Notes Made By

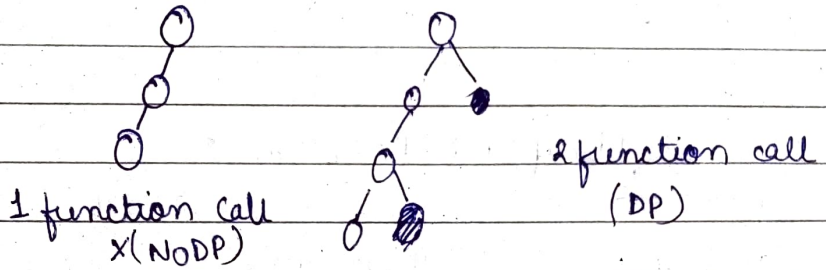
- Riti Kumari

Dynamic Playlist (Aditya Verma)

DP = enhanced recursion

How to identify DP problem → (2 calls)

1) Where there is recursion, DP is used (for overlapping problem)
   a) Choice



   b) Optimal - Min, max, largest

How to write DP code?

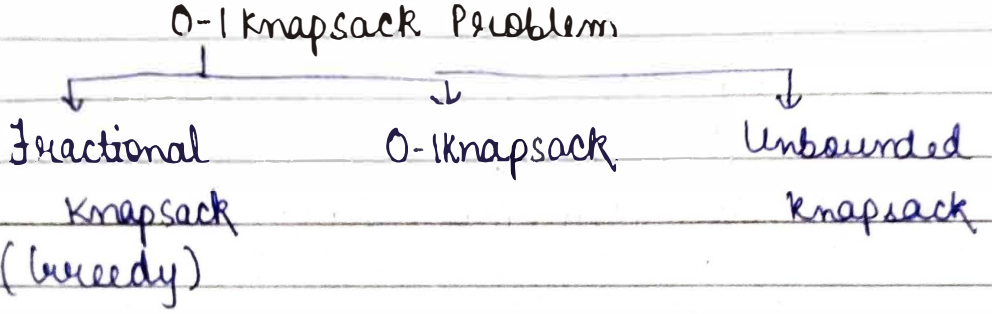
Recursive solution → memoization → Top down approach

Questions on DP.

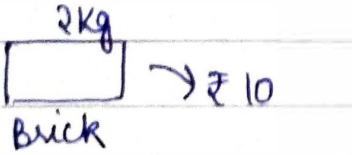
1) 0-1 Knapsack (6)
2) Unbounded knapsack (5)
3) Fibonacci (7)
4) LCS (15) (longest common subsequence)
5) LIS (10) (longest increasing subsequence)
6) Kadane's Algorithm (6)
7) Matrix chain Multiplication (7)
8) DP on trees (4)
9) DP on grid (14)
10) Others (5)

Types of Knapsack

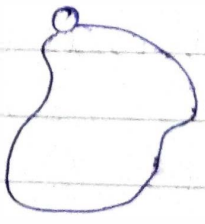
1. Subset sum
2. Equal sum partition
3. Count of subset sum
4. Minimum subset diff
5. Target sum
6.



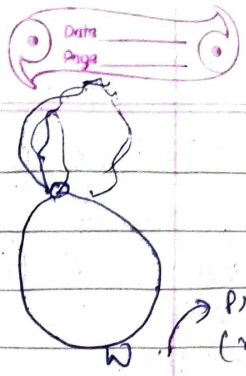
0-1 Knapsack → We are given some weight & some value. Then a max weight W. pick items so that the profit is maximum. And the weight has a given bound W.



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



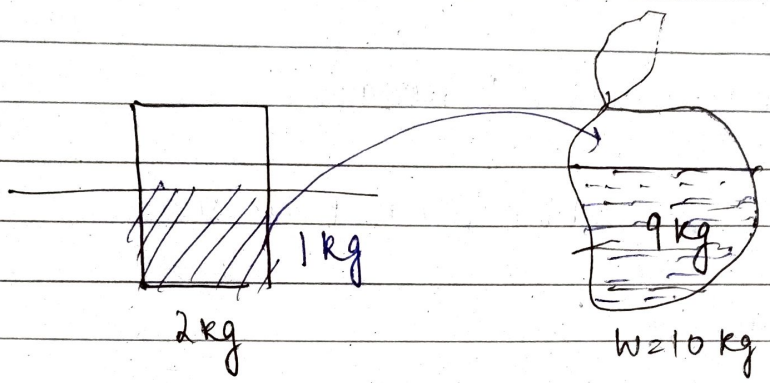
Max Profit = ?



P_1 P_2 P_3 P_4

W_1 W_2 W_3 W_4

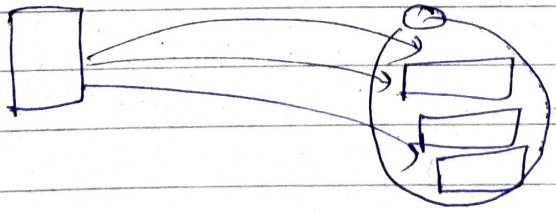
Fractional knapsack



Greedy approach

Unbounded knapsack

Unlimited supply of every item



0-1 knapsack

i) How to identify

wt[] : 1 3 4 5 W : 7kg

val[] : 1 4 5 7

o/p : max profit

1) Choice → (either put in knapsack) (either don't put in knapsack)

2) Optimal (max profit)

DP: Recursive → Memoization (DP) → Top down (DP)

DP → recursion + storage

0-1 knapsack Recursive

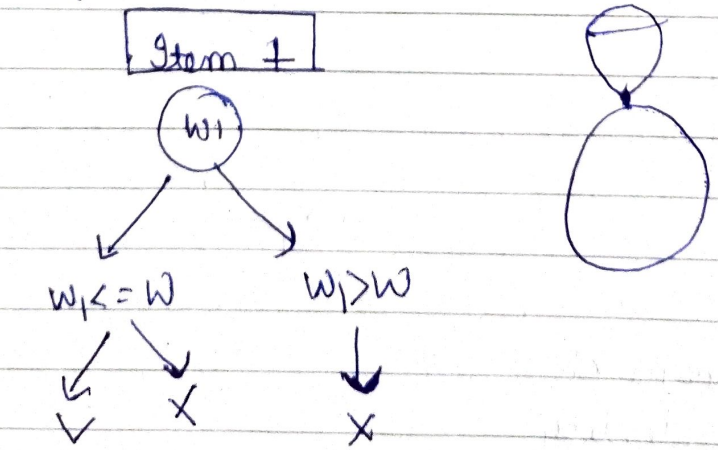
Identify

DP → Recursive → DP (top down)
→ DP (memoization)

I/P wt[] = [1|3|4|5]
val[] = [1|4|5|7]
W = 7 kg (capacity of knapsack)

O/P → Max Profit

Choice diagram



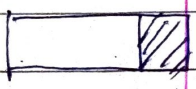
We have to return the max profit so return type would be int.

Base cond^n → think of the smallest valid i/p.

I/P wt[]: [ ][ ][ ] → (n) → 0.
val[]: [ ][ ][ ]
W : 10kg ⟶ 0 kg

```
int Knapsack ( int wt[], int val[], int W, int n) {
// base condition
  if (n == 0 || W == 0)
      return 0;

// choice diagram

  if (wt[n-1] <= W) {        → weight less hain & including
    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) {    → weight less hai but don't include
    return Knapsack (wt, val, W, n-1);
  }
```

max profit



01 Knapsack Memoization

Memoization = Recursive + 2 lines

t

func (I/P doesn't change , I/P change)

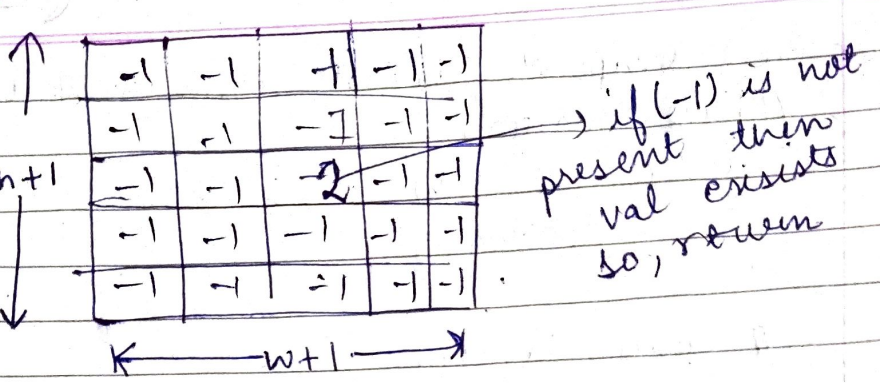
Val of n, m {

n-1

W - wt[n-1]

KnapSack (wt[], val[], n, W)

t [n+1] [W+1]



initialise this matrix with -1.

```
int t[n+1][w+1]
memset (t, -1, sizeof(t))
```

Changes

Now declare the matrix globally.

```
int static t[102][002]; constrainst
                        n<=100
memset (t, -1, sizeof t)   w<=1000

int knapsack (int wt[], int val[], int w, int n)
  {
    if (n==0 || w==0)
        return 0;
    if (t[n][w] != -1)
        return t[n][w];

if (wt[n-1] <= w)
return t[n][w] = max (val[n-1] + knapsack (wt, val, w-wt[n-1], n
                 , knapsack (wt, val, w, n-1));
```

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

}

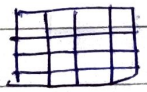
The complexity of top down & memoization remains same but the problem with memoization is the stack gets full. due to repeated func^{n} calls.

(0-1 Top Down Knapsack) → Real DP

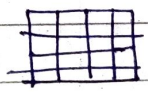
Recursive → Memoize → Top down → 6 Problems

↓ BC + recursive calls

↓ RC + table

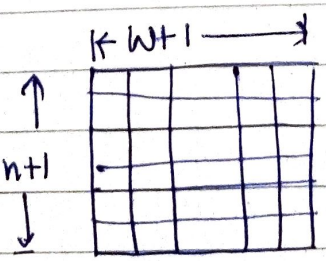


↓ only Table



Recursive → Memoization — Initialising matrix with -1

→ Top-down (totally omit the recursive call & use the table only)



We only make table for the i/p which is changing.

2 steps to make table for top down

Step1: Initialization

Step 2: Recursive code $\xrightarrow{\text{changes}}$ Iterative code

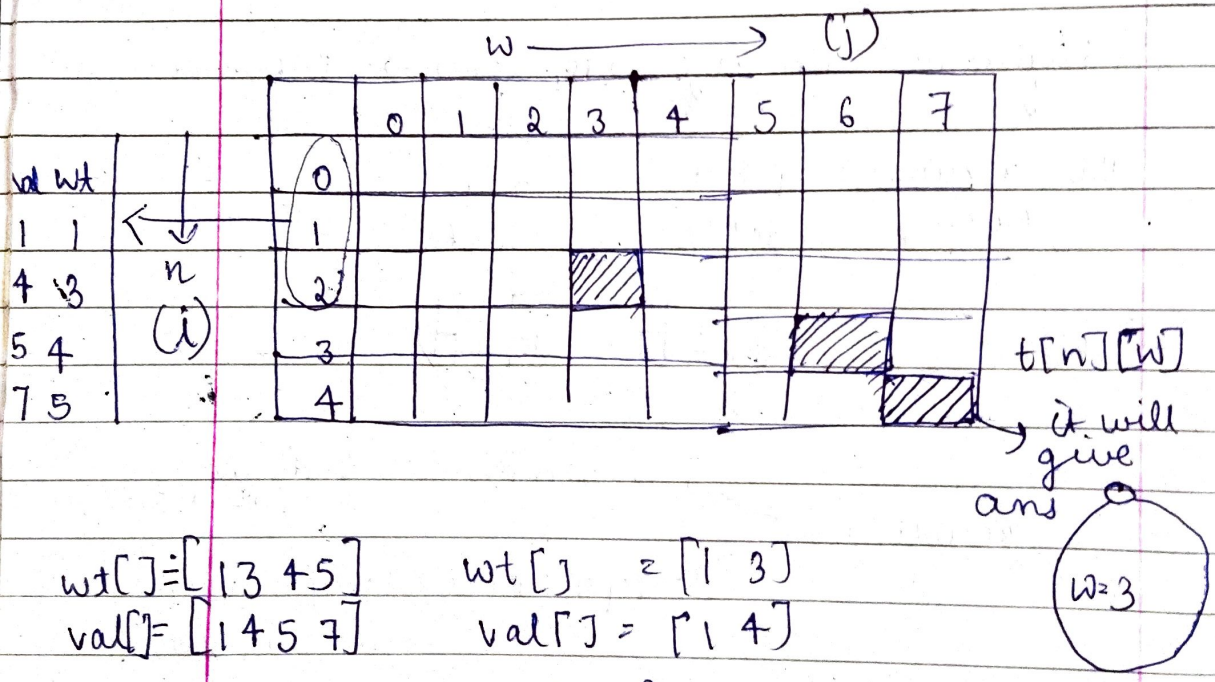
Step1: Initialize

W = 7

n = 4

wt[] = [1 3 4 5]

val[] = [1 4 5 7]



wt[] = [1 3 4 5]

val[] = [1 4 5 7]

wt[] = [1 3]

val[] = [1 4]

W = 3

wt[] = [1 3 4]

val[] = [1 4 5]

W = 6

RC + table ⟶ table

Base cond^n ⟶ Initialization

RC

Base cond^n

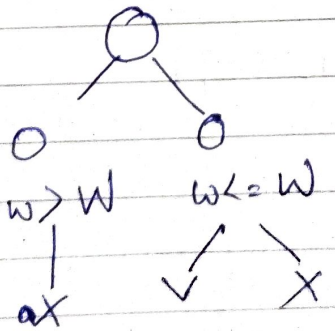
```
if(n==0 || W==0)
    return 0;
```

table

	w=0	1	2	...
1	0	0	0	0
2	0			
3	0			
4	0			

```
for(int i=0; i<n+1; i++){
    for(int j=0; j<w+1; j++)
    {
    if(i==0 || j==0)
        t[i][j]=0;
    }
}
```

Choice diagram



n, w ⟶ i, j

i++

j++

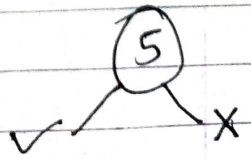
dp[i][j] RC.

wt = 1 3 4 5

val = 1 4 5 7

W = 7

max(val[n-1] + dp[i-1][j-wt[i-1]], dp[i-1][j])



Recursive

```
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)
```

Top down

```
if (wt[n-1] <= w)
t[n][w] = max( val[n-1] +
          t[w-wt[n-1]]
            [n-1],
          t[n-1][w])
else
t[n][w] = t[n-1][w]
```

0 1 2 3

wt = 1 3 4 5 W = 7

val = 1 4 5 7 n = 4

W = 0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

Top-down approach

pseudo code

```
int t[n+1][w+1];
for (int i=1; i<n+1; i++)
    for (int j=1; j<w+1; j++)
        {
        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]
return t[n][w];
```

i=1 wt[i-1] <= 1 → true

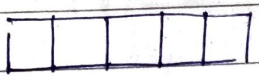
t[i][j] = max(1 + t[0][0], t[0][1])

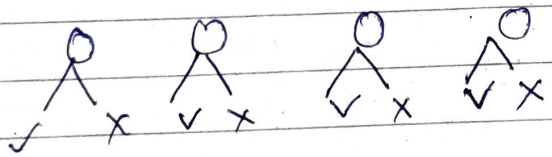
= max(1, 0)

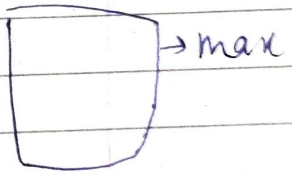
= 1

Identification of knapsack problem

1) Subset sum problem
2) Equal sum partition
3) Count of subset sum.
4) Minimum subset sum diff
5) Target sum
6) No of subset with a given diff.

Ip: Item array : 
W: Capacity





1. Subset Sum problem

arr[]: 2 3 7 8 10
Sum = ...

