Page No.
Date

Dynamic Programming.

Identify → ① Choice.
→ ② Optimisation is asked.

Always Recursion lagega.
if Overlapping in Recursion → then DP ✓.
DP = Enhanced Recursion → (Recursion + Storage)
First try to solve problem by
① Recursion → ② Memoization → ③ Top down Approach.
↑ Must ↓ 1 or 2 lines for optimization. ↑ Dont do directly.

Parent Questions:

① 0-1 knapsack → 6
② Unbounded knapsack → 5
③ Fibonacci → 7
④ LCS → 15
⑤ LIS → 10
⑥ Kadane's Algo → 6
⑦ Matrix chain Multiplication → 7
⑧ DP on Trees. → 4
⑨ DP on Grid. → 14
⑩ Others. → 5.

I] 0-1 Knapsack Problem. → Parent Problem.

1. Subset Sum
2. Equal Sum Partition
3. Count of Subset Sum
4. Minimum Subset Sum
5. Target sum
6. No of subset ō given ∂/f

→ Minor Changes in Parent Problem

Knapsack Problem.

- Fractional Knapsack → Greedy.
- 0/1 → DP
- Unbounded → DP.

0-1 Knapsack:

wt[] = 1 3 4 5

val[] = 1 4 5 7

W = 7 kg. O/p: Max profit.

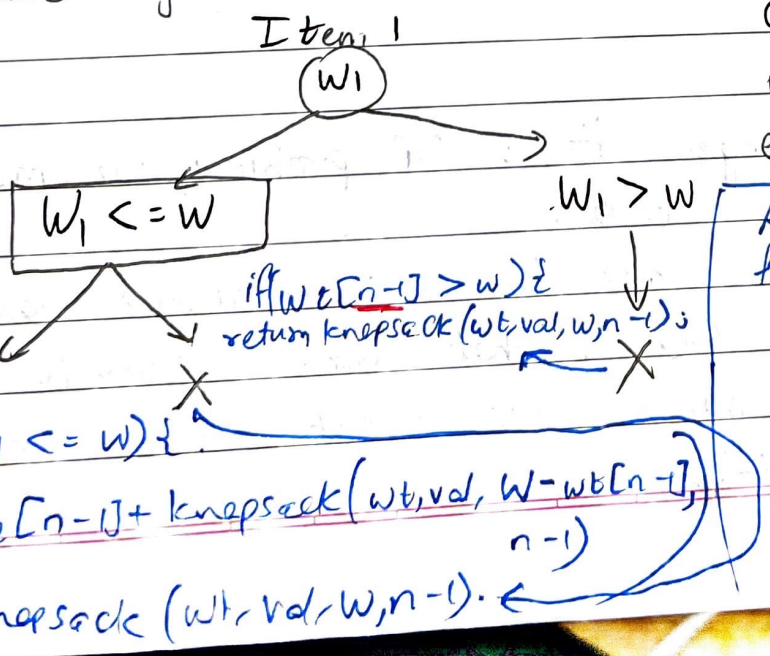
Identification: Max (optimisation).

For each wt we have choice → 0 → we dont include
→ 1 → we include.

∴ DP lagega.

↳ First Recursive solution.

Choice diagram:



Once Choice diag is made code is easy.

Always call Recursive func^n in smalla i/p

Page No.
Date

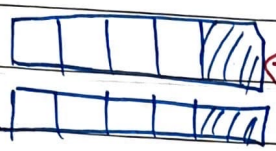
Recursive code:

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

① Base Condition → Think of smallest valid input.

② Choice diagram.

```
if(n==0 || w==0){
    return 0;
}
```
↑ Base Condition



we start from behind (n-1).

```
int knapsack (int wt[], int val[], int W, int n){
    if(n==0 || W==0){
        return 0;
    }
```
Base Cond^

as if we start from front we can't reach base condition

```
    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));
```
we start from behind

we select item

we dont select the item.

maximum of (we select or we dont select)

```
    else if (wt[n-1] > W){
        return knapsack(wt, val, W, n-1);
    }
}
```
Choice diagram

Now To change it to DP problem, we memoize the recursive code.

We store the ans to each recursive call in a matrix and if that same recursive call is called again we check if it is present in the matrix.

Memoization (Recursively)

How to know what to create in matrix.
Dimensions are # those who change in recursive calls.

knapsack(wt, val, W, n) → n-1
↳ can be W - wt[n-1]

Global declaration: int t[n+1][W+1]; as n & W changes in recursive calls.
memset(t, -1, sizeof(t)); → initilize all with -1 in start.

