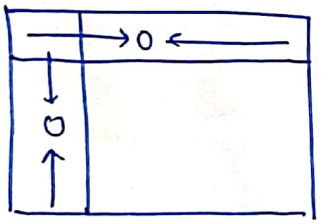
S_1 = "abcde" → e, c

S_2 = "abfce" → ab

These are the substrings

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" ⟶ m = 5
S_2 = "abcdaf" ⟶ n = 6

dp[m+1][n+1]

→ a → ab → abc

	ϕ	a	b	c	d	a	f
	0	1	2	3	4	5	6

n (size of S_2) → abcdaf
↳ like this string is there

	ϕ	a	b	c	d	a	f
ϕ 0	0	0	0	0	0	0	0
a 1	0	1	1 (max)	1	1	1	1
c 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

ac ←

m (size of S_1)

At i=0, j=0 ⟶ $S_1[i] == S_2[j]$ ⇒ a == a
so dp[i][j] = dp[i-1][j-1] +1

At i=0, j=1 ⟶ $S_1[i] \neq S_2[j]$
so dp[i][j] = max(dp[i-1][j], dp[i][j-1])

	ø	a	b	c	d	a	f
ø	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

'a' 'b' 'c' "f"

Not matched

matched

Here f,f are same so we will go diagonally

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

"fcba"

Reverse()

"abcf"

This is our answer

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

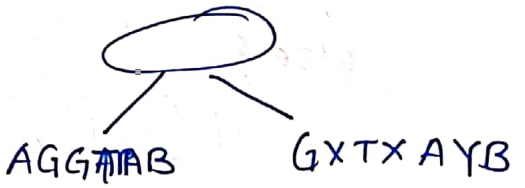
merge them in such a way that

geelce → both a & b strings are present

This is the shortest length

Ex:2 :-

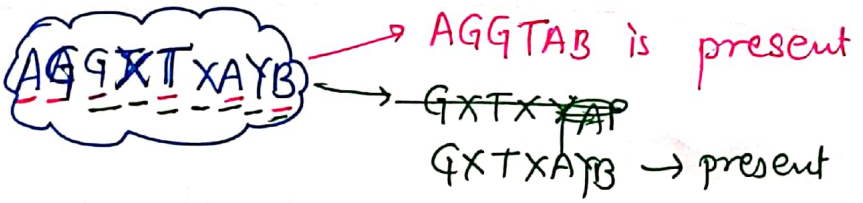
a: "AGGTAB"
b: "GXTXAYB"



AGGTABGXTXAYB → Supersequence

Find the shortest Supersequence (length)

a: "AGGTAB"
b: "GXTXAYB"



This is the shortest supersequence

length = 9

a: AGGTAB

b: GXTXAYB

BG are there → only 1 has taken

G A T B → Write once

G A T B

AGGXTXAYB

⇓

GTAB

Longest Common Subsequence

What would be the worst case of making supersequence :-

a: AGGTAB

b: GXTXAYB

m → AGGTAB + n → GXTXAYB

LCS = "GTAB" LCS = "GTAB"

⇓

AGGTABGXTXAYB worst case

One LCS can be removed IDEA

Length of ~~Shorte~~ Supersequence = m+n

= m+n - 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; j<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"

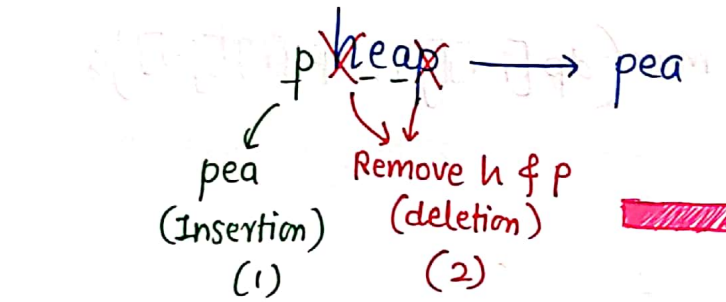
output: Insertion : 1
Deletion : 2

a —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.

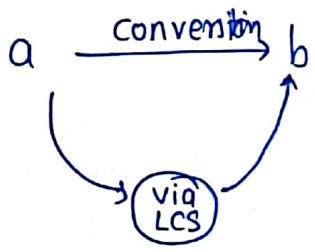
→ This is an variation of LCS problem



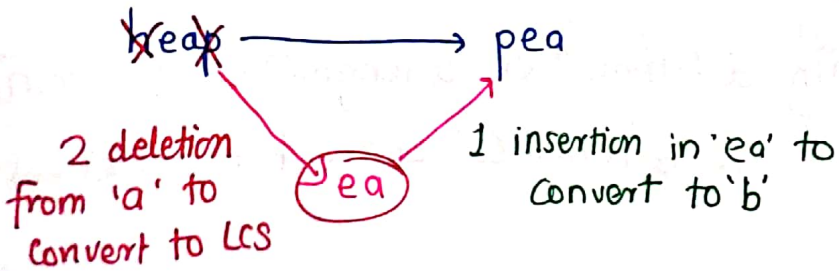
"ea" remain untouched

WHAT IS "ea" ?