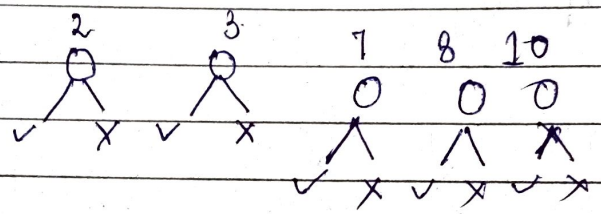
a) Problem statement
b) Similarity with knapsack
c) Code variation

1) Problem statement : Find if there is a subset present in an array with given sum.

arr[]: 2 3 7 8 10
Sum : 11

2) Similarity

item array[] : → 2 3 7 8 10
weight : capacity → 11



3) Code variation

t[n+1][w+1]

t[5+1][11+1]
t[6][12]

Initialization:

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 ↓

Q → True/false

arr[]:
sum : 0
arr[] : 1
sum = 0 {}

arr[] : 2
sum = 0 {}

arr[];
Sum = 2

when array is empty sum can't be anything

arr[]: no elements

Sum: 1 → not possible

t[n+1][sum+1] n

Initialisation: for(int i = 1 ... n+1)

for(int j = 1 ... W+1)

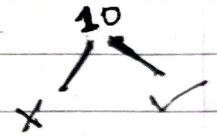
if(i == 0)

t[i][j] = false.

2 3 7 8 10

if(j == 0)

t[i][j] = True



Knapsack → arr.

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

t[i][j] = max(val[i-1] + t[i-1][j - wt[i-1]], t[i-1][j])

there is no max in true or false.

else

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

Subset sum

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

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

||

t[i-1][j]

else

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

return t[n][sum];

t[i][0] = t[i][0 - arr[0]]

||

t[0][0]

= t[i][-2] || t[0][0]

||

F

2, 3, 8

t[i][j]

{3,8} → true ✓

{3} → false ✗

2. Equal Sum Partition Problem

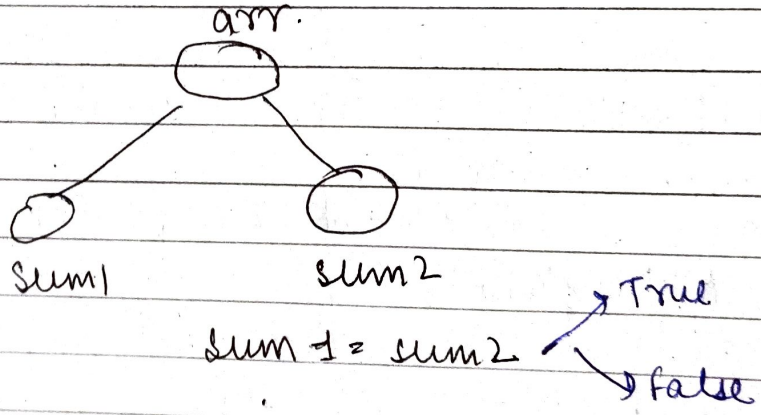
1) Problem statement
2) Subset sum similarity
3) Odd / Even significance
4) Code variation

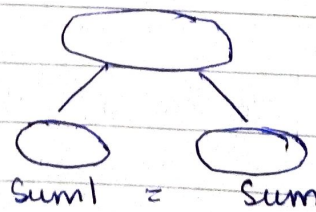
1) Problem statement

arr[] = {1, 5, 11, 5}

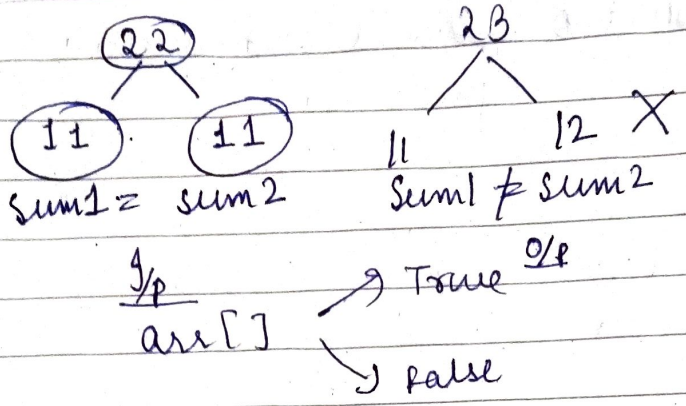
O/p : T/F

Is it possible to divide the array such that both the subset gives an equal sum

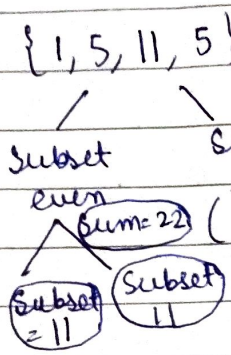




Sum 1 = Sum 2 → It is equal when the no is even. Sum of all the array elements should be even to change it into equal parts.




```
for (int i = 0; i < size; i++) {
        sum = sum + arr[i];
    }
```



(false) `if (sum % 2 != 0)` (sum is odd)
`return false`

`else if (sum % 2 == 0)`

* → we need to find one subset with sum 11 the next subset would automatically be 11.

`return subsetsum(arr, sum/2);`

3. Code:

```
i/p → arr[], n.
    int sum = 0;
for (int i = 0; i < n; i++)
        sum += arr[i];
if (sum % 2 != 0)
        return false
else
    return subsetsum(arr, sum/2);
```

3. Count of Subsets sum with a given sum.

I/p:

arr[] = 2 3 5 6 8 10

sum = 10.

Flow

1) Problem statement
2) Similarity to Subset sum
3) Code variation
   - Initialisation
   - Code
4) Return Type.

1) Problem statement

arr[] 2 3 5 6 8 10

sum : 10 O/p = 3.

{2,8} → Yes/True (in subset sum)

{2,8} = 10
{5,2,3} = 10
{10} = 10
} count = 3

2) Similarity

2, 8 | return count

Yes No

3. Code variation

Subset sum

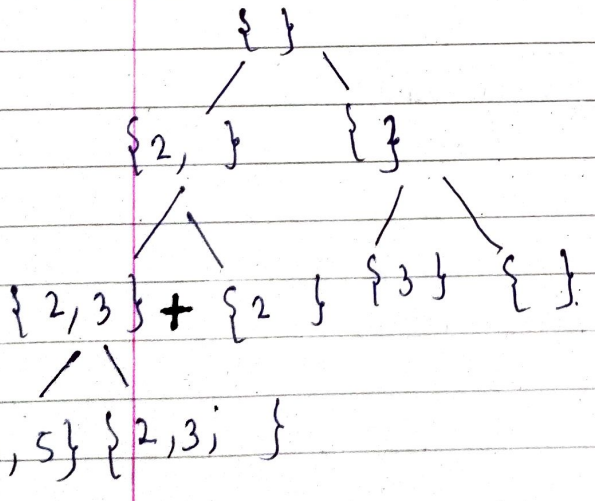
due to T/F ← bool

False → 0 (no of subset 0)

True → null subset (no of subset 1)

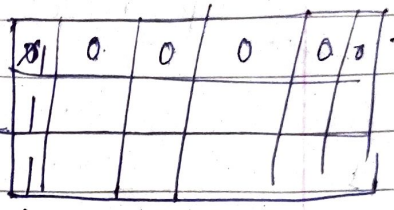
. . . . . .

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



Count

int



We will add all the subset so or would be changed to +. True & false case we can use ~~bool~~ or.

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

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

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

else

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

arr[] = 2 3 5 6 8 10

Sum = 10

O/p = 3

1, 3, 2, 5 5

O/p → {3, 2}

{5}

arr[0] <= 1

dp[N+1][sum+1] = dp[6+1][10+1]

= dp[7][11]

i=1 j=1 2-2=0

2 <= 1

i=1 j=2

arr[0] <= 2

2 <= 2 ✓

dp[0][0]

dp[0][2]

Sum →

	0	1	2	3	4	5	6	7	8	9	10
0	1	0	0	0	0	0	0	0	0	0	0
1	1	0	1								
2	1										
3	1										
4	1										
5	1										
6	1										

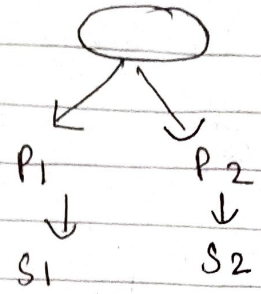
Minimum subset sum difference

1) Problem statement
2) Similarity
3) Solve using its previous concept +

1) Problem statement

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

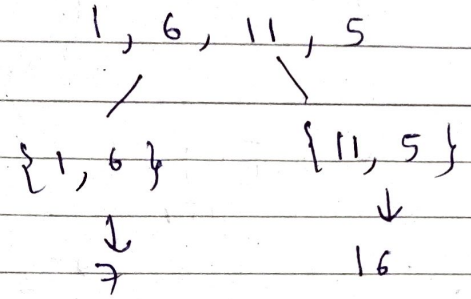
OP : 1



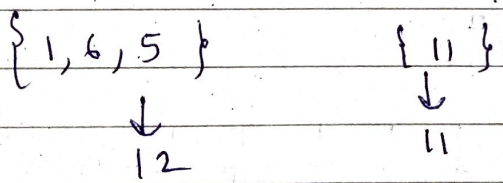
Equal sum: $S1 - S2 = 0$

Minm Subset: $S1 - S2 = min$

$abs(S1 - S2) = min$ (should be min)



$16 - 7 = 9$ → minimize of find could it be done in a better way.

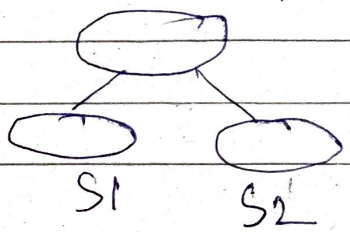


$12 - 11 = 1$ → Can't be minimized further

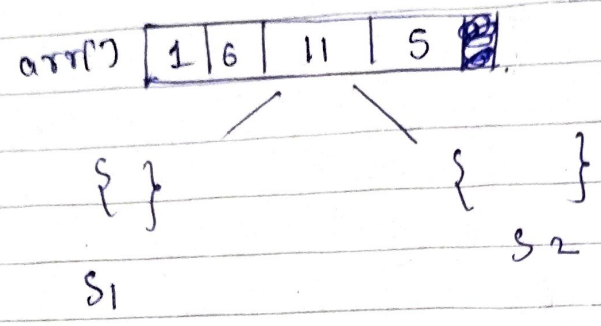
↳ O/P.

2) Similarity

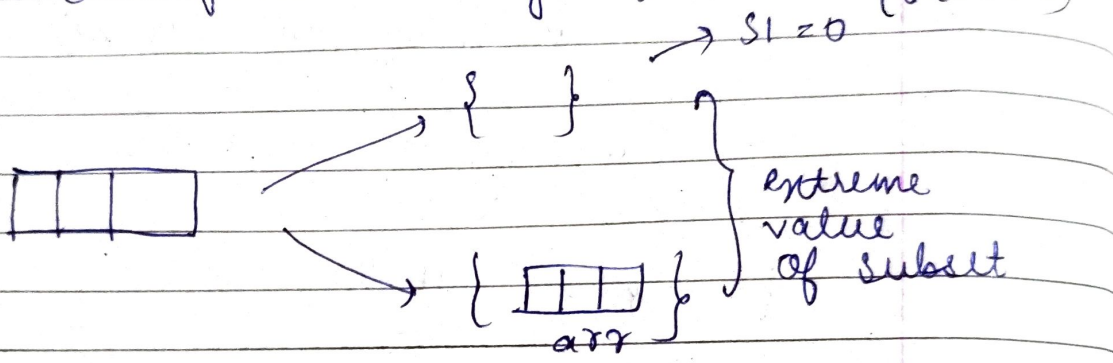
It is similar to equal sum partition.

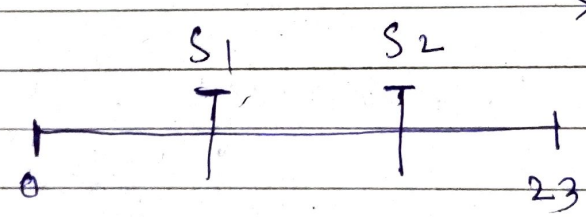


We need to find S1 & S2.

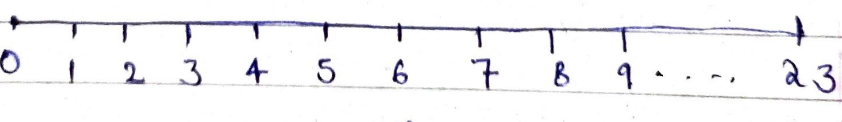


We can find the range of S1 & S2.

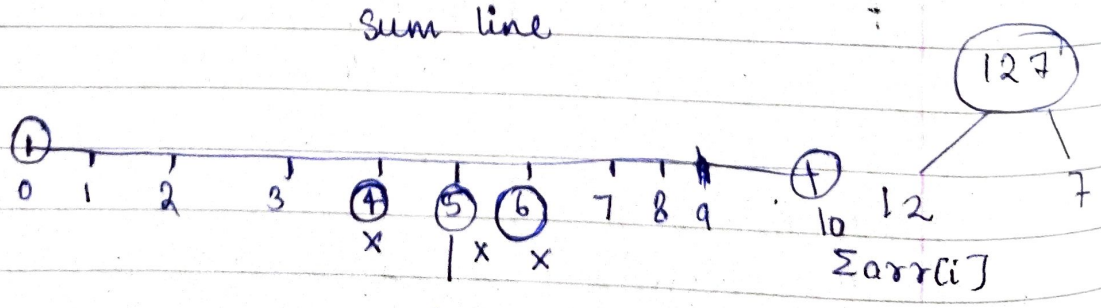




arr[]: {1, 2, 7}

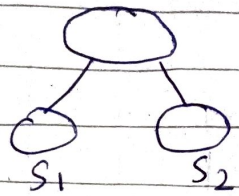


Sum line



S1/S2 can't be 5

S1/S2 = {0, 1, 2, 3, 7, 8, 9}

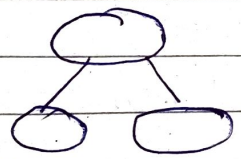


$S_1 + S_2 = \Sigma arr[i]$

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

S_1 (S_1) — Range - S_1 (S_2)

$S_1 - S_2$ = minimize

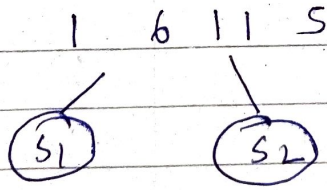


abs ($S_2 - S_1$) or abs ($S_1 - S_2$)

abs ($S_2 - S_1$) = minimize

abs (Range - S1 - S1) = minimize. abs (Range - 2S1) → minimize

Sum up

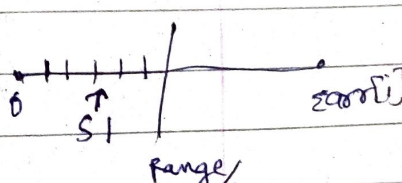


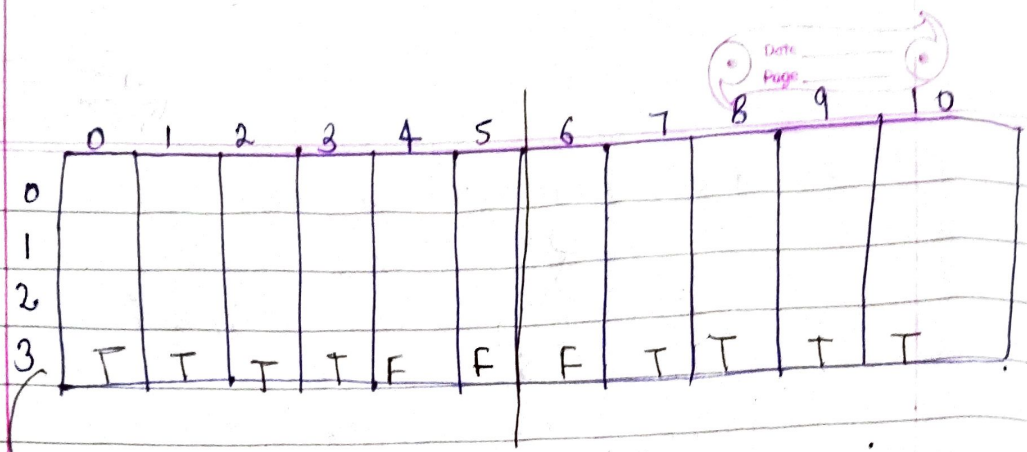
$S_1 - S_2$
$S_2 - S_1$ } minimize

↓

((Range - S1) - S1)

(Range - 2S1)



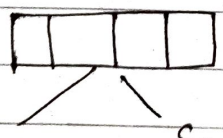


→ we will push the last row in vector till half way.