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

  if (n == 0 || W == 0){
    return 0;
  }

  if (t[n][W] != -1){ → Checking if some value is already present in matrix then return that; no need to find it again recursively.
    return t[n][W];
  }
  if (wt[n-1] <= W) → Storing in matrix each recursive call
  return t[n][W] = max(val[n-1] + knap(wt, val, W-wt[n-1], n-1)
                       , knap(wt, val, W, n-1));
  else if (wt[n-1] > W){
    return t[n][W] = knap(wt, val, W, n-1);
  }

}
```

TIME: $O(n^2)$

Top-Down Approach

↳ No recursive calls.
Get full from Table.
Why is this better? → If we solve recursively, there is chance that recursive stack gets filled completely then we can get error. (stack overflow).

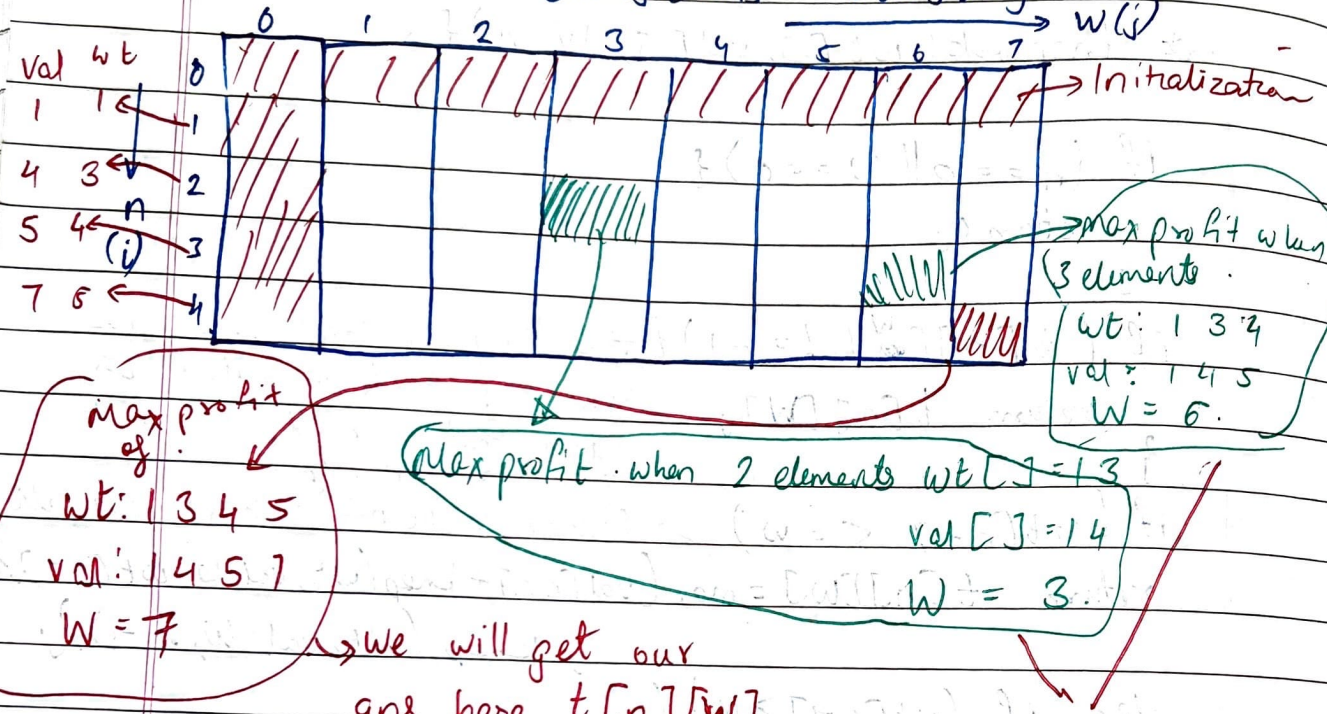
Time Complexity of TopDown & Recursive Memoized will be same.

Page No.
Date

Create Matrix for storing.
Step ① Initialize.
Step ② Store values (~~Recursively~~ → Iteratively).

wt: 1 3 4 5 ∴ n = 4.
val: 1 4 5 7
W = 7

Make Matrix t[n+1][W+1] = t[5][8]



These are subproblems

Changing ~~Recursive~~ Code ~~to~~ Top Down.

Recursive → Top Down
Base Condition → Initialization

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

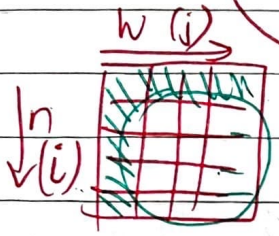
```
for(i=0; i<n; i++){
    for(j=0; j<W; j++){
        if(i==0||j==0)
            t[i][j]=0;
    }
}
```

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[n-1][W-wt[n-1]],
              t[n-1][W]);
}
else if(wt[n-1] > W){
t[n][W] = t[n-1][W];
```



Now change n & W to i & j

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

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

Page No.
Date

Child Problems to 0/1 Knapsack.