Longest Common Subsequence

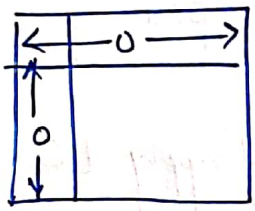


We will do not jump directly on conversion



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 << "Deletion Min. deletions " << a.length() - LCS(a,b,m,n);
    cout << "Min. Insertion " << b.length() - LCS(a,b,m,n); << endl;
}
```

Longest Palindromic Subsequence

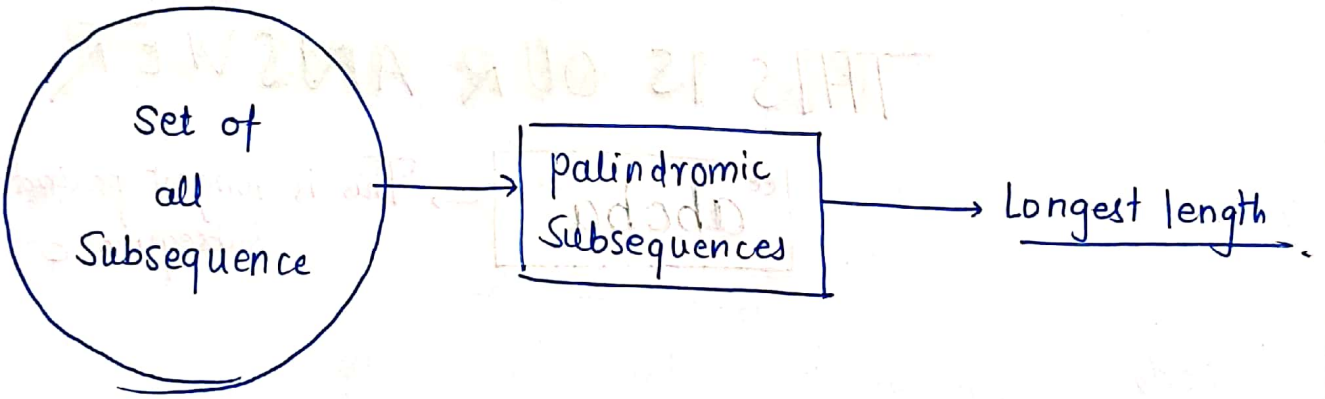
One string is given, you have to return the longest palindromic subsequence.

S: "agbcba" ⟶ In this, "abcba"

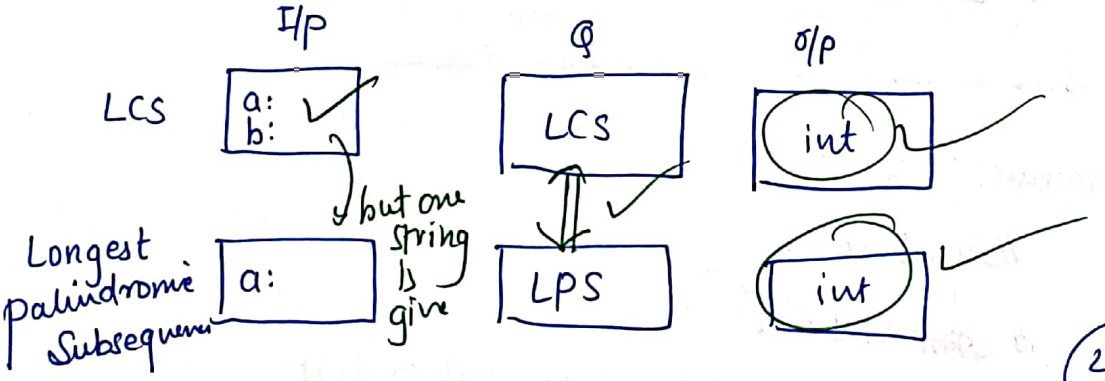
o/p = 5

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 :-



2/3 Grade

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

So LCS can apply

a: → a
b → fun(a)

Longest Palindromic Subsequence

String a = "~~abc~~ agbc 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 WHATT ???

THIS IS OUR ANSWER

"abcba" → This is longest palindromic Subsequence

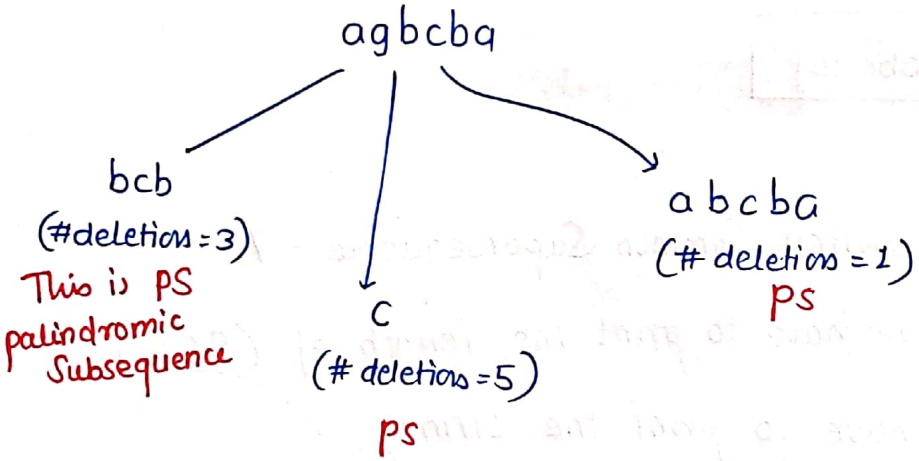
Code

```
LCS (a, b, m, n) {
  // LCS code
}

main() {
  //Input "a";
  // ~~cin~~ str1 = a;
     reverse (str1.begin(), str2.end());
     b = str1;
  cout << LCS (a, b, m, n) << endl;
}
```

Minimum no. of deletions required to make a string palindrome

I/P: S = "agbcba"

O/P = 1



Think ?

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

Hence,

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

Simply

→ Find the length of longest palindromic Subsequence.

→ return (length of String) − (length of longest palindromic Subsequence)

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.

	ϕ	a	b	c	d	a	f
ϕ	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

match ✓

Here a & c are not Equal then

f ✓

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

but add the character of lower cell

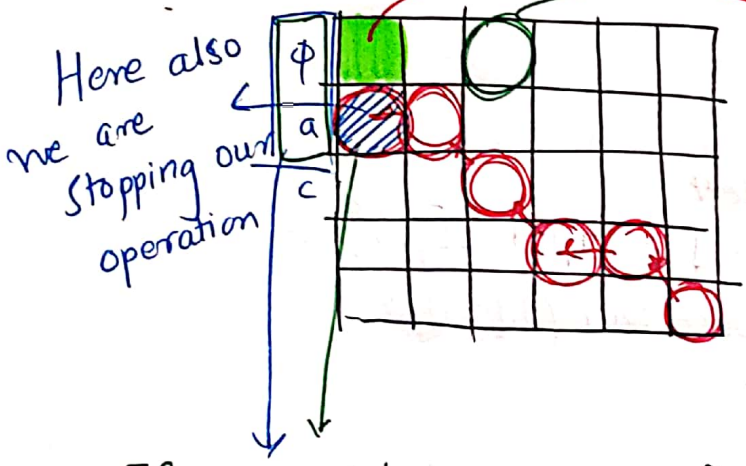
Here, After taking 'f' it will move to ~~cell~~ left

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

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

```
int i = a.length();
int j = b.length();
```

In LCS we are talking about, we can stop our iteration



If I would have stopped here then add 'j' string.

It was not mandatory that it should reach the cell (0,0) in printing of LCS

Let's take an example

a: "ac"

b: " "

Then LCS = " "

but in SCS = "ac"

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

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--;
}
```

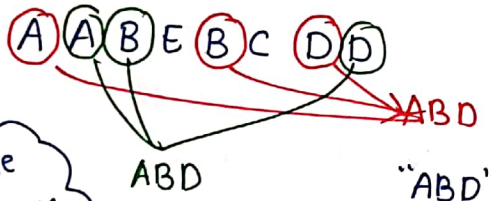
Longest Repeating Subsequence

String S = "AABEBCDD"

↓

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

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

↓

AABEBCDD → This will never come in longest Repeating Subsequence

↓ → Why ?

AABBDD

ABD ABD → Our Answer

Index

E → 3 (in both string)

C → 5 (in both strings)

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

Then why AA is taken?

?