0	1	2	3.

```
int min = INT_MAX;
for (int i = 0; i < v.size(); i++) {
    {
        mn = min (mn, Range - 2v[i])
    }
return mn;
}
```

Concept



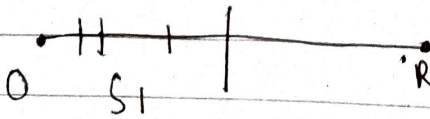
$(S_2 - S_1) = min$

↓

smaller

(S_1) $(Range - S_1)$

$Range - 2S_1 \rightarrow Minm$



Code

```
Subset sum( int arr[ ], int range)
{
last row filled in vec till halfway
```

0	0	0

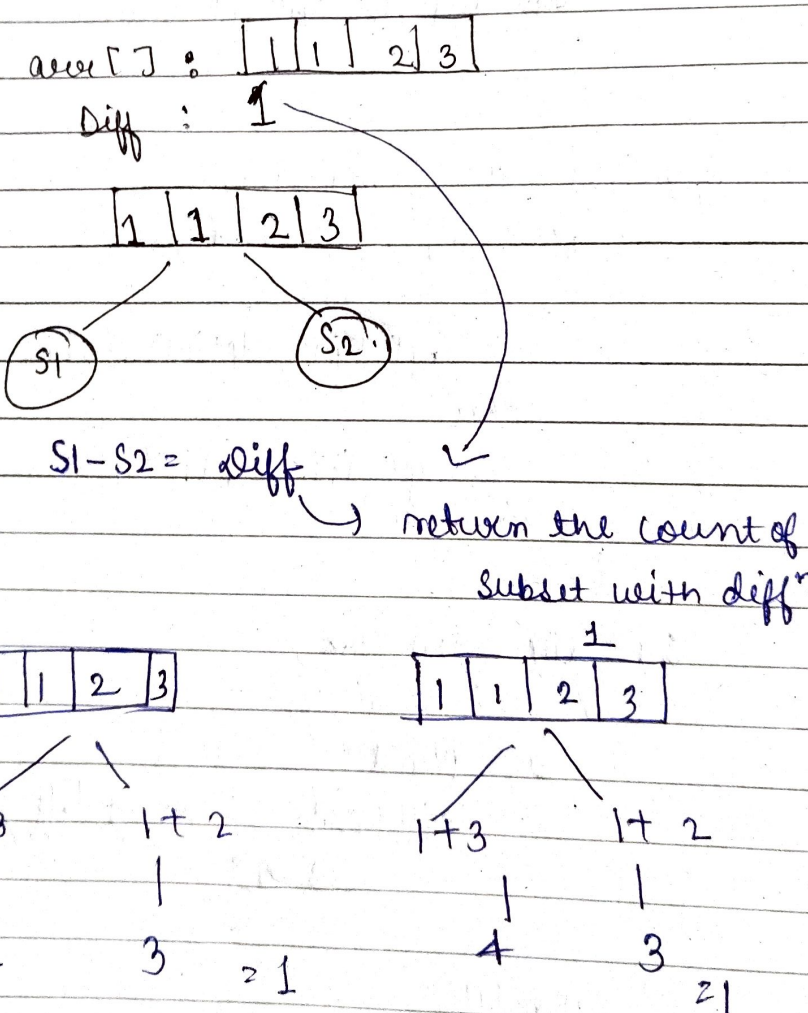
```
}
```

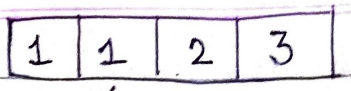
```
int mindiff (int arr[], int n) {
  int Sum = 0;
  for (int i = 0; i < n; i++) {
      Sum += arr[i];
    }
  bool dp[n+1][sum+1];
  for (int i = 0; i < n+1; i++) {
      for (int j = 0; j < sum+1; j++) {
      if (i == 0)   dp[i][j] = false;
      if (j == 0)  dp[i][j] = true;
      }
  }
for (int i = 1;  i < n+1; i++) {
      for (int j = 1; j < sum+1; j++) {
        if (arr[i-1] <= j)
          dp[i][j] = dp[i-1][j-arr[i-1]] || dp[i-1][j];
        else
          dp[i][j] = dp[i-1][j];
      }
    }
  int diff = INT_MAX;
  for (int j = sum/2; j >= 0; j--) {
      if (dp[n][j] == true) {
        diff = min(sum - 2*j, diff)
      }
  return diff;
}
```

Count the number of Subset with a given diff

1) Problem statement
2) Will try to reduce the actual statement
3) Solve it using already solved problem.

1) Problem statement





(diff) S = Sum of arr

S_1 S_2

$Sum(S1) - Sum(S2) = diff$

$Sum(S1) + Sum(S2) = S$

$2S_1 = diff + Sum(arr)$

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

$= \frac{1+7}{2} = \frac{8}{2} = 4$

$S_1 = 4$ $S_2 = S_1 - diff$

$= 4 - 1$

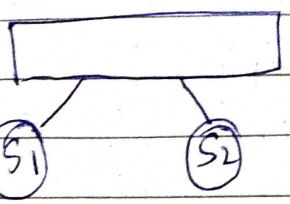
$= 3.$

Count = ?

- Find count when sum of S1 = 4

no of count of Subset with given diff ⟶ count of Subset sum.

Sum up.



$S_1 - S_2 = diff$

$S_1 + S_2 = sum(arr)$

$2S_1 = diff + sum(arr)$

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

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

return countofsubsetsum(arr, sum);

int countofsubsetwithdiff (int arr[], int n){
int diff

int arrsum = 0
for (int i=0; i<n; i++){
arrsum += arr[i];
}

int sum = 0;
sum = $\frac{diff + arrsum}{2}$.

return countofsubsetsum(arr, sum, n);
}

int countofsubsetsum(int arr[], int sum, int n){
int dp[n+1][sum+1];
for (int i=0; i<n+1; i++){
for (int j=0; j<sum+1; j++){

if (i==0) dp[i][j] = 0;
if (j==0) dp[i][j] = 1;
}
}

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

        if (arr[i-1] <= j) {
        dp[i][j] = dp[i-1][j - arr[i-1]] + dp[i-1][j];}
        else {
        dp[i][j] = dp[i-1][j];
        }
    }
}
return dp[n][sum];
}
```

Target Sum.

arr : | 1 | 1 | 2 | 3 |

Sum : 1

+/-

+c : [0, 0, 0, 0, 0, 0, 1]

target = 1

o/p : 3

Target value = 2

+1 -1 -2 + 3 = 1

-1 + 1 - 2 + 3 = 1

+1 + 1 + 2 - 3 = 1

{0, 2}

(+0, 2) (-0, 2)

count = 2

arr[] → | + | - | + | + |

{0, 0, 2}

{+0, +0, 2} {+0, -0, 2} {-0, +0, 2}

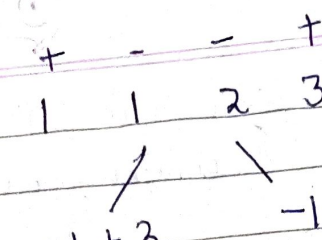
{-0, -0, 2}

= 4 (count

S1 S2

S1 - S2 = diff

So your answer is increased by a power of (2^1 no of zeroes) pow(2, no of zeroes)



(1+3) - (1+2)

S1 - S2 → count of subset with given diff n.

~~0/1 Knapsack~~

```
int findTargetSumWays(vector<int>& nums, int S){
    int cnt = 0, sum = 0;
    int n = nums.size();
    for(int i = 0; i < nums.size(); i++){
        sum = sum + nums[i];
        if(nums[i] == 0)
            cnt = cnt + 1;
    }
    S = abs(S);
    if(S > sum || (S + sum) % 2 != 0)
        return 0;
    int s = (S + sum)/2;
    int dp[n+1][s+1];

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

Cnt the no of zeroes in subset

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

        if(nums[i-1]==0)
            dp[i][j]=dp[i-1][j];

        else if (num[i-1]>j)
            dp[i][j]=dp[i-1][j];

        else
            dp[i][j]=dp[i-1][j-nums[i-1]+
                        dp[i-1][j]
    }.
}.
return pow(2,int)*dp[n][s];
}.
```

if 0 is there in the array dont take it

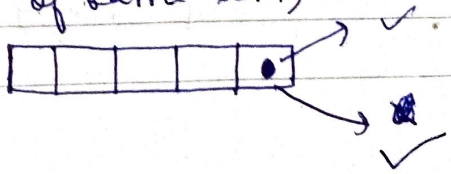
13. Unbounded Knapsack

Related Problems

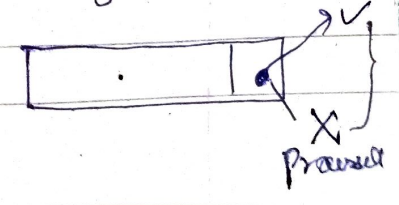
1) Road cutting
2) Coin change I (Max no of ways)
3) Coin change II (Min no of ways)
4) Max^m Ribbon cut

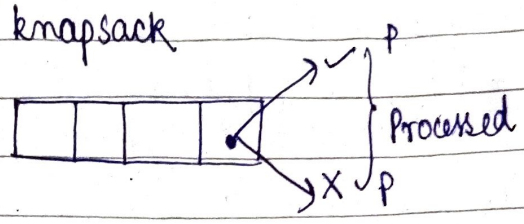
(Variations of unbounded knapsack)

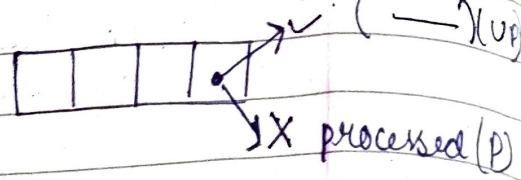
Unbounded (multiple occurences of same item)

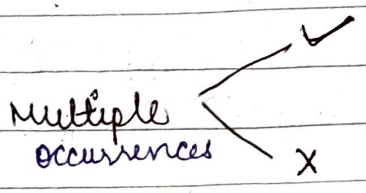


Knapsack (only one occurence)









Comparision betn.

Knapsack

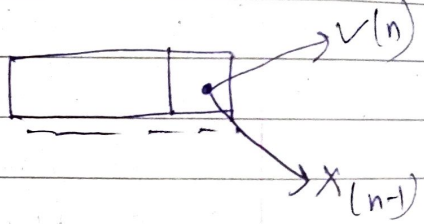
	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				

t[n+1][w+1]

```
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];
}.
```

Unbounded knapsack

	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				



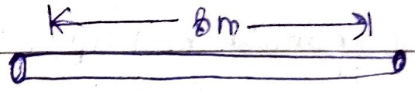

```
if (wt[i-1] <= j)
t[i][j] = max (val[i-1] + t[i][j-wt[i-1]],
                t[i-1][j] );
else
  t[i][j] = t[i-1][j]
```

14. Road Cutting Problem

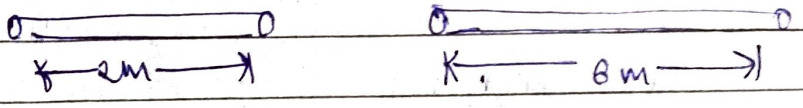
length [ ] :	1	2	3	4	5	6	7	8
price [ ] :	1	5	8	9	10	17	17	20

N : 8

1) Problem statement



length of rod is given. we need to cut it in such a way the profit is max^m.



Rs 5 Rs 17

17
5
22

Total = 5+17
= Rs 22

knapsack

wt
val
W

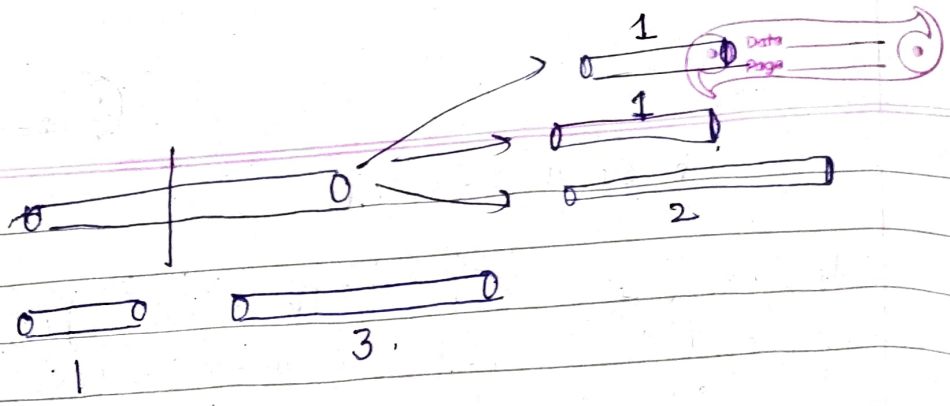
length = 1 to N
Price =
N = 8

if length array is not give make it & fill it with 1 to N.

knapsack
val[ ]
wt[ ]
W

Unbounded knapsack

price [ ]
length [ ]
N



Code variation

t[N+1] [N+1]

if (length[i-1] <= j)

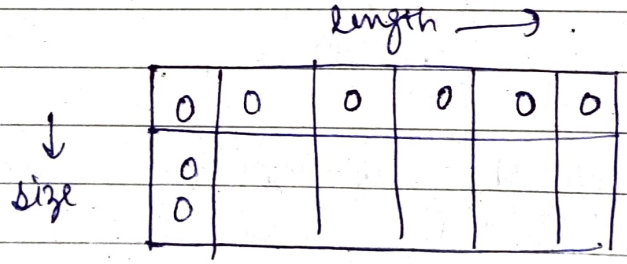
dp[i][j] = max(price[i-1] + dp[i][j- length[i-1]], dp[i-1][j])

else

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

Sometimes our size changes therefore we need to find the size of array.

dp[size+1][N+1].



Coinchange Problem

Maxm no of ways

Minm no of coins

Max^m no of ways.

Problem statement

coin[]: | 1 | 2 | 3 | → (unlimited)

Sum: 5.

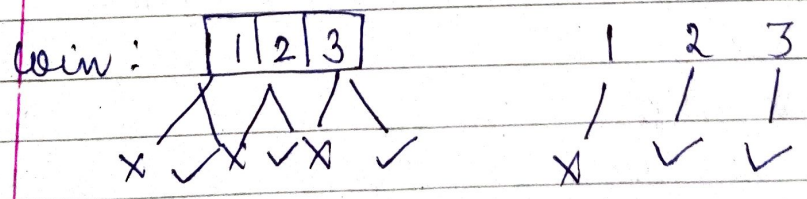
Use the coin array unlimited time & get the sum 5. Now find the max^m no of ways in which we can get 5.

5 ways.

	O/P
2+3	5.
1+2+2	5.
1+1+3	5.
1+1+1+1+1	5.
1+1+1+2	5

Q Why knapsack?

→ Every coin has a choice to include or not



item — wt[] ✓

item — val[] ✗ (when one array is given)

Matching

wt[] ⟶ coin[]
W ⟶ sum

1	2	3.

Sum 5 : 1 + 1 + 3

one item used many times

If taking one item many times does the sum allows then it is unbounded knapsack.

Subset sum

1	2	3	5

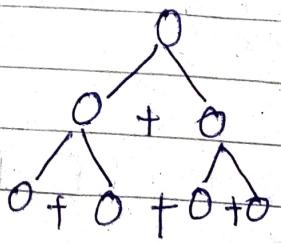
Sum : 8 → T / F.

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];
}
```

for count we remove || by +.

count / no of ways



We need to find maxim. no of ways.