→ Identification
- Given array $I_1\ I_2\ I_3\ I_4$.
- Given max capacity or max value (W)

Every Item has choice lene that ki nai lene.

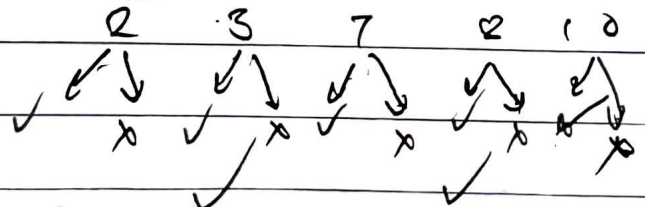
1. Subset Sum
2. Equal Sum Partition
3. Count of Subset Sum.
4. Min subset sum diff.
5. Target Sum
6. No of subset given s/t.

I] Subset Sum Problem → (Parent (0/1 → knapsack))

arr[] = 2 3 7 8 10.
Sum = 11
any subset present whose sum is 11?
↳ Yes or No me ans.
Yes; {3, 8} = 3+8 = (11) = Yes ← Ans
if Sum = 14. → No ← Ans. ∴ return boolean.
∴ Return True/false in Ans.

→ Similarity with 0/1 knapsack (Identification).

array given: arr[]: 2 3 7 8 10.
Sum = 11
You have choice for each item to take it or not in subset



{3, 8} ⇒ True.

0/1 knapsack → Subset Sum.

wt[] → arr[]

W → sum.

Page No.
Date

Make Matrix

t[n+1][W+1] → t[n+1][sum+1]

arr[] = 2 3 7 & 10.

sum = 11

∴ t[6][12].

arr size = 0 ie arr[] = { } sum = 1 ∴ not possible

for all ● arr size = 0

Item (j) →

Initialization.

	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											

Arr size (i)

your arr size = 3 ie arr[] = 2 3 7

Sum = 0

∴ { } ✓ → True ∴ all.

Sum 0 will be True.

Initilization Code:

```
for(i=0; i<n+1; i++)
  for(j=0; j<sum+1; j++){
    if( i==0 && j!=0)
      t[i][j] = false;
    else if (i!=0 && j==0){
      t[i][j] = true;
    else if (i==0 && j==0){
      t[i][j] = true;
    }
  }
}
```

Code: max → ||ly.

Knapsack

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

II] Equal sum Partition → (Parent → 0/1 knapsack)

arr: {1 5 11 5}.

o/p: T/F.

{ } { }

divide in 2 subsets such that sum of both are equal.

{1 5 5}, {11}.

(11) = (11)

∴ True ← Ans ∴ return boolean.

You have to divide sum into 2 parts.

{ } → (25) → Even

{ } { } { }

S = S

1 5 11 5 → (22) → have to be even to divide equally.

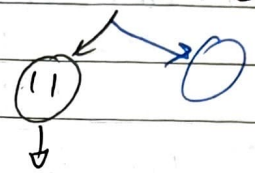
(11) (11)

If sum of arr was odd (23) You cannot divide equally.

So first calc sum, if its false then you cannot divide equally ∴ return false.

If sum is even, you need to find one subset whose sum = sum/2.

1 5 11 5



if a subset sum = 11 then other's sum will be 11 only.

```
int sum = 0.
for (i=0; i<size; i++){
 sum += arr[i];
}
if (sum % 2 != 0)  ← Sum is odd
  return False;
else {  ← Sum is even
  return SubsetSum(arr, sum/2, n);
}
```

if Subset sum is equal to sum/2, we can divide it into 2 subsets of equal parts.

```
boolean SubsetSum (int arr[], sum, int n){
{
  int [n+1] [sum+1];
  for (i=0; i<n+1; i++){
   for(j=0; j<sum+1; j++){
      if (i==0 && j!=0){
         t[i][j] = False;
  else if (i!=0 && j==0){
         t[i][j] = True;
  else if (i==0 && j==0){
         t[i][j] = no;
      }
} }
 for(i=1; i<n+1; i++){
   for(j=1; j<sum+1; j++){
    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];
  }
 }
}
```

Subset sum code. (full same).

III] Count of Subset Sum → (Parent → 0/1 knapsack)

Input: arr[] 2 3 5 6 8 10.

Sum = 10.

PS → Count number of Subset whose sum = = 10.

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

(3) ← Ans. ∴ return Integer.

Similar to Subset sum.

Matrix make. arr: 2 3 5 6 8 10

t[n+1][sum+1] Sum = 10.

∴ int t[7][11].

[INITIALIZATION]

Size of arr [i]	Sum → [j] 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										
2	1										
3	1										
4	1										
5	1										
6.	1										

Here we store count i.e no of subsets equal to given sum in the col w.r.t size of array of given row.

we will get ans here.