AABEBCDD

AABEBCDD

WH?

index

$S_1 \Rightarrow A \rightarrow$ 0 1

$S_2 \Rightarrow A \rightarrow$ 0 1

> Take the cross

↳ Now we can take, Right ??

YES

SO

Now code it ⇒

LCS

↗ Cause we don't want index same

```
if(a[i-1] == b[j-1] && i!=j) {
    dp[i][j] = 1 + dp[i-1][j-1];
}
else {
    dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
}
```

Sequence Pattern Matching

Input ⇒ a = "AXY"
b = "ADXCPY"

output T/F

Subsequence
order preserved ✓

is "a" is a subsequence of 'b'.
String

A D X C P Y → AXY → True

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

But how length will decide?
There is a possibility of getting the LCS different.

a: AXY (3)
b: ADXCPY (6)

Then LCS (0 → ~~max~~ min (3,6)) → length range

[0 – 3] ⇒ Hence length will be Sufficient

Code

```
~~for(int~~ //Initialization
for(int i=1 ; i<m+1 ; i++) {
    for(int j=1 ; j<n+1 ; j++) {
        if ( a[i-1] == b[i-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) ⟶ return true;
  else ⟶ return false;
```

• Minimum no. of ~~deletions~~ Insertion in a string to make it palindrome

Input :- s: "aebcbda" output = 2

aebcbda

adebcbeda

~~adde~~ xadebcbeda x

2 insertions

4 insertions

⇓

minimum = 2 insertions

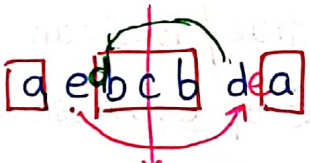
s = "aebcbda"

↓ chang to LPS

Longest palindromic Subsequence

↓

abcba ⟶ e & d are removed



LPS में transform करने के लिए

one 'e' can added

one 'd' can added

So eventually, **# deletions = # Insertions** wow!!

⇒ same code as the min. no. of deletions

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 parenthezization
6. Minimum/maximum value of expression
7. Palindromic Partitioning
8. Scramble String
9. Egg Dropping problem

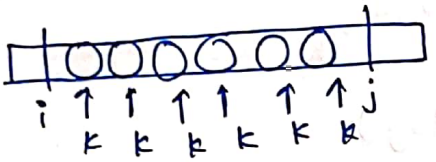
Matrix Chain Multiplication

MCM format

① string or array will be given.

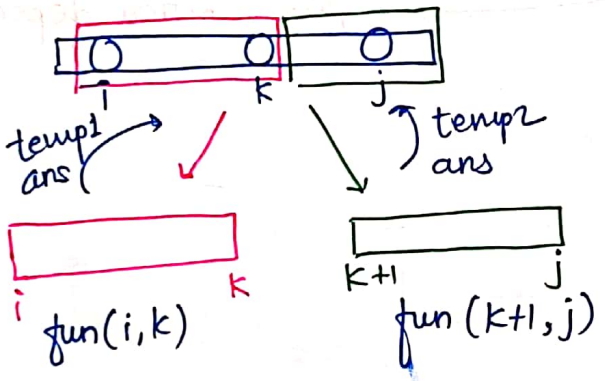
② You will get the feeling of breaking the array/string while solving the problem.

Ex:-

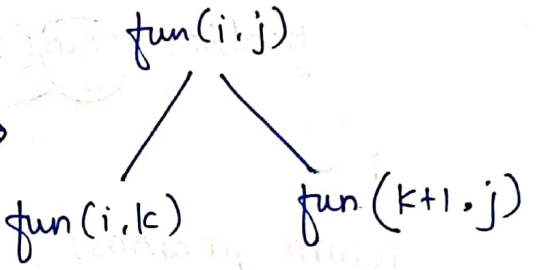


k will move i to j

Ex:-



⇒



③ जो भी temp ans मिलेगा उस पे एक function लगेगा & we 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)
                      (+) ⟶ This also depend on question
                solve(arr, k+1, j);

      finalAns = (fun)(tempAns) :
                          ⟶ This function depends on question
  }