Matching → knapsack → 0-1
→ Unbounded

wt → coin
W → sum.

```
if ( coin[i-1]<=j )
  t[i][j] = t[i-1][j] + t[i][j-coin[i-1]].
else
  t[i][j] = t[i-1][j]
```

size ↓ \ sum →	0	1	2	3	...	...
0	1	0	0	0	0	0
1	1					
2	1					
3	1					

Coin change-II (Minm no of coins)

coin[] = [1 | 2 | 3]

sum = 5

2+3 → 5 → 2 coins
1+2+2 → 5 → 3 coins
1+1+1+2 → 5 → 4 coins
1+1+3 → 5 → 3 coins
1+1+1+1+1 → 5. → 5 coins

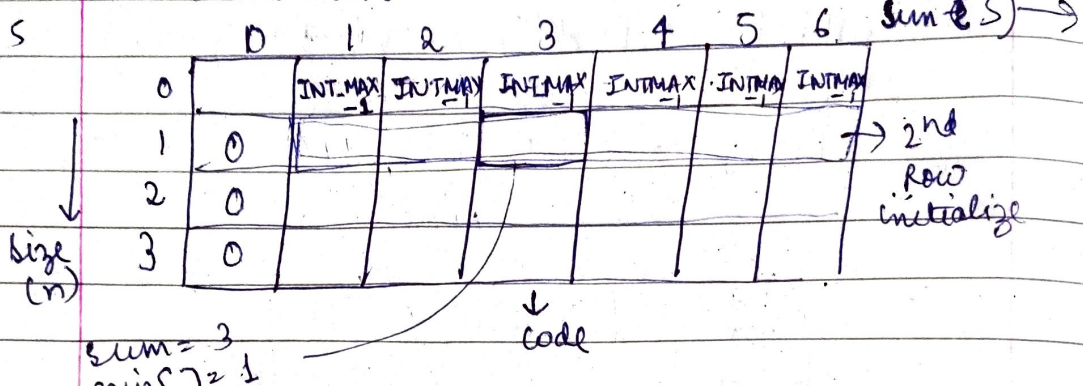
} Find minm. no. of coins to make sum as 5.

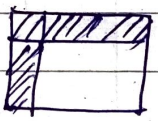
coin[] = [1 2 3]
Sum = 5
O/p : 2 (2+3 = 5)

Initialisation: t[n+1][w+1]
↓
t[n+1][sum+1]

wt → coin[] = n
val → x
W → sum.

n=3
sum=5



Initialisation  + Twist

coin[] : empty
sum : 1
} → INT_MAX (∞ coins)

coin[] : empty
sum : 0
} → INTMAX - 1

coin[] : 1
sum : 0
} 0 coins

coin[] : 1 2
sum : 0
} 0 coins

For Quee

sum = 5.
arr = [3, 5, 2]

when
arr [3]
sum = 4

$\frac{3}{3} = 1$

$\frac{4}{3} = INT\_MAX$

i \ j	0	1	2	3	↑	5
0	↓	INT_MAX	-1	→		
1	↑	1				
2	0					
3	↓					
4	↓					

sum = 1] 1
coin = 1

sum = 2
coin = 1 (1)

$\frac{4}{3} = INT\_MAX$

$\frac{j}{arr[0]} = \frac{3}{3} = 1$

size will always remains 1 so we will use arr[0] as it is the second row

For second row

```
for(int i=1; j<sum+1; j++) {
    if (j % arr[0] == 0)
        t[i][j] = j / arr[0]
    else
        t[i][j] = INT_MAX-1
```

$\frac{j}{arr[0]}$ — $\frac{3}{3} = 1$

$\frac{4}{3} = INT\_MAX$

Code variation

1st row → INT_MAX 1st col → 0

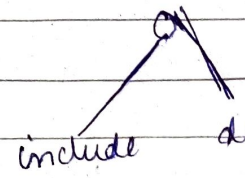
(INT_MAX-1)

if (coin[i-1] <= j)

t[i][j] = min(t[i-1][j], t[i][j-coin[i-1]] + 1)

else

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




```
int minCoins( int coins[], int M, int V) {
    // m = arrsize   v = sum.
    int dp [M+1][ V+1];
    for (int i=0; i<M+1; i++){
        for (int j=0; j<V+1; j++) {
            if (j==0) dp[i][j] = 0;
            if (i==0) dp[i][j] = INT_MAX-1;
        } }

    for(int i=1, j=1; j<V+1; j++) {
        if(j % coins[0] == 0) dp[i][j] = j/coins[0];
        else                  dp[i][j] = INT_MAX-1;

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

    if (dp [M][V] == INT_MAX-1) return -1;
    return dp[M][V];
}.
```

Longest Common Subsequence

1) Longest common substring
2) Print LCS
3) Shortest common supersequence
4) Print SCS
5) Min^m no of insertion and deletion a→b
6) Largest repeating subsequence
7) Length of largest subsequence of a which is a substring is b.
8) Subsequence pattern matching
9) Count how many times a appear subsequence in b
10) Largest Palindromic subsequence
11) Largest palindromic substring
12) Count of palindromic substring
13) Minimum no of deletion in a string to make it a palindrome
14) Minimum no of insertion in a string to make it a palindrome

Longest Common Subsequence (Recursive)

Problem Statement

X : a b c d g h
Y : a b e d f h r.

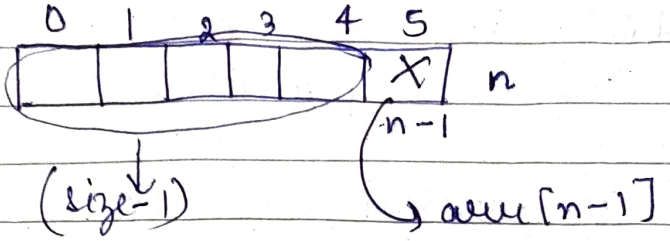
String X = abcdgh
String Y = abedfhr

LCS → abdh
↳ 4 (length of string)

diff chtr. → longest common subsequence - abdh.
cont chtr. → longest common substring - ab

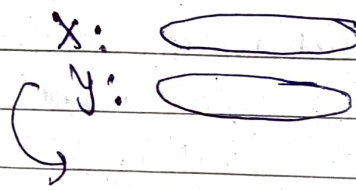
Recursive approach: Base condn + Choice diagram + i/p small

(x, y)




```
fun^n(x, y)
{
fun(x, y)
}
```

Base condn: Think of the smallest valid input.



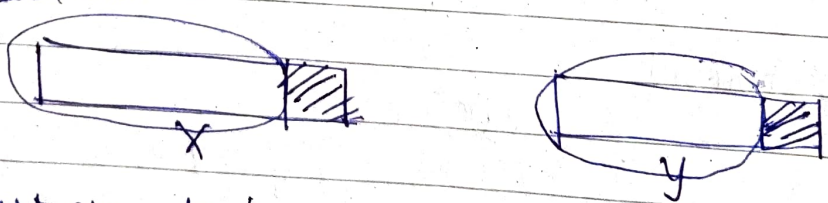
n = 0 m = 0 LCS = 0
(empty string)

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

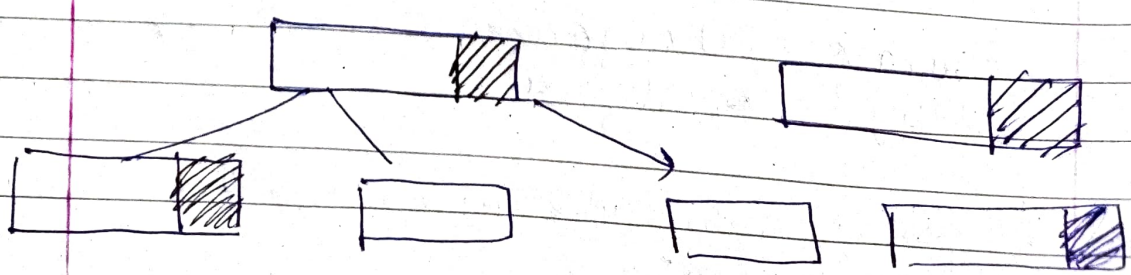
Choice diagram:

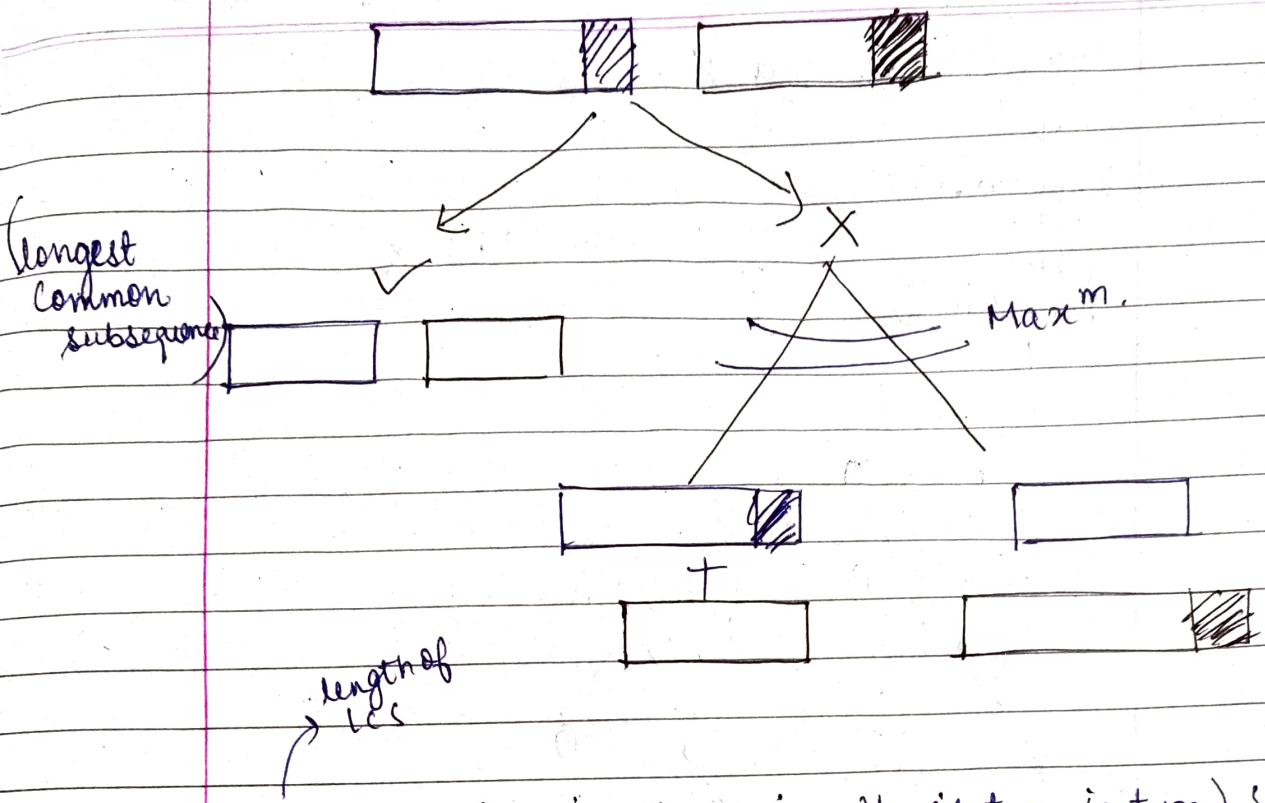
x : abcdgh
y : abedfhr

(1) When last char matches



(2) When last char don't match



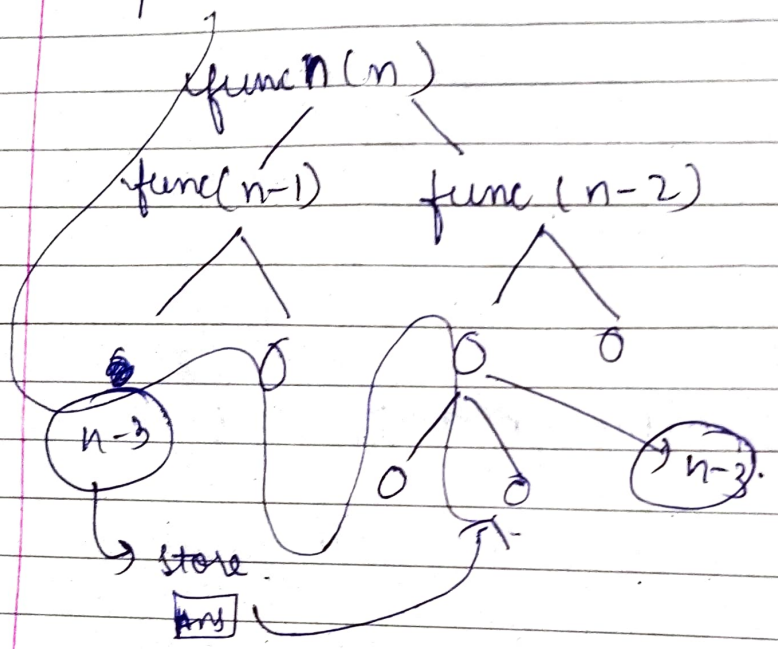



```
int LCS (string X, string Y, int n, int m) {

    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, m-1),
                    LCS( X, Y, n-1, m));

}
```

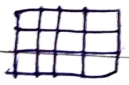
LCS memoization (bottom up approach)

func(n)
↓
func(n-1)
|
func(n-2)

```
func( )
{
func( )
}
```
→ DP X



R.C + table



"AXyT" "AyZx"

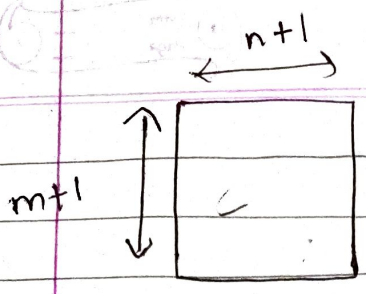
LCS(AXy, AyZx)

LCS(AXyT, AyZ)

LCS(AX AyZ) LCS(AXy, AyZ) LCS(AXy, AyZ)

LCS(AXyT, Ay)

Sub-Problem repeating twice



→ R((change)
(n+1, m+1).

```
int t[1001][1001];
```

-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

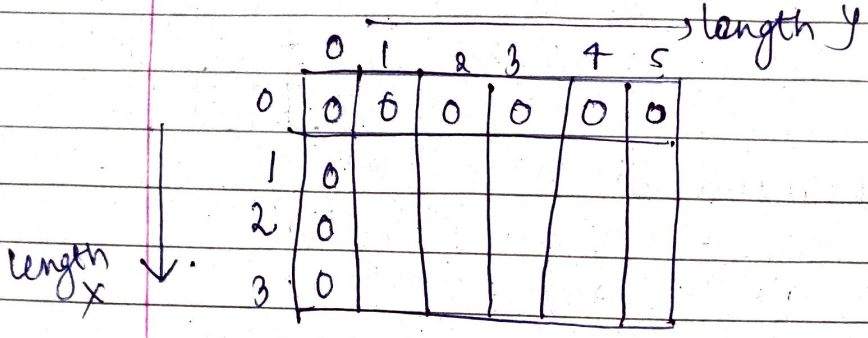
```
int main(){
  memset(t, -1, sizeof(t));
   LCS( )
}

int LCS( string X, string Y, int m, int n){

  if (n==0 || m==0)
        return 0;
  if (t[m][n] != -1)
        return t[m][n];

  if (X[m-1] == Y[n-1])
return t[m][n] = 1+LCS(X, Y, m-1, n-1)
  else
return t[m][n] = max( LCS(X, Y, m, n-1),
                      LCS(X, Y, m-1, n))

}
```

exponential → $O(n^2)$

LCS Top-down Approach

x : a b c d a f
y : a c b c f

o/p → 4 (abcf)



Base cond^n → initialization
(RS) (Top down)

```
int LCS( String X, String Y, int n, int m) {
    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 || 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

I/p a :→ abcde
b :→ abfce

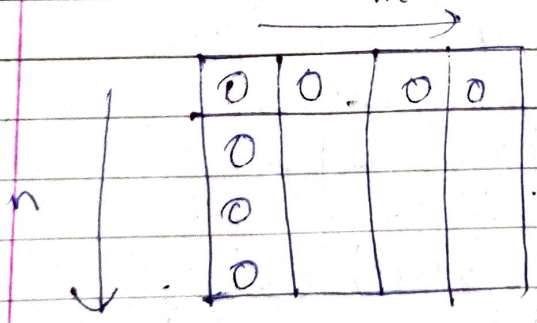
O/p → 2 (ab) ab, c, e

longest = ab

a b c d e
a b f c e

✗ ② ✗ ✗ 1

length = 0



discontinuity = 0

```
if (a[i-1] == b[j-1])
    dp[i][j] = dp[i-1][j-1] + 1
else
    dp[i][j] = 0
```

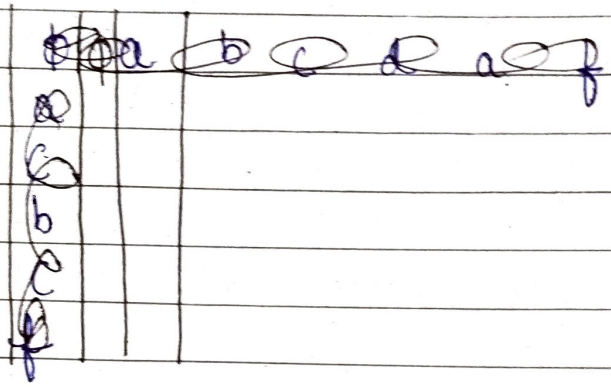
// for max^m val.

```
int maxi = 0;
for (i = 0 → n+1)
    for (j = 0 → m+1)
        maxi = max(maxi, dp[i][j])
return maxi;
```

Print LCS btwn 2 string

a : @ c b c f
b : @ b c d a f

o/p → a b c f



	@	a	b	c	d	a	f
@	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

Now, we need to print LCS

"f c b a"

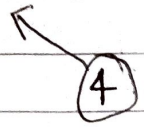
abcf ← reverse

if equal i, j → (i--, j--)

if not equal i,j → max ((i-1, j), (i, j-1))

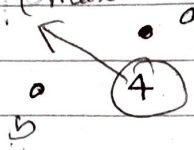
When equal

(i--, j--)



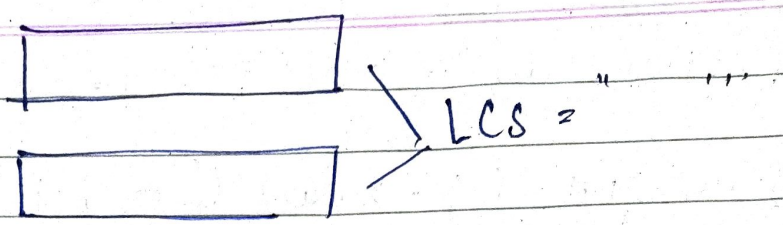
when not equal

(max bet'n a & b)



add in string : nothing to be added

```
int i = m , j = n ;
String s = "" ;
{
while (i>0 && j>0)
  { a      b
  if (t[i-1] == t[j-1])
    {
    s.push_back ( t[i-1])
      i--;
      j--;
    }
  else {
      if ( t [i][j-1] > t[i-1][j])
        j--;
      else
        i--;
    }
reverse ( s. begin(), s.end() ) ;
```



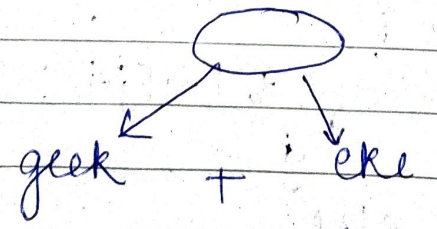

```
if (a[i-1] == b[j-1])
    S.push_back( a[i-1] )
        i--;
        j--;

else
    move in the direc^n of max'm
    reverse (s.begin(), s.end())
```

24. Shortest Common Supersequence

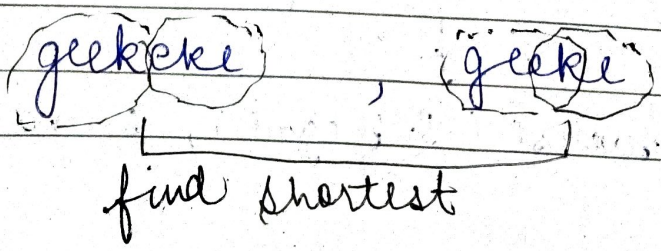
a: "geek"
b: "eke"

1) Problem statement

→ 2 string are given. We need to merge the string such that we get both the subsequence. Mix 2 sequence to make it a super sequence.



Op



Subsequence → Sequence of events

S_1 S_2 S_3

Order should be maintained but don't need to be continuous always

a : A G G T A B

b : G X T X A Y B

Super sequence : A G G T G X A B T X A Y B

A G G X T X A Y B → shortest supersequence

A G G T A B G X T X A Y B

Give the length in o/p.

The one letter which is common write once.

A G G T A B

G X T X A Y B

G T A B is common in both i.e LCS.

Now thinking the brute force approach we can merge both the string & then subtract G T A B.

a: AGGTAB b: GXTXAYB

length of SS
a+b

Worst case: a+b = AGGTAB GXTXAYB

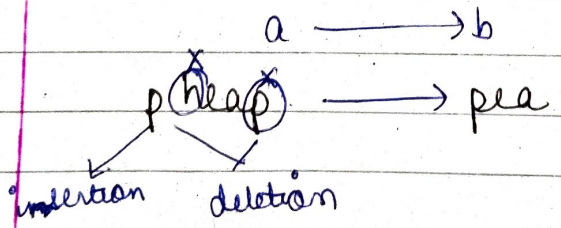
AGGTABGXTXAYB − GTAB

AGGXTXAYB

Shortest length = (m+n) − LCS

25. Minimum no of insertion & deletion to convert a string a to string b.

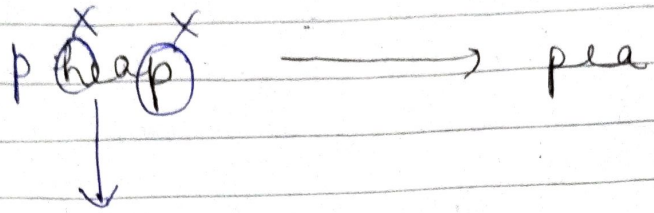
a: heap o/p = 1 (Insertion)
b: pea 2 (deletion)



When to use LCS.

	I/p	Q	o/p	
LCS Given Ques	a: ⬭ b: ⬭	LCS	int	→ Pattern matching algorithm
	a: ⬭ b: ⬭	insertion/ deletion	int	

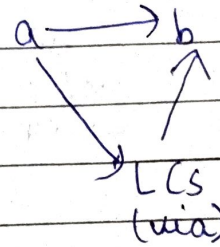
Convert heap to pea.

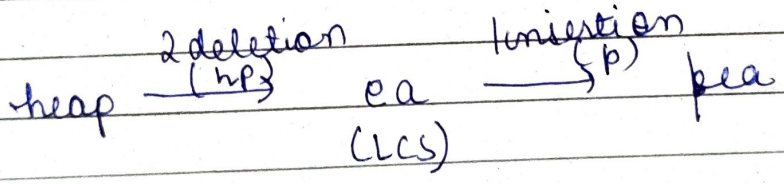


"ea" is the LCS of both string

heap → pea

Insertion/deletion





No of deletion = a length - LCS
No of insertion = b length - LCS

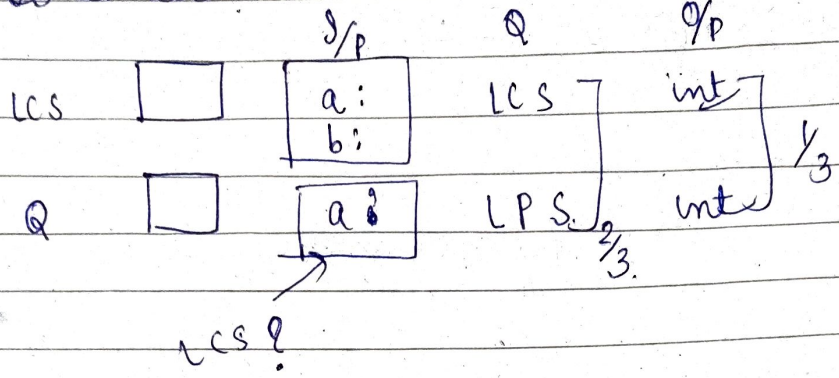
26. Longest Palindromic Subsequence

Problem Statement : S : agbcba
O/P = 5

5 ← longest [abcba ✓
bcb ✓
b ✓

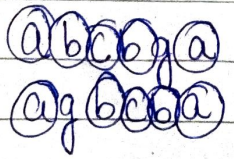
O/p → 5 (abcba)

2) Is it LCS?



LCS I/p a, b
LPS I/p a b=func(a)
(hidden string / redudant)

"agbcba"
↓
LPS
↙
a → agbcba
b → reverse(a) abcbga.
↓
LCS



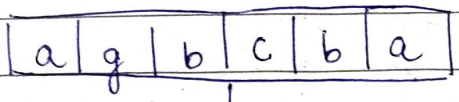
LPS(a) ≡ LCS (a, reverse(a))
LPS(agbcba) → return (agbcba,

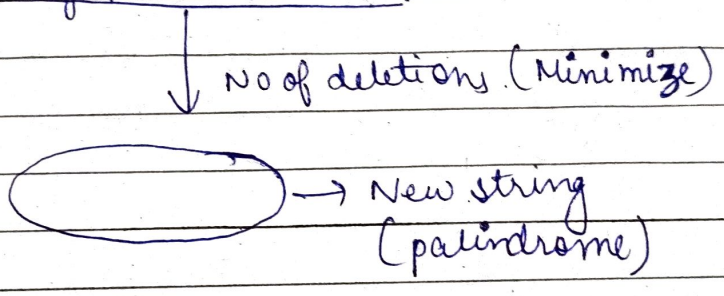
28. Minimum no. of deletion in a string to make it palindrome

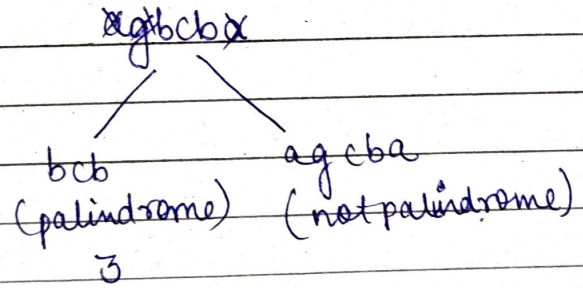
a : "agbcba"

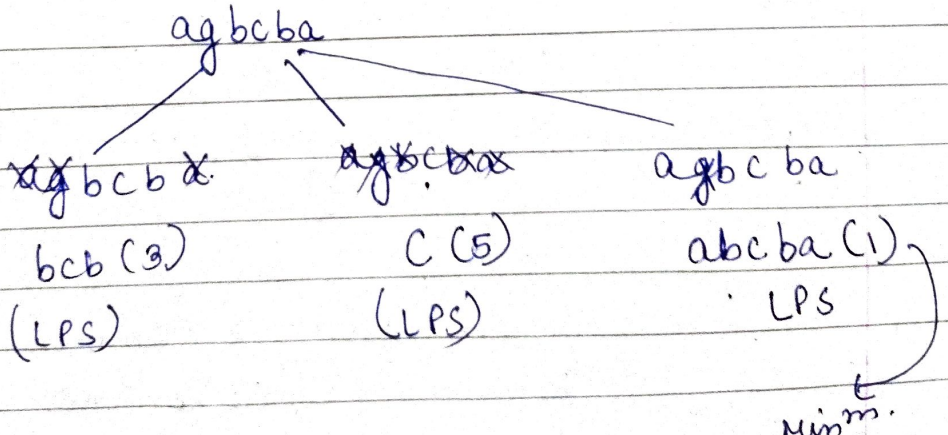
o/p = 1

S : agbcba



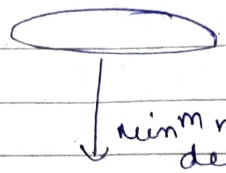






↑ length of LPS ∝ 1 / no of deletions ↓

LPS



→ Palindromic subsequence.

length of LPS ↑ ∝ 1 / no of deletions ↓

(longest palindromic subsequence)

agbcba

↓

LCS(s, reverse(s))

↓

abcba (LPS)

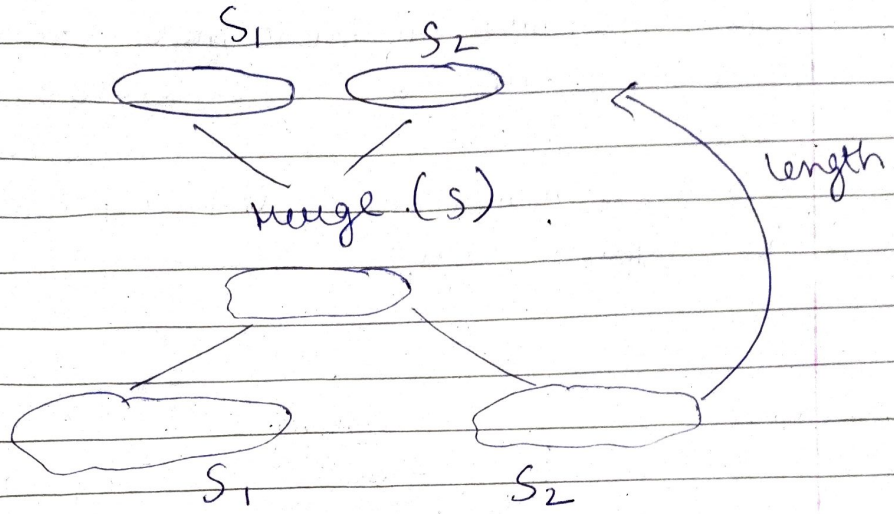
Min no of deletion = S (length) − LPS(length)

29. Print shortest common subsequence

i/p a: acbcf
b: abcdaf

o/p: acbcdaf

SCS



worst case

acbcf abcdaf

acbcdaf → print the whole string

m+n

acbcf abcdaf

↓

now

m+n-LCS.

LCS (similar) SCS

Print LCS Print SCS

	ϕ	a	b	c	d	a	f	
ϕ	0	0	0	0	0	0	0	→ LCS
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	

LCS → common ni hai to move kar jao;
common hai to print karlo.

SCS → common hai to ek bar print karo
else print karo.

LCS.

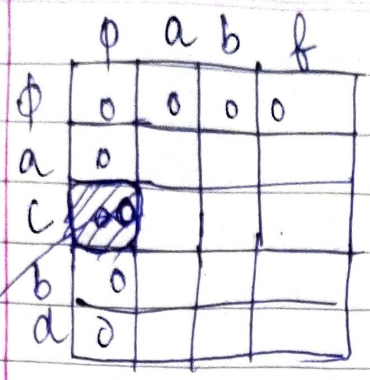
```
string s = " ";
while (i>0 && j>0){
  if (a[i-1] == b[j-1]){
    s.push_back(a[i])
      i--;
      j--;
    }
else{
  if (t[i][j-1] > t[i-1][j])
        j--;
  else if (t[i-1][j] > t[i][j-1])
        i--;
}
```

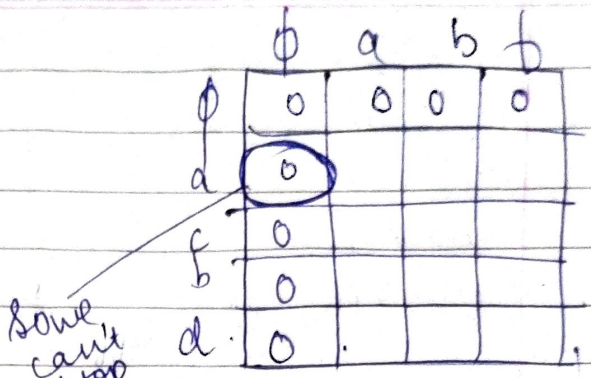
SCS

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

1) Now in SCS code we include the letters on moving at i-1 or j-1.
2) Now in (i>0 && j>0) we need to change because in case of LCS we don't need to stop at the topmost but in SCS we need to.

In case of LCS





Some can't stop here.

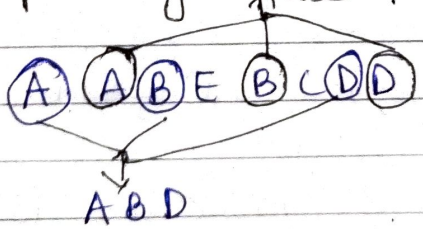
a c
" "] LCS (" ")

a c
" "] SCS ≠ (ab

Longest repeating subsequence

Str = "A A B E B C D D"
O/P = 3

Problem statement - Find a subsequence which is repeating



A B D (2X)
A B (2X)] longest → "A B D"
O/p = 3

S = A A B E B C D D

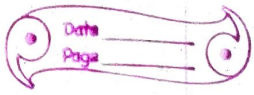
0 1 2 3 4 5 6 6
A A B E B C D D
A A B E B C D D

E → 3
C → 5

LCS = A A B E B C D D.
X X

A A B B D D
once ←→ cons

E & C are occuring once.
letter at the same index don't take



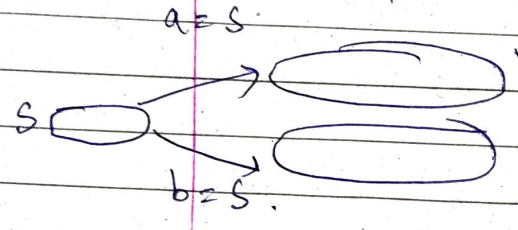
E → 3 → i, j i == j X
C → 5.

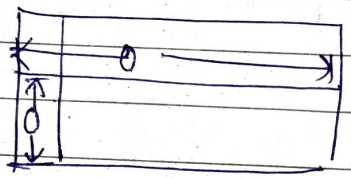
A → 0, 1
A → 0, 1
same index, diff index

B → 2, 4
B → 2, 4

So don't take A of same index take from other index.

Sum up:





We can't take same letter for both the string during LCS i.e. (i == j)