  return finalAns;
}
```

MCM

- **problem Statement**

Array is given

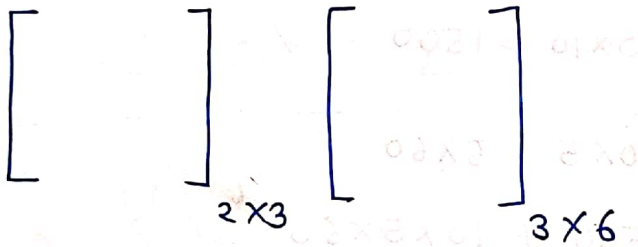
arr[] = {40, 20, 30, 10, 30}

matrices are given

A_1 A_2 A_3 A_4

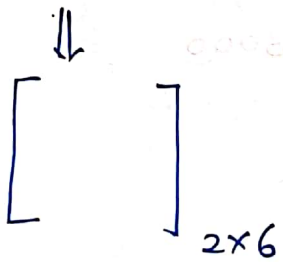
dimension (2×5) (20×20) (30×10) --- (Any dimension can be there)

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



⇒ a×b b×d

⇒ a×d (order)



Min. cost = 2×3 3×6

2 * 3 * 6

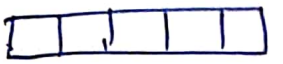
= 6 * 6

= 36

$(A_1(A_2\ A_3))\ A_4$ → cost C_1

$((A_1A_2)(A_3A_4))$ → cost C_2

$A_1(A_2(A_3A_4))$ → cost C_3

arr[] = 

min cost

Ex:-

mat. A = 10*30

mat. B = 30*5

mat. C = 5*60

(A.B) C =

AB = 10 x 30 30x5

= 10* 30*5

AB = 150X10 = 1500

(AB) C = 10X5 5X60

= 1500 + 10X5X60

= 1500 + 3000

(AB)C = 4500

A(BC)

BC = 30X5 5X60

30X5X60 = 300X30 = 9000

A(BC) = 10x30 30x60

= 10x30x60 + 9000

= 18000 + 9000

A(BC) = 27000

150
x 60
9000

minimum cost return करना है।

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

then (n-1) matrices will be given

$A_1 \longrightarrow 40 \times 20$

$A_2 \longrightarrow 20 \times 30$

$A_3 \longrightarrow 30 \times 10$

$A_4 \longrightarrow 10 \times 30$

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

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

Cause here we have to put brackets

$(A_1(A_2\ A_3))A_4$

$(A_1\ A_2)(A_3\ A_4)$

$A_1(A_2(A_3A_4))$

} so we are putting the brackets

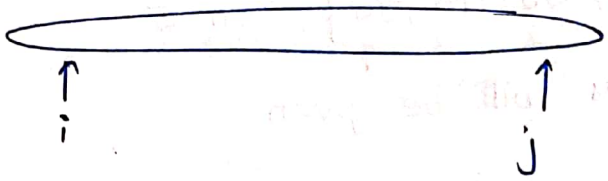
$A_1(A_2A_3\ A_4)$

k (k+1 j)

temp → minCost

tempAns → currCost

$\Longrightarrow$ Select minimin cost



arr[] = | 40 | 20 | 30 | 10 | 30 |

i (crossed out) i j

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

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

$= arr[-1] * arr[0]$ → (-1) index 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'

$A_j \rightarrow arr[j-1] * arr[j]$

$\rightarrow arr[4-1] * arr[4]$

$\rightarrow arr[3] * arr[4]$ ✓ This is correct.

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

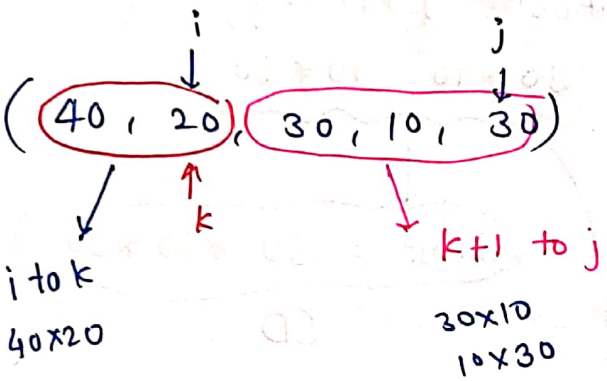
i=j

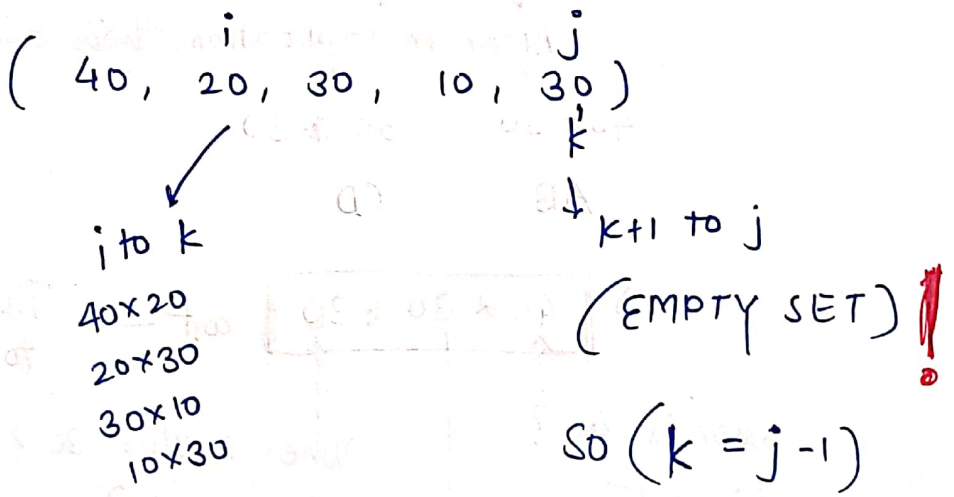
means only 1 element in array

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

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

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

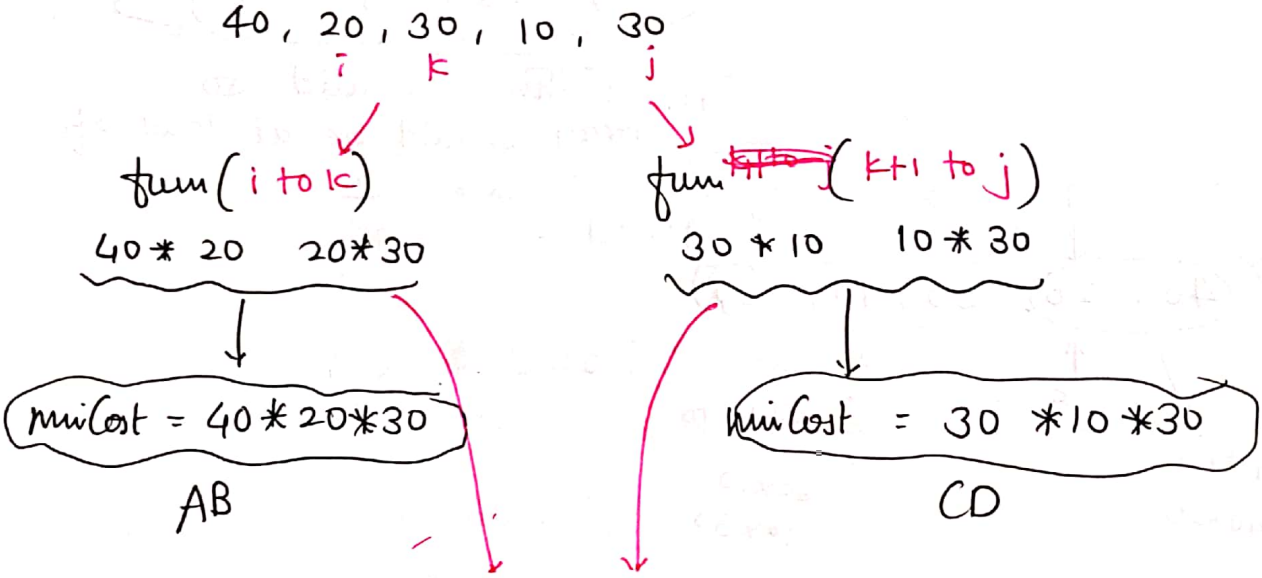




So (k = j-1) तक चलाना पड़ेगा

So k=i to k=j-1

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



After multiplication, their dimension

40 * 30 30 * 30

AB CD

⇒ 40 * 30 * 30 cost → This cost we have to calculate manually

Wher is 40 ?
(i-1) पे

Wher is this 30 ?
(j) पे

This 30?
→ at k

⇒ So formula = arr[i-1] * arr[k] * arr[j]
Extra Cost

Final Code

```
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

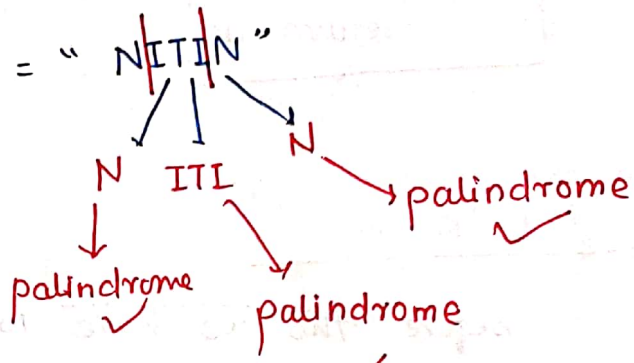
Given a string

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

~~s = "pavap"~~

~~p a v a p~~

s = " NITIN "



we have to find the minimum partitions.