```
if (a[i-1] == b[j-1] && i != j)
    t[i][j] = t[i-1][j-1] + 1
else
    t[i][j] = max(t[i][j-1], t[i-1][j])
```

31. Sequence Pattern Matching

a = "AXY"
b = "ADXCPY"

O/P → T/F

Problem statement: Is string "A" a subsequence of "B"?

a : AXY b : ADXCPY

b : (A) D (X) C P (Y)
a : A X Y

O/p → True.

LCS : AXY

	I/p		
LCS	a b	sequence	int
Q	a b	sequence	T/F

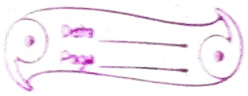
I/p
b : ADXCPY
a : AXY
LCS : AXY (LCS == a)

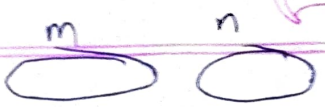
b : ADXCPY
a : AXYZ
LCS : AXY
(LCS != a)

```
if (LCS == A)  return true
else
    return false
```

Can it be done using length. ?

a, b → LCS





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

LCS = 0 to min(m,n)

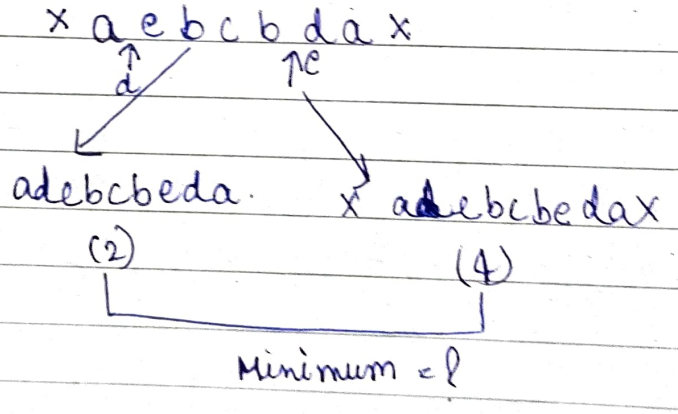
LCS = 2

```
if (LCS == a.length())
    return true
else
    return false.
```

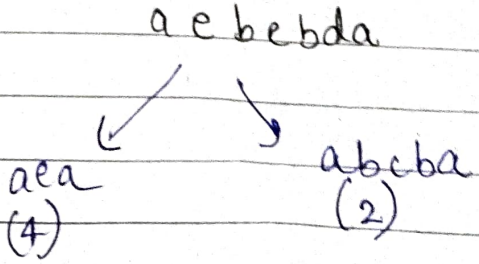
32. Min^m no of insertions in a string to make it palindrome

I/p: S: "aebcbda"

O/p → 2



Min^m. no of deletion to make a string palindrom.



S.length() → LPS. (deletion)

S : "a e b c b d a"
(a, b) → Palindromic string

[a] e [b c b] d [a]

e ✗ } delete
d ✗ } e & d

a e ✓ } insert e & d
d ✓ } & make their pair

no of insertions = no of deletions
= S.length − LPS.

Deletion → single (pair) → ✗
Insertion → single (pair) → ✓

33. MCM (Matrix Chain Multiplication)

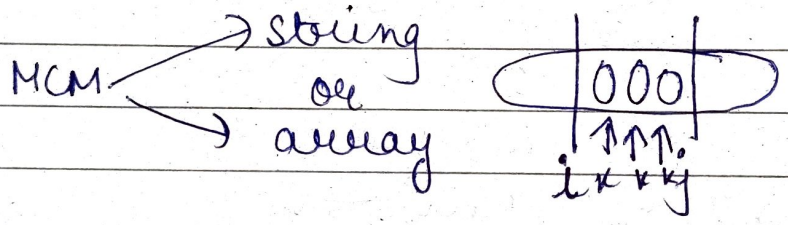
1) MCM
2) Printing MCM
3) Evaluate Exp. to True / Boolean Parenthesization
4) Min / Max value of an Expr
5) Palindrome partitioning
6) Scramble string
7) Egg Dropping problem

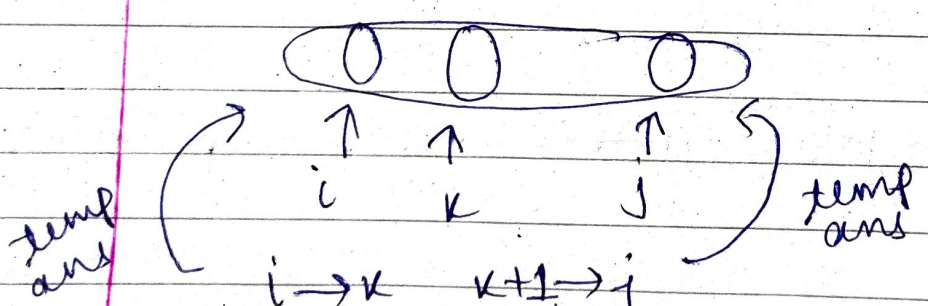
MCM format

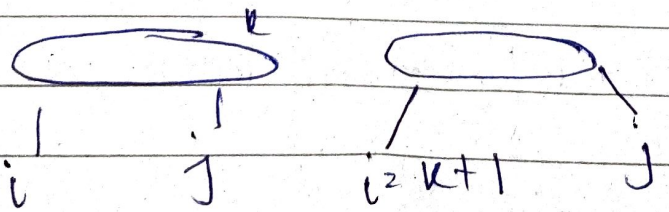
1) MCM
2) Printing MCM
3) Evaluate Exp^{n} to True / Boolean parenthesization
4) Min / Max value of an Expr
5) Palindrome partitioning
6) Scramble string
7) Egg dropping problem

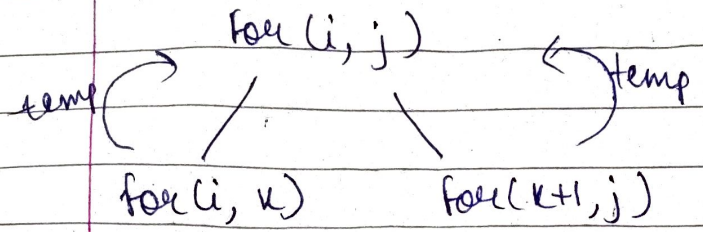
Identify → MCM → basic format

Identification + Format

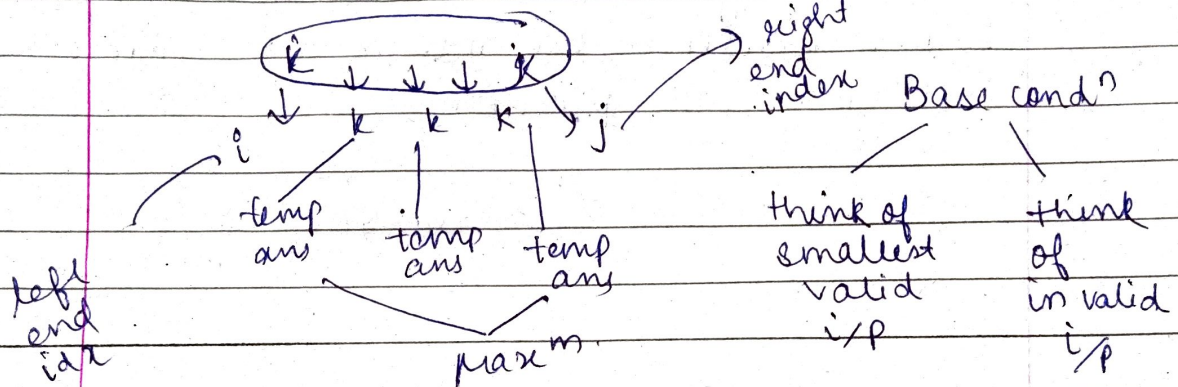








ans ← f (temp ans)




```
int solve ( int arr[], int i, int j)
  {
    if (i>j)  →  this may be diff acc^n to ques
    return 0;

  for (int k=i ; k<j ; k++)
    {
      // Calc. temp ans
        temp ans = solve ( arr, i, k) +
                   solve (arr, k+1, j);

        ans = func (temp ans)
    }
```

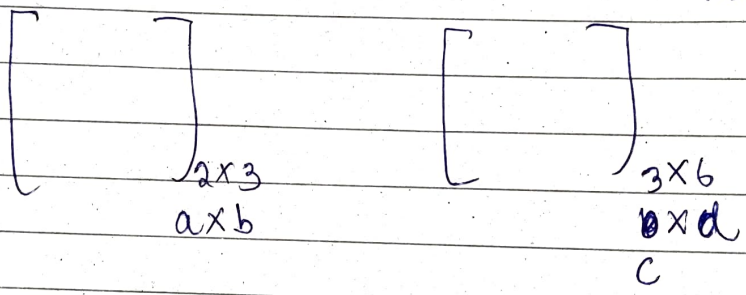
MCM (Recursive)

Problem Statement : arr[] = {40, 20, 30, 10, 30}.

A_1 (m×n), A_2 (m×m), A_3 (n×n), A_4

Some matrix are given like A_1, A_2, A_3, A_4 we have to multiply the matrix to reduce the cost (no of multiplications)

b = c (then only we can multiply)



2 × 3 3 × 6

↓ ↓ ↓

2 3 6 = 36 cost (no of multiplications)

A_1 A_2 A_3 A_4

dimension [] = {x, y, z, w}.

$A_1\ A_2\ A_3\ A_4$

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

C_1 C_2 C_3

$A \rightarrow 10 * 30$

$B \rightarrow 30 * 5$

$C \rightarrow 5 * 60$

$(A\ B)\ C = 10 * 30 * 30 * 5$

$10 * 5 \quad 5 * 60$

$5 * 5$

$(A\ B)\ C = (10 * 30 * 30 * 5)\ 5 * 60.$

$= 10 * 5 * 30 * 5 * 60$

$=$

$10 * 30 * 5 * 60 * 10 * 5$

$= 4500.$

$A_1 A_2 A_3\ A_4$

↓ do parenthesiation in such a way cost is the least

$A_1(A_2 A_3)A_4$ C_1

$(A_1 A_2)(A_3 A_4)$ C_2 $A(B\ C)$ → min cost

$A_1(A_2(A_3 A_4))$

C_3

$30 * 5 \quad 5 * 60$

$B\ C \rightarrow 30 * 5 * 60$

$A * B\ C \rightarrow 10 * 30 \quad 30 * 60$

$10 * 30 * 60$

cost

Bracket lagane par alag alag cost aa rahi hai.

0	1	2	3	4
40	20	30	10	30

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

arr size n n-1 matrix

$A_1 \rightarrow 40 * 20$
$A_2 \rightarrow 20 * 30$
$A_3 \rightarrow 30 * 10$
$A_4 \rightarrow 10 * 30$

$A_1 * A_2 * A_3 * A_4$

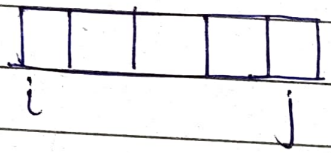
Cost (minm)

(no of multiplication should be less)

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

$A[i] = arr[0] + arr[1]$

Next step → Format



(dimension)

arr[]: | 40 | 20 | 30 | 10 | 30 |
0 1 2 3 4

$A_1\ A_2\ A_3\ A_4$

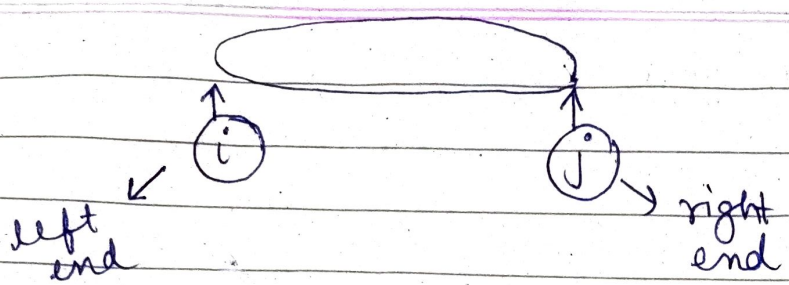
$(A_1)\ (A_2\ A_3\ A_4)$

k

Min cost + Min cost → temp ans

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

put brackets on different places to get the answer
slide k at different positions.



40	20	30	10	30

i i j

Ai → arr[i-1] * arr[i]

when i=0 Ai → arr[-1] * arr[0] ✗

when i=1 Ai → arr[0] * arr[1] ✓

Aj → arr[j-1] * arr[j]

arr[3] * arr[4]

i>j → size=0

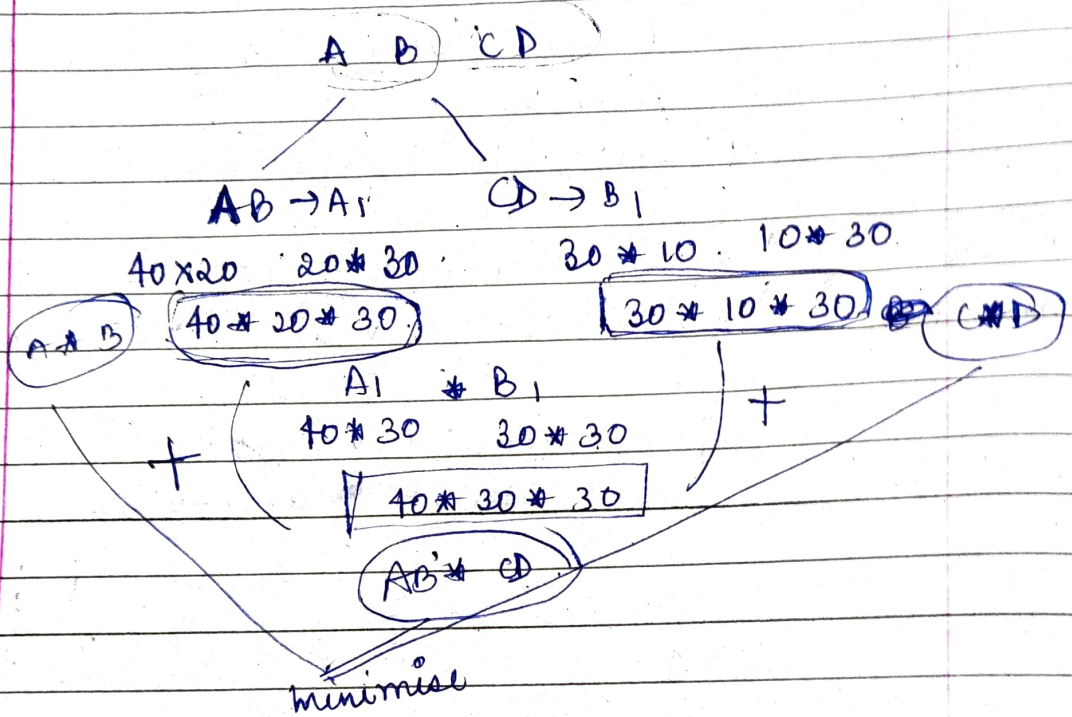
i==j → size=1 [(i-1) ni milega]

i = 1
j = n-1
} return cost

1 → i, n-1 → j

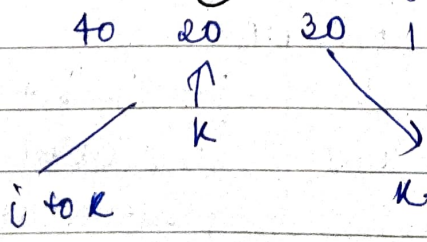
```
int solve ( int arr[], int i, int j)
{
if (i>=j)
   return 0;
   int mn = INT_MAX;
for(int k=i ; k <=j-1 ; k++){
   int tempans = solve(arr, i, k) +
                 solve(arr, k+1, j)
              + arr[i-1] * arr[k] * arr[j];

if(tempans)
   {
   mn = tempans;
}
```

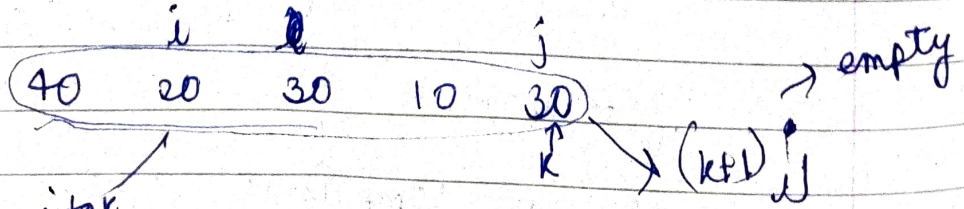
return mn.
}

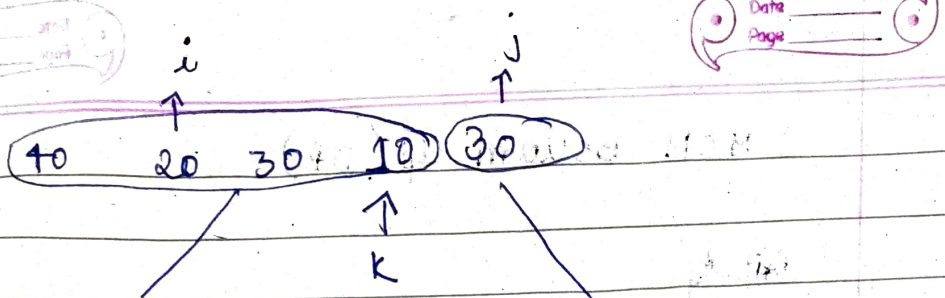


Steps for MCM

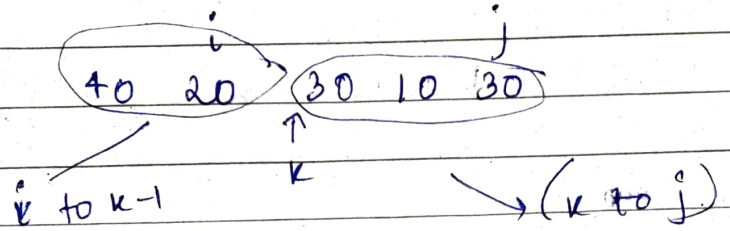
1) find i & j value
2) find right Base cond^n.
3) Move K → i to j. (find K-loop scheme)
4) Calculate (k) cost for (i) temp ans.







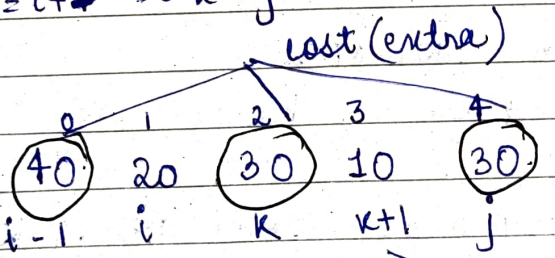
$k = i$ $k = j-1$ $(i \to k \quad k+1 \to j)$



2 schemes.

$k = i$ to $k = j-1$ | i to k | k+1 to j

$k = i+1$ to $k = j$ | i to k-1 | k to j



for (i to k)

40 * 20 * 20 * 30

↓

40 * 20 * 30.

Solve (i to k)

for (k+1 to j)

30 * 10 10 * 30

↓

30 * 10 * 30

Solve (k+1 to j)

40 * 20 * 30 * 30

arr[i-1] ← 40 * 30 * 30 → arr[j]

arr[k]

dimension arr: [ ]

MCM Bottom Up (DP)

(dimension) arr[ ] : [ ]

```
int dp[100][100]
memset ( dp, -1, sizeof(dp))
int solve ( int arr[], int i, int j)
  if (i <= j)
    return 0;
  if (dp[i][j] != -1)
    return dp[i][j];
  int mn = INT_MAX;
  for (int k = i; k<j; k++) {
  int temp_ans = solve(arr, i, k) +
                 solve (arr, k+1, j) +
                 arr[i-1] * arr[k] * arr[j];
  if (temp_ans < mn)
    mn = temp_ans;
  }
  dp[i][j] = mn;
return mn;
```

Palindrome Partitioning

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

class Solution {
public:
    int solve(int arr[], int i, int j) {
        if (i >= j)
            return 0;

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

        int mn = INT_MAX;
        for(int k = i; k <= j-1; k++) {
            int temp = solve(arr, i, k) + solve(arr, k+1, j)
                       + arr[i-1] * arr[k] * arr[j];
            mn = min(temp, mn);
        }
        dp[i][j] = mn;
        return dp[i][j];
    }

int matrixMult(int N, int arr[]) {
    memset(dp, -1, sizeof(dp));
    int ans = solve(arr, 1, N-1);
    return ans;
    }
};
```

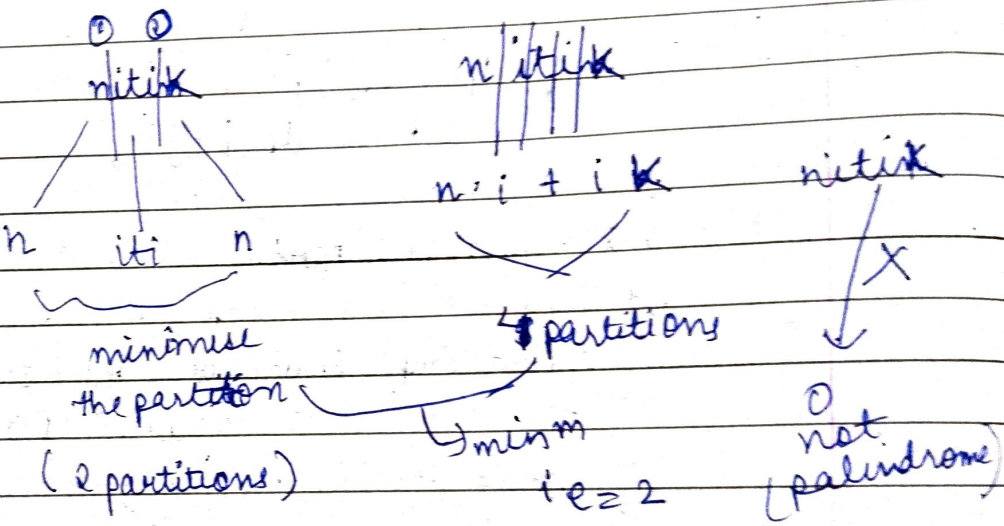
Palindrome partitioning

1) Problem statement

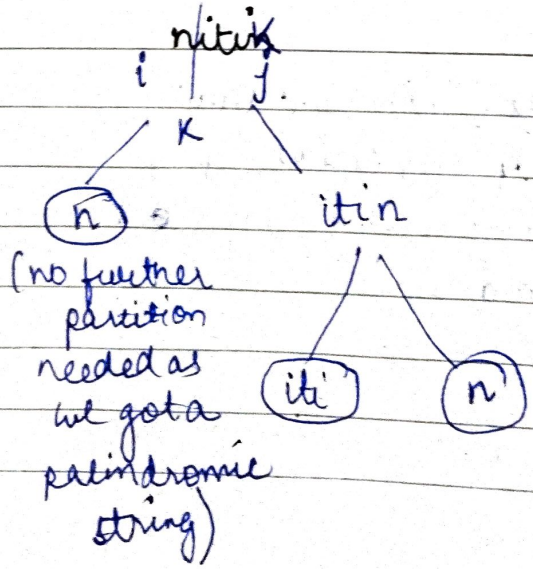
S : nitik → it is not a whole is a palindrome as

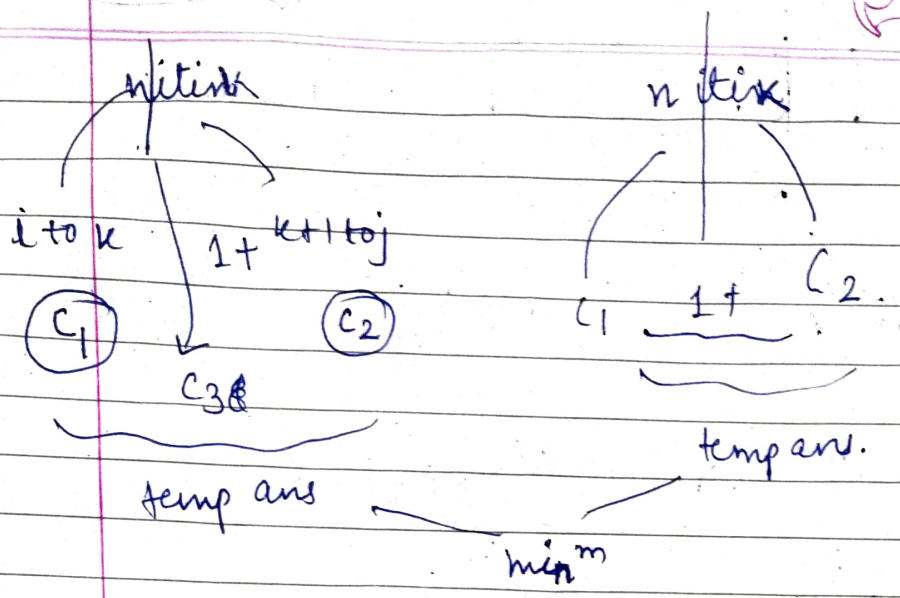
o/p : 2

divide string in such a way all the strings are palindrome



worst case partition : n-1





Format

1. Find i & j
2. Find BC.
3. Find K loop (scheme)

4. temp ans → min^m

Recursive code

n i t i K

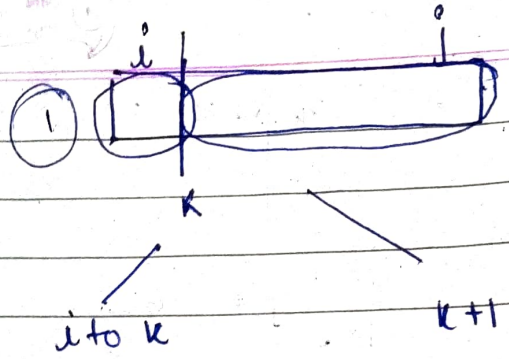
i j (no need to start i from 1 since it doesn't require i-1)

i=0 j=n-1

B.C i==j size=1
i>j size=0 } (if size is not given or 0 or equal no of partition would be 0 as string is empty)

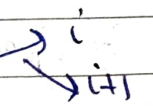
isPalindrome(s, i, j) & if palindrome partition should be 0.

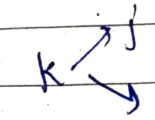
loop

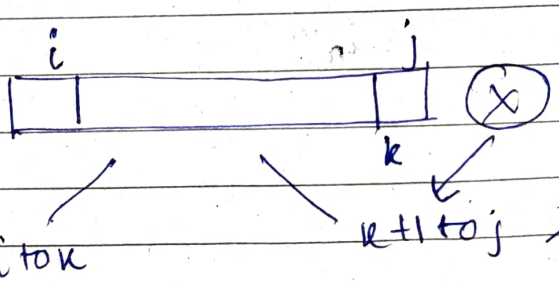


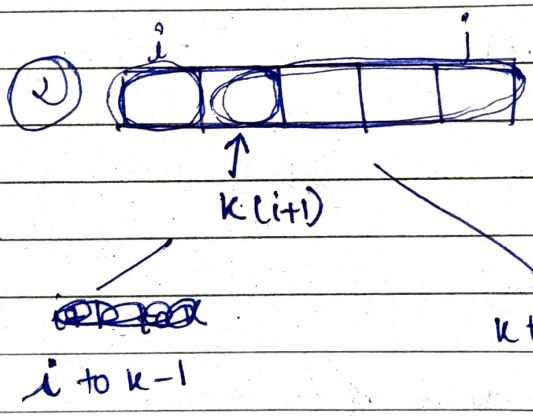
k=i k=j-1

i to k k+1 to j





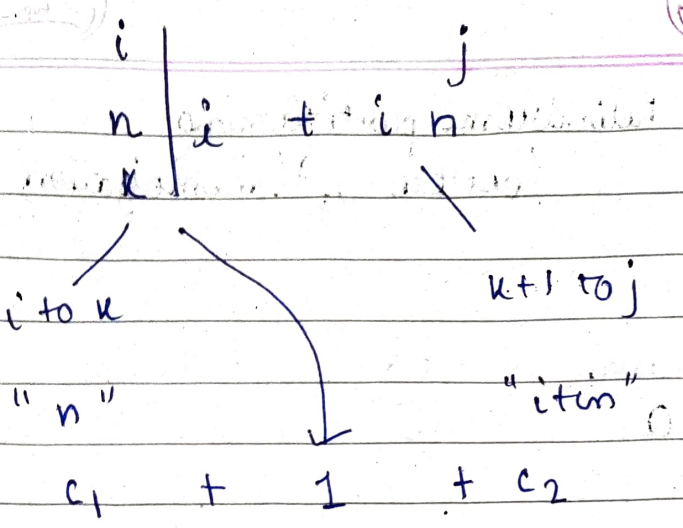




k = i+1 k=j

1)	k=i	k=j-1	i to k	k+1 to j
2)	k=i+1	k=j	i to k-1	k to j

```
for (int k= i ; k=j-1 ; k++)
    {
int temp = solve( s, i, k) + Solve( s, k+1, j)
    +1
```




```
    ans = min(ans, temp)
}
return ans;
}
```

Final code

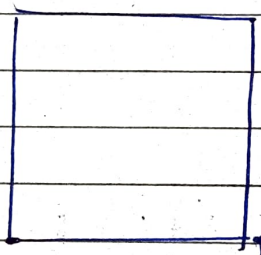
```
int solve(string s, int i, int j) {
    if (i >= j)
        return 0;

    if(isPalindrome(s, i, j) == true)
        return 0;
    int mn = INT_MAX;

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

        mn = min(mn, temp)
    }
    return mn;
}
```

memset < -1 / 0 > only allow 2 values

Palindrome partitioning (bottom up) (memoization)

i/p = nitin nitik
o/p = 0 2

1) Initialise the matrix with -1
2) Check if the value is
   - -1 (value isn't evaluated)
   - ≠ -1 value is evaluated so we store & return the value.



n x m
↓
variables which changes basically

i & j changes.

```
int dp [100][100];
memset (dp, -1, sizeof(dp));
int solve ( string s, int i, int j) {
    if ( i >= j)
        return 0;

    if (is palindrome (s, i, j))
        return 0;
```

```
    if (dp[i][j] != -1)
        return dp[i][j];
    int mn = INT_MAX;
for(int k = i ; k <= j-1 ; k++) {
    int temp = solve(s, i, k) + solve(s, k+1, j) + 1
    mn = min (mn, temp);
    }
    dp[i][j] = mn;
    return mn;
}

int palindrome part( string s) {
    int n = s.length() - 1;
    int ans = solve( s, 0, n);
    return ans;
}

bool ispalindrome (string s, int i, int j) {

    if (i == j)
        return True;

    if (i > j)
        return True;

    while (i < j)
    if (s[i] != s[j])
        return false
    else
        i++;
        j--;
```

Geeksforgeeks = ✓
Interviewbit = ✗

→ Further Optimization

Why the above code is not most optimized?

Since we are calling

int temp = 1 + solve(s, i, k) + solve(s, k+1, j)
RC (left) RL (right)

There is a possibility, might be one of the RC is solved or called.