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

n/i/t/i/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   j
n i t i n
```

i=0 → works

i = 0

j = n-1

② find base condition

```
if (i>=j)
    return 0;
```

→ if (i==j)

↓

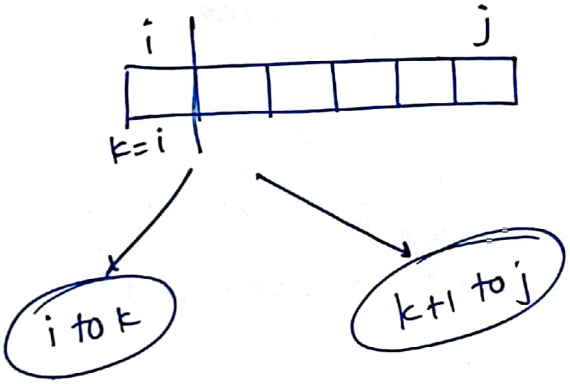
string contains only one character

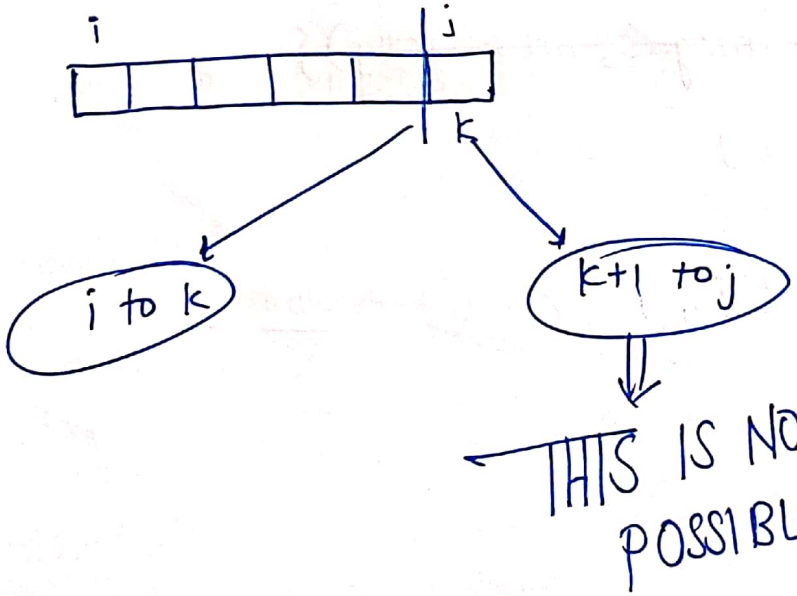
Hence zero partition

③ find 'k' loop

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

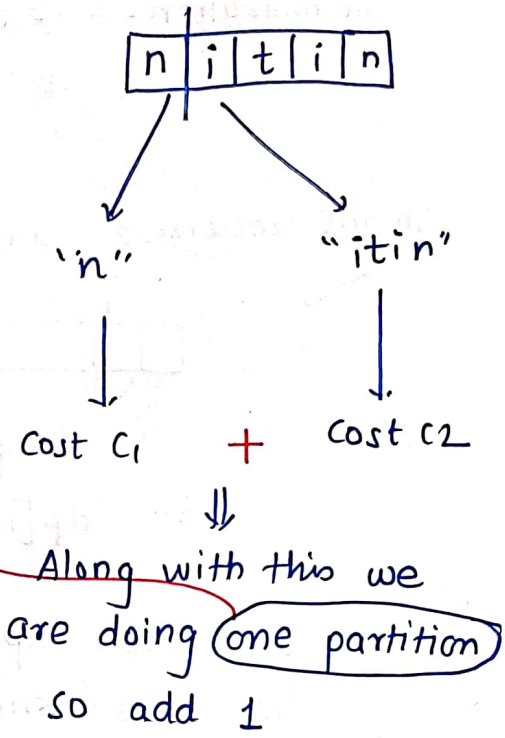
```
if (isPalindrome(s, i, j) == true)
    return 0;
```





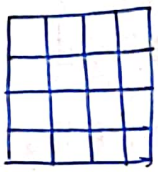
So k should be till j-1 from i

```
for (int k=i ; k<=j-1 ; k++){
int tempAns = solve(s, i, k)+
              solve(s, k+1, j)
              +
              1;
mini = min(mini, tempAns);
}
return mini;
```

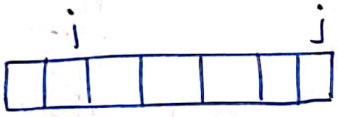


Palindromic Partitioning memoization

s = "nitik"

memoization → R.C. + 

In the recursive solution, i and j are changing



$0 \le i, j \le 1000$

So dp[1001][1001];

```
int static dp[1001][1001];
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

see,

In the for loop, we are calling two recursive calls but what if we check the left part → 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~~ 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 or F and T"

↓

s = "T | F & T"

string → T, F } characters + { &, |, ^ } 3 symbol } string will be made up of these combinations only

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

s = "T | F & T"

(T | F) & T
= T & T
= T

T | (F & T)
= T | F
= T

(T) | (F & T ^ F)
 ↑ k

(T | F) & (T ^ F)
 ↑

} k is moving with k = k + 2

(T) | (F & T ^ F)

k

Expression | operator | Expression

k

means, k will always dt operator

4 steps :-

① find i & j :-

T | F & T ^ F

i j

→ No conflict

i = 0
j = n-1

② Base Condition :-

(T or F and T) xor (F)

i to k-1 k k+1 to j

Expression xor Expression

i to k-1 k k+1 to j

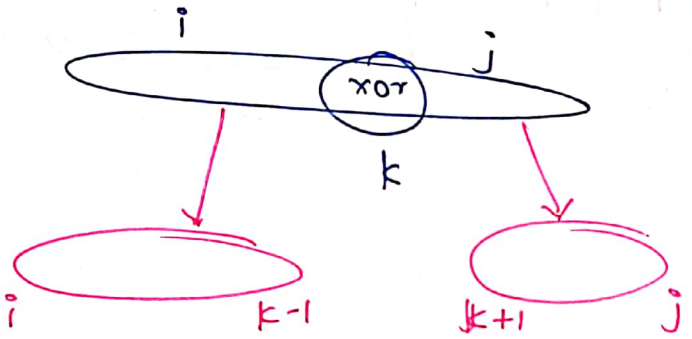
exp1 [xor] exp2

No. of ways of 'True' ⇒

(leftTrue * RightFalse + LeftFalse * RightTrue)

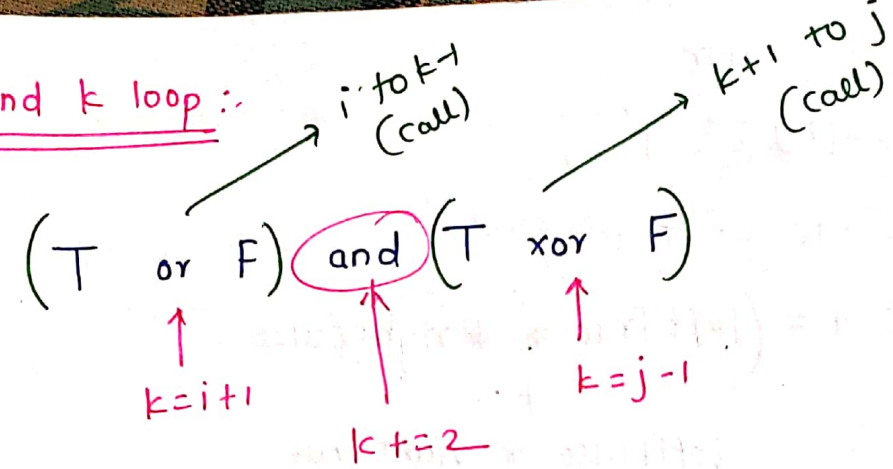
Means we have to find the ways ~~to find~~ for exp1 true as well as false.

exp1 [xor] exp2




```
//Base condition
if(i>j)
    return false
if(i==j) {
    if(isTrue == true)
        return S[i] == 'T';
    else
        return S[i] == 'F';
}
```

3) find k loop :-




```
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);

// for and operator
 if(s[k] == '&'){
   // No. of ways for True
      if(isTrue == true){
        ans += leftTrue * rightTrue;
      }
     else{
          ans +=(leftTrue * rightFalse
                  +
              rightTrue * leftFalse
                  +
              leftFalse * rightFalse);
      }
}
```

→ TempAns.

For True
LT RT

For False
LT RF
OR 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, ~~#~~-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 ~~rig~~leftfalse = Solve(s, i, k-1, false);
  int rightTrue = solve(s, k+1, j, true);
  int rightfalse = solve(s, k+1, j, false);
}
```

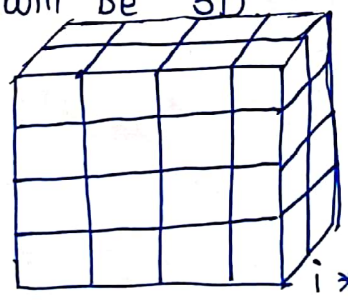
No. of variables = 4

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[1001][1001][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

}

Code :-

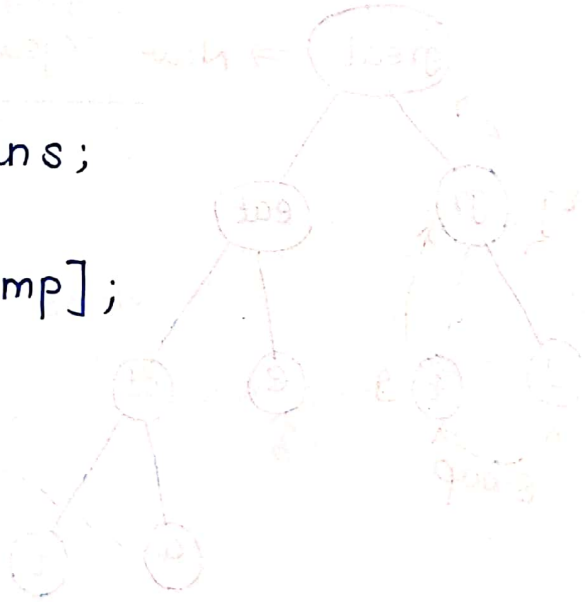
```
int Solve(string s, int i, int j, bool isTrue) {
    if(i>j) return true;
    if(i==j){
      if(isTrue == true)
          return s[i]=='T';
      else
          return s[i]=='F';
    }

    // New code (map)

    string temp = to_string(i);
        temp.push_back(" ");
        temp.append(to_string(j));
        temp.push_back(" ");
        temp.push append(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+" "+isTru

```
// 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];

}
```



Scrambled String

Problem Statement :-

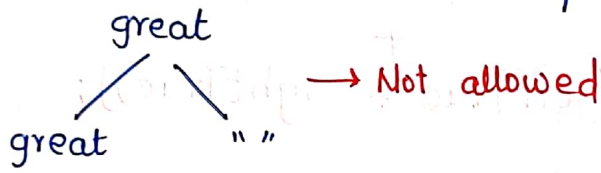
Given two strings 'a' and 'b', check whether they are scrambled string or not.

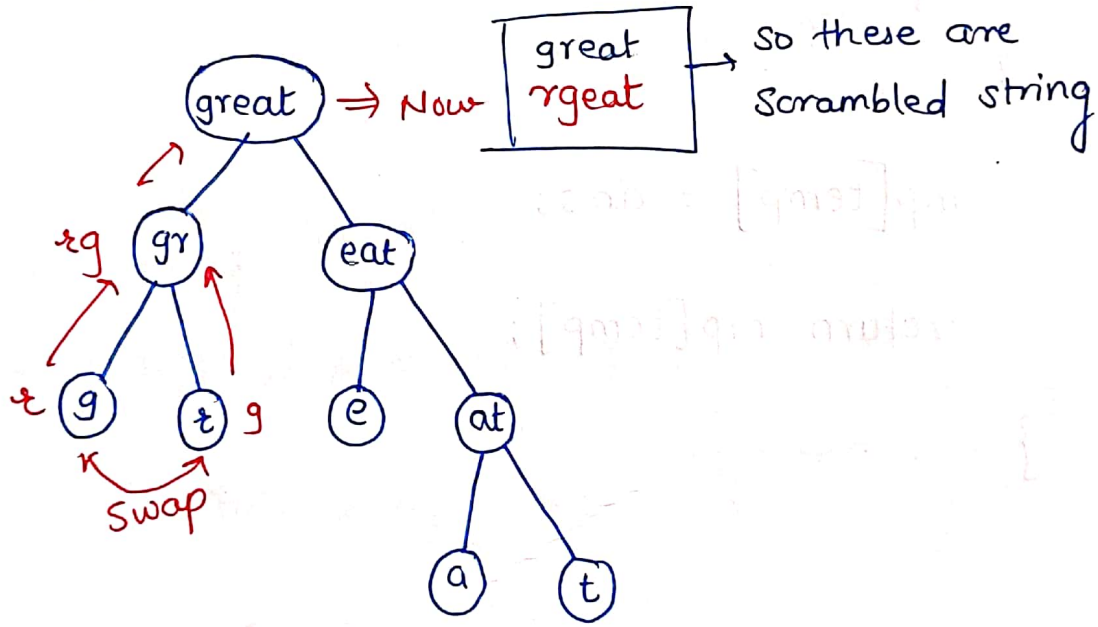
I/p = a: "great" o/p = True
b: "rgeat"

- What is scrambled string ?