The code will remain same but the diff^n is

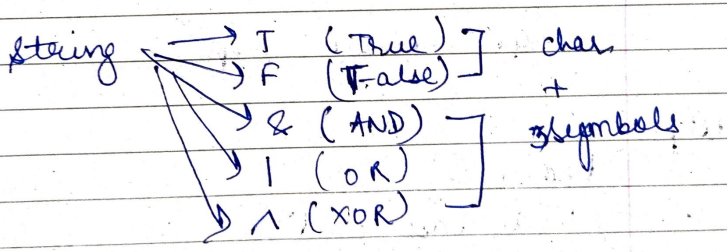
```
int temp = 1 + solve(s, i, k) + solve(s, k+1, j)

if (t[i][k] != -1)
    left = t[i][k]
else
    left = solve(s, i, k)
    t[i][k] = left

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

```
else
    right = solve(s, k+1, j)
    t[k+1][j] = right.

int temp = 1 + left + right;
```

Evaluate Expression to True
Boolean parenthesization

String : T or F and T
O/P : 2

Problem Statement : A string is given. String might have some characters like T → true F → false



How to put bracket such that the expression evaluates to true.

Find the no of ways in which when bracket is put it evaluates to True.

i/p : "T | F & T ^ F"

no of ways

( ) ( ) → T
( ) ( ) → F
( ) ( ) → T
( ) ( ) → T

} 3 ways.

In MCM we put brackets for min^m cost & in this also we do the same.

(T | F & T ^ F)

K-1 K K+1

We need to break bracket on operator

Expr^n operator Expr^n

4 steps:

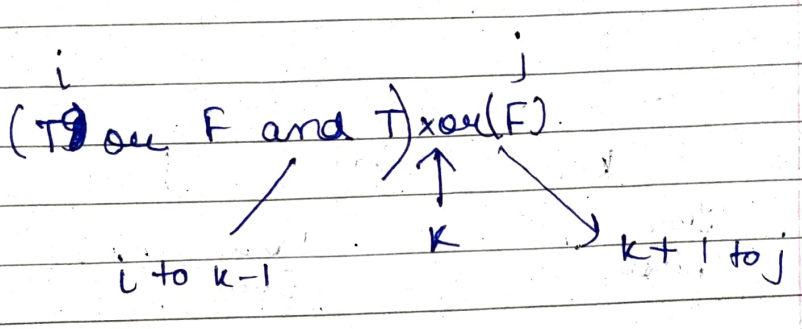
1) find i & j
2) find base cond^n
3) find k loop
4) temp ans + func^n

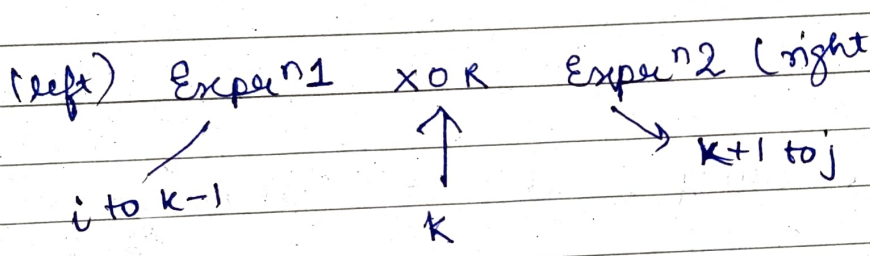
main ans.

```
      i       j
T | F & T ^ F
```

1) i = 0
   j = str.length() - 1

2) BC

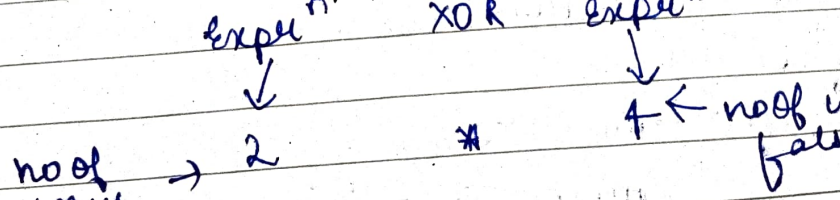


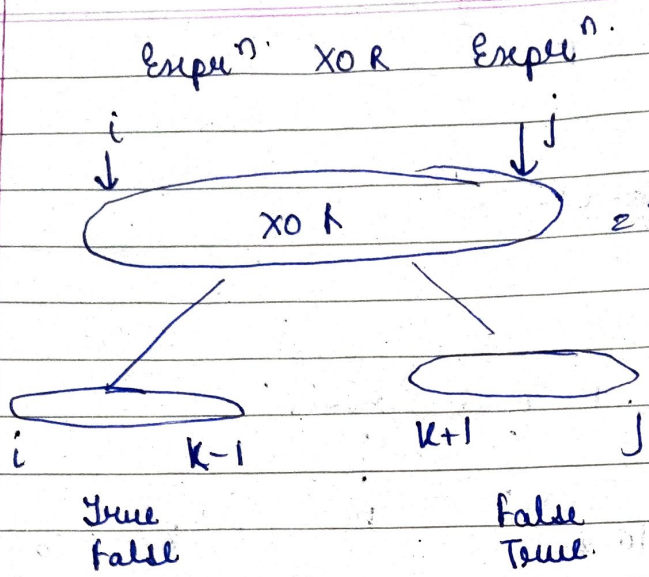


T ^ T = False
F ^ F = False
T ^ F = True
F ^ T = True

no of true ways = (no of T * no of F) + (no of F * no of T)

T ^ F = T
F ^ T = T





int solve (string S, int i, int j, boolean isTrue or isFalse)

Base cond^n

i j

T or F and T xor F

T | F & T ^ F

boolean isTrue

if (i > j) return false

if (i == j)

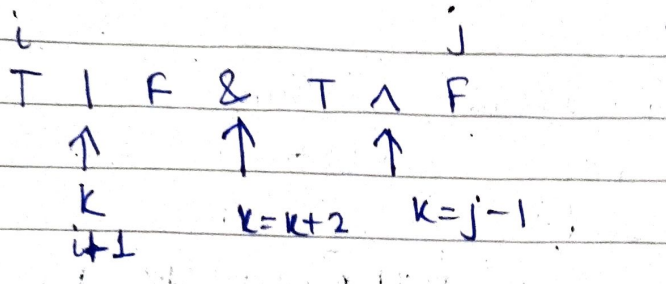
if (isTrue == Trues)

return S[i] == 'T'

else

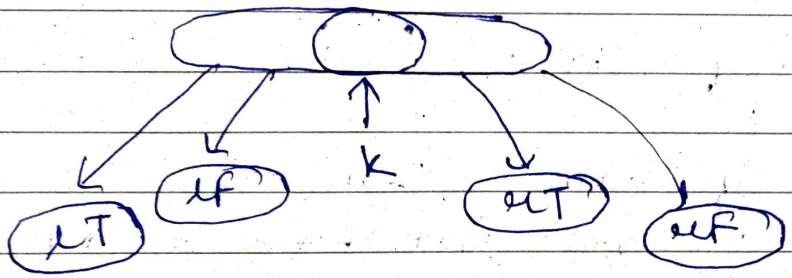
return S[i] == 'F'

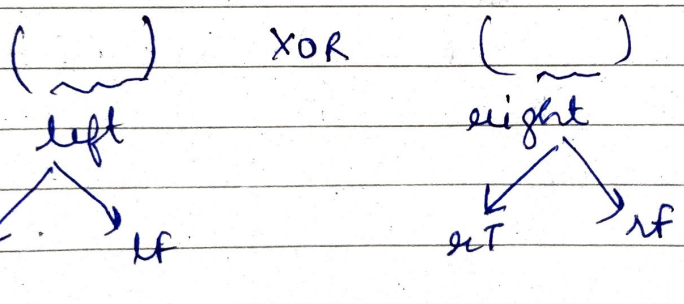
loop




```
for (int k = i+1; k <= j-1; k = k+2)
    int ans = 0;
    int lT = (s, i, k-1, T)
    int lf = (s, i, k-1, F)
    int rT = (s, k+1, j, T)
    int rf = (s, k+1, j, T)
```

} temp ans.





ans += lT * rf + lf * rT

```
if (s[k] == '&')
{
if (isTrue == True)
  ans = ans + lT * rT
else {
ans = ans + lf * rT + lT * rf + lf * rf;
}
else if (s[k] == '|') {
  if (isTrue == True)
    ans = ans + lT * rT + lT * rf +
                lf * rT.
  else
    ans = ans + lF * rF.
}

else if (s[k] == '^')
  {
  if (isTrue == True)
    ans = ans + lf * rT + lT * rf
  else
    ans = ans + lT * rT + lf * rf
  }
return ans;
}
```

Whole code

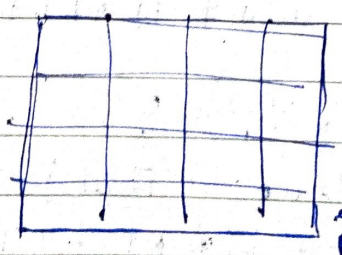
```
int solve (string s, int i, int j, bool isTrue ) {

    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 ans = 0;
    int LT = solve (s, i, k-1, T);
    int LF = solve (s, i, k-1, F);
    int RT = solve (s, k+1, j, T);
    int RF = solve (s, k+1, j, T);

    if (s[k] == '&') {
      if (isTrue == = True)
            ans = ans + LT * RT ;
      else
            ans = ans + LF * RT + LT * RF + LF * RF;
      }
    else if ( s[k] == '|') {
          if (isTrue == = True)
          ans += LT * RT + LT * RF + LF * RT;
        else
          ans += LF * RF;
      }
```

```
else if ( S[x] == '^' ) {
    if (isTrue == True)
      ans += LF * RT + LT * RF;
    else
      ans += LT * RT + LF * RF;
  }
}
return ans;
}
```

Evaluate Expression to True Boolean
Parenthesization - (memoization)
(Bottom Up - DP)
3D dp

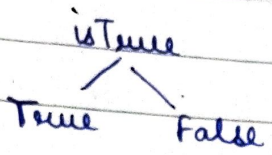
Recursive → Recursive call (RC)
Top down → table
Bottom up → RC + table

PS → put brackets in such a way that the expression evaluates to true.



i * j * isTrue
(3d)

int t [100][100][2]



More better way: We can use map

map	value
i, j, isTrue	
i+" "+j+" "+isTrue	

```
unordered_map < string, int > mp;

int main(){
   mp.clear()
   Solve( )
}
```

After Base cond^n :-

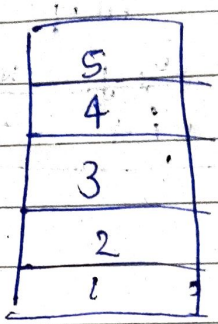
```
string temp = to_string(i);
   temp.push_back (' ');
temp.append ( to_string (j));
   temp.push_back (' ');
   temp.append ( to_string(isTrue));

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

```
    return mp[temp] = ans;
}
```

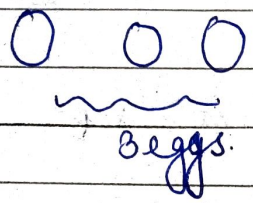
Egg Dropping Problem

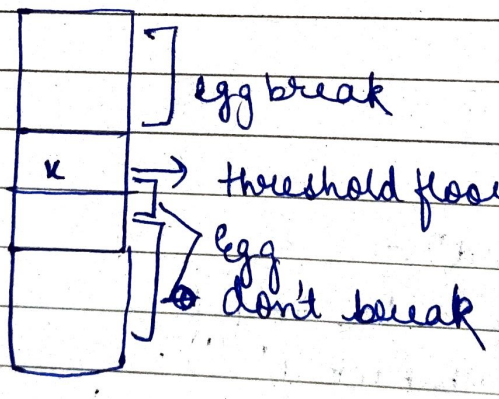
I/p : e = 3
      f = 5

O/p : 3 → minimize no of attempts in worst case.

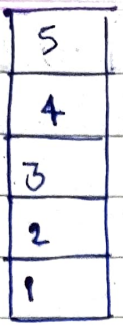


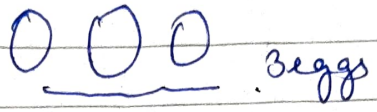
Problem statement :



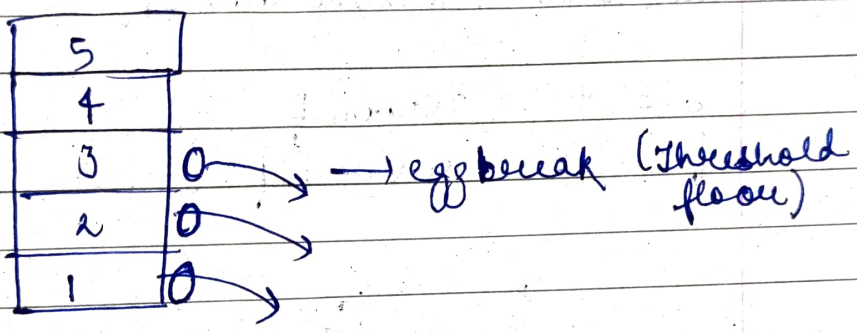


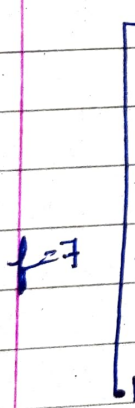
We are in a building, we need to find the minm no of attempts to find the critical floor.





Safe strategy → Worst case (1 egg) · so drop from the last it won't break until threshold.

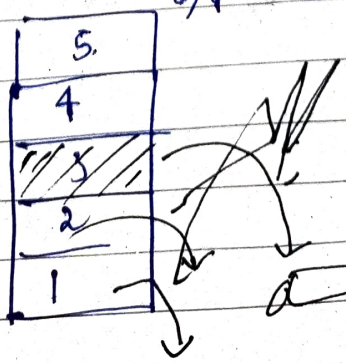




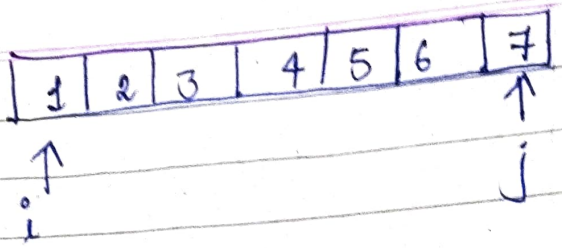
Worst-case → minimize no of attempts to find threshold floor.

I/p e=3
f=5

O/p = 3 attempts



(floor) f

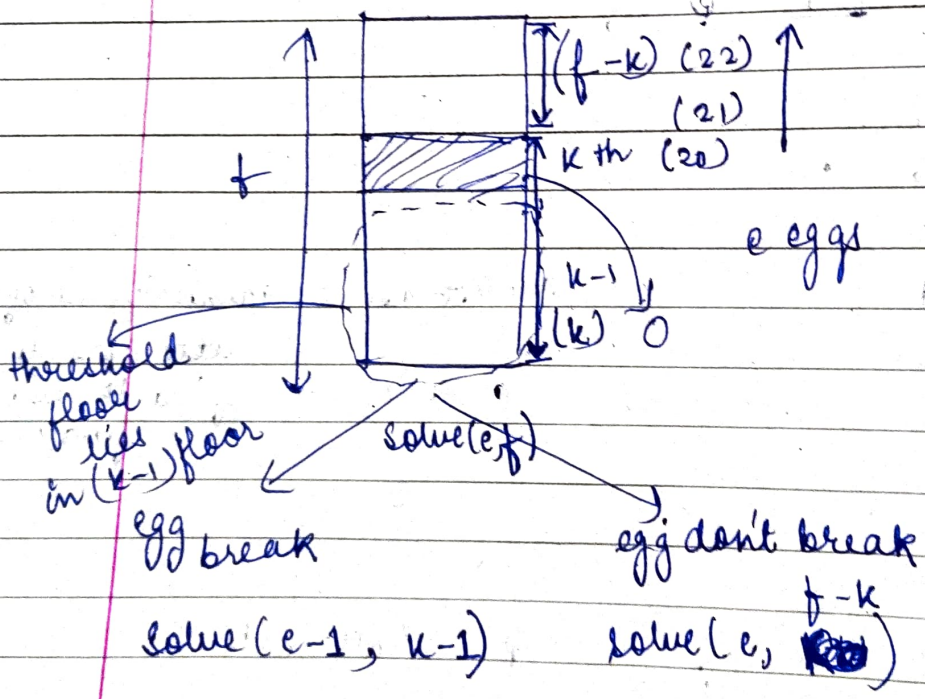


→ Building looks like an array

→ from where to take k?
We will check for all k values.
for k = 1 k = f k++

Base condⁿ. (smallest valid i/p)

e = 1 return f
f = 0/1 return f

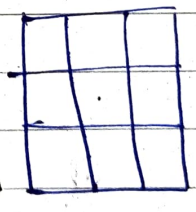


```
int solve (int e, int f)
{
    if (f == 0 || f == 1)
        return f;

    if (e == 1)
        return 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 mn;
}
```

→ for the worst case

Egg Dropping Memoization

$1 <= T <= 30$
$1 <= e <= 1000$
$1 <= f <= 1000$




```
// globally defined
int dp[1001][1001];
memset (dp, -1, sizeof(dp));

int solve ( int e, int f) {
    if (e == 1)
        return f;
    if (f == 0 || f == 1)
        return f;
```

```
if (t[e][f] != -1)
    return t[e][f];
int mn = INT_MAX;
for (int k = i ; k <= f ; k++) {

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

    mn = min(mn, temp);
}
return t[e][f] = mn;
}
```

Further Optimization

Inside the loop.

```
if (t[e-1][k-1] != -1)
    int low = t[e-1][k-1]
else
    low = solve(e-1, k-1)
    t[e-1][k-1] = low;

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

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