① Create a binary tree.

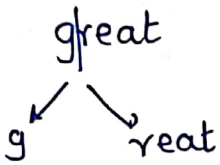
② You cannot make child of binary tree empty

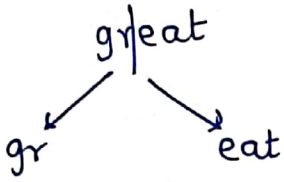


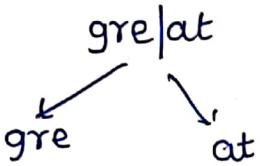


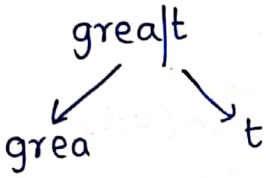
You can break down in any way.

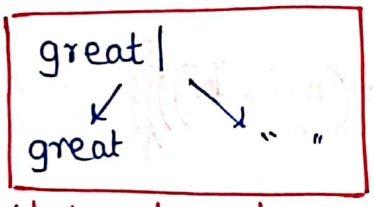
we can do the swapping (zero or more) times of the child nodes of non-leaf nodes.



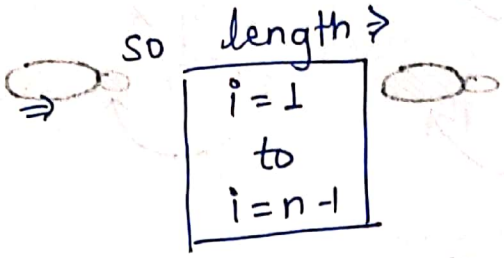






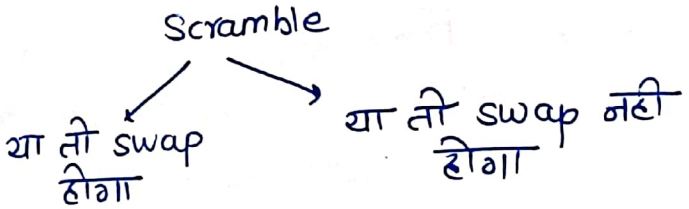


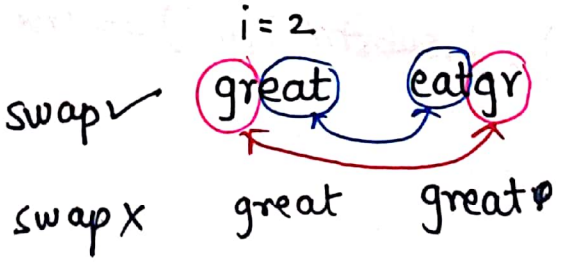
Not allowed

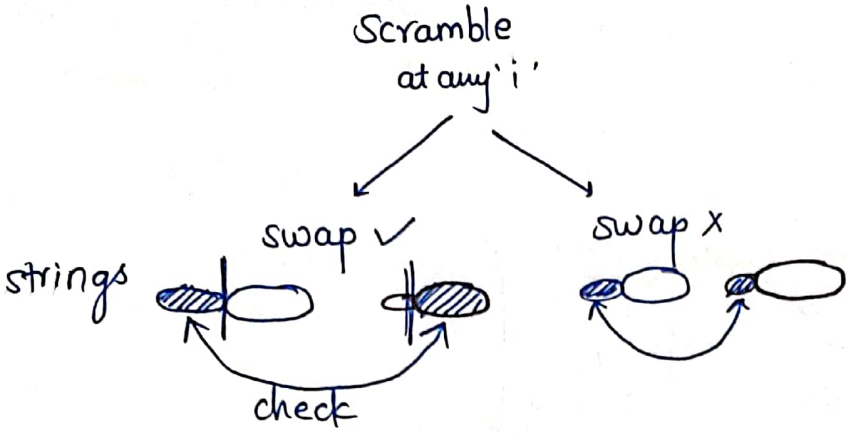


So as we are breaking at each length
this is MCM problem

Approach :-







a ⟶ b
a(last) ⟶ b(first

a(first) ⟶ b(first)
a(last) ⟶ b(last)

Case-1

swap ✓

i=2

0 1 2 3 4
great

0 1 2 3 4
eatgr

bool Solve(a,b)

Condition-I

```
if(Solve(a.substr(0,i), b.substr(n-i, ))
                    ↓ True
&&
solve(a.substr(i,n-i), b.substr(0, n-i)))
                    ↓
                  True
```

Case-II

great great

great grate

Condition-II

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

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

return true

//Base Condition

great rgreat ω

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) == a) 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 बना लो

unordered_map <string, ~~int~~ bool> mp

Changing variables ⟶ a & b

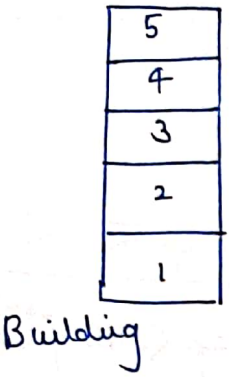
key ⟶ string = "a" + " " + "b"

```
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;
}
```

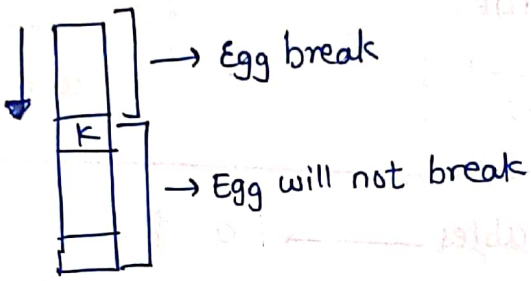
Egg Dropping Problem

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

Example: e = 3, f = 5 → o/p = 3



~~find~~ the critical floor such that egg will break



~~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 → minize the number of attempts.

→ minimum no. of attempts = 3

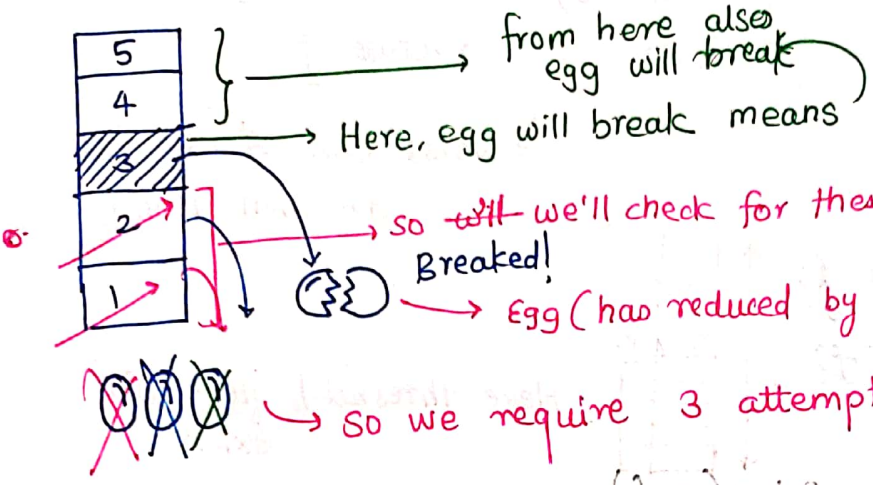
→ Best Technique/ Strategy

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.

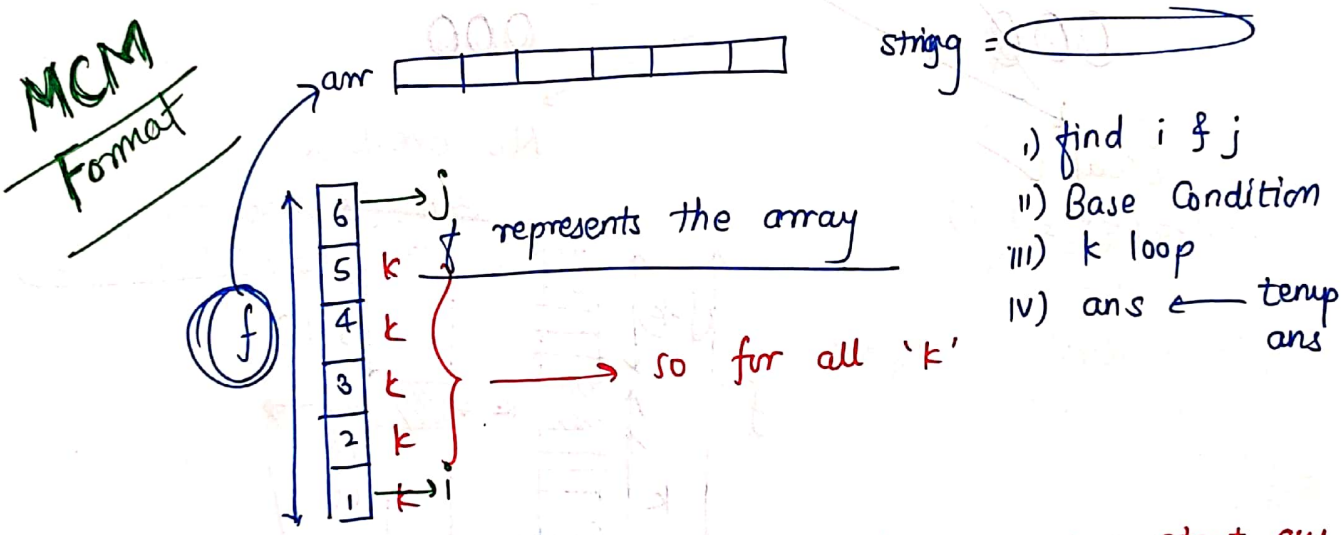
If e=1 and 'f' floors are given then in worst case, 'f' attempts required

e=3 f=5



→ So we require 3 attempts in worst case.

MCM Format



i) find i & j
ii) Base Condition
iii) k loop
iv) ans ← temp ans

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

⇒ 'means' find the k loop

→ हर जगह पे loop लगा के देखेंगे!

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

// Base Condition

think of the smallest valid input

I/P

e → 0/1

f → 0/1

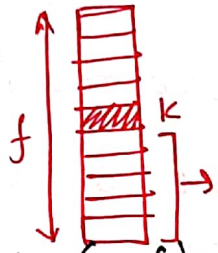
if (e==0) → we never find the ans

if (e==1) → return f

↳ worst case में last floor तक जाना पड़ेगा।

(f==1) return f;

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



→ Here threshold floor will exist

TempAns :-

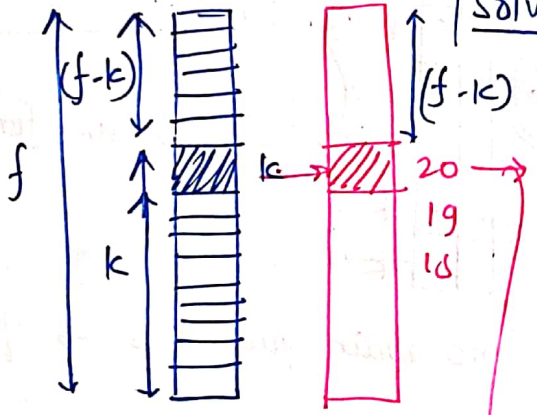
Solve (e, f)

OO~~O~~ → Break ✓ → Solve (e-1, k-1)

OOO → No break ✗

~~Solve (e, k-1)~~

Solve (e, f-k)



Suppose f=20 pe break नहीं कर रहा

so means 19, 18 पे break नहीं करेगा

and this is for sure

and अपने को first floor चाहिए जहाँ से egg break हो जाती।

So ↑ ऊपर जाना पड़ेगा

```
int Solve(int e, int f){
    // Base Condition
    if(f==0 || f==1) return f;
    if(e==1) return f;

    // 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) );
        mn = min(mn, temp);
    }

    return mn;
}
```

TASK

minimum no. of attempts in worst case

in each iteration, one attempt we are taking

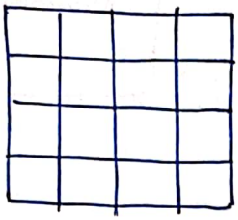
Why "max()" ? Cause we have to find in worst case

min → minimum no. of attempts.

Don't get Confused :)

Egg Dropping Problem Memoization

matrix dimension : number of changing variable.

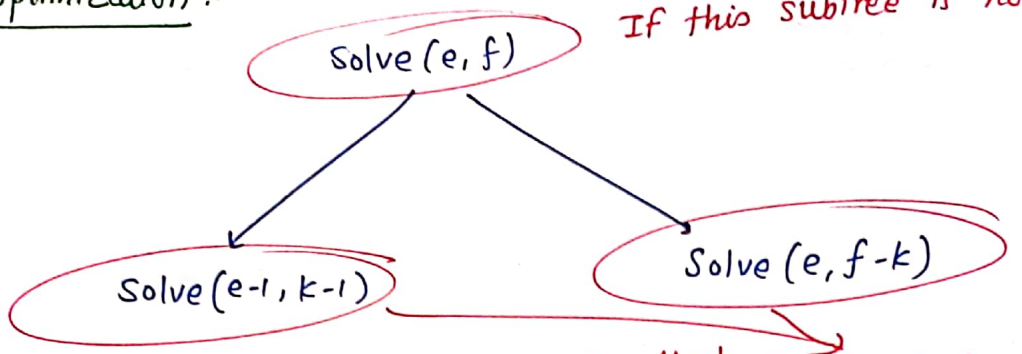


e x f

as e and f are changing

```
//global declaration of table
    int static 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;
    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;
```