* Sum = 0 subset = { } → 1 subset possible.

In subset sum we did || es if we got any true then we got that subset is possible. Here we add as we add all the subsets that are possible to sum. * we dont stop on finding 1 subset; we continue to find all ahead.

Code:

Initialization

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

return t[n][sum]

{2,3,5} got 1 but we continue ahead to find total count

{2,3,5,6} {2,3,5} {2,3,6} {2} ...

{ }

{2} { }

{2,3} {2} {5} { }

{2,3} {2,5} {2} {3,5} {3} {5} { }

Page No.
Date

Minimum Subset sum Difference → (Parent → 0/1 Knapsack)

Input: arr[] 1 6 11 5.

You have to return minimum diff of sum of subsets.

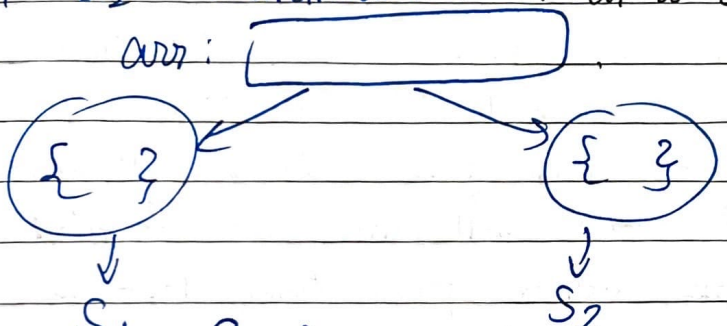
1 6 11 5

{1+6+5} → $S_1 = 12$ { 11 } → $S_2 = 11$

$S_1 - S_2$ = minimum = 12 − 11 = 1 → Minimum.

In Equal Sum Partition; You needed to check $S_1 - S_2 = 0$ or not

here $S_1 - S_2$ = minimum → Similar to Equal Sum Partition.



Eg: arr[] = 1 2 7. → Max Sum = 10. = Range

Find range of Subset sum can be

∴ range will be 0 to Sum of array

of S_1 & S_2

$S_1 = 1$ | $S_2 = 10 - 2 = 8$

$S_1 = 2$ | $S_2 = 10 - 4 = 6$

$S_1 = 3$ | $S_2 = 10 - 6 = 4$

4 − 3 = 1 ← MIN

sum of 1, 2, 7 cannot be 4/5/6.

0 ——— 10 Range = 10.

0 1 2 3 4 5 6 7 8 9 10

{ } {1} {2} {1,2} {7} {1,7} {2,7} {1,2,7}.

∴ S_1 / S_2 = {0, 1, 2, 3 | 7, 8, 9, 10}

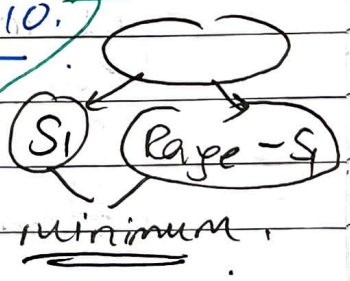
So if $S_1 = S_1$ → S_1; S_2 = Range − S_1 → Range − S_1

S_2 = Range − S_1. (if $S_1 = 2$; $S_2 = 8 (10-2)$)

You needed to ~~find~~ min ($S_1 - S_2$) or min ($S_2 - S_1$)

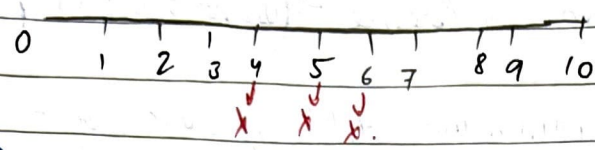
∴ now you need to find min (Range − S_1 − S_1)

min (Range − 2S_1).



Where Range = Sum of array.

$(\Sigma arr[i])$



S_1 cannot become 4, 5, 6.

You need to find this what S_1 cannot become.

How? → Subset Sum Problem.

Check S_1 till 5 only, as S_2 will be beyond 5 (5 to 10) (range − S_1)

arr: 1 2 7

sum (j) →

size of arr (i) ↓	0	1	2	3	4	5	6	7	8	9	10
0	T	F	F	F	F	F	F	F	F	F	F
1	T										
2	T										
3	T	T	T	T	F	F	F	T	T	T	T

Store in vector

True as when arr: 1 2 7 it can form subset whose sum = 1 i.e {1} → sum = 1 ∴ T

False as when arr: 1 2 7 cannot form subset of sum 5 ∴ False.

Arr: 1 2 7 subset = {2, 7} sum = 9 ∴ T.

∴ The last row will give which all sums will be formed & which will not be formed by subsets when arr size = 3.

S_1 will take values True from last row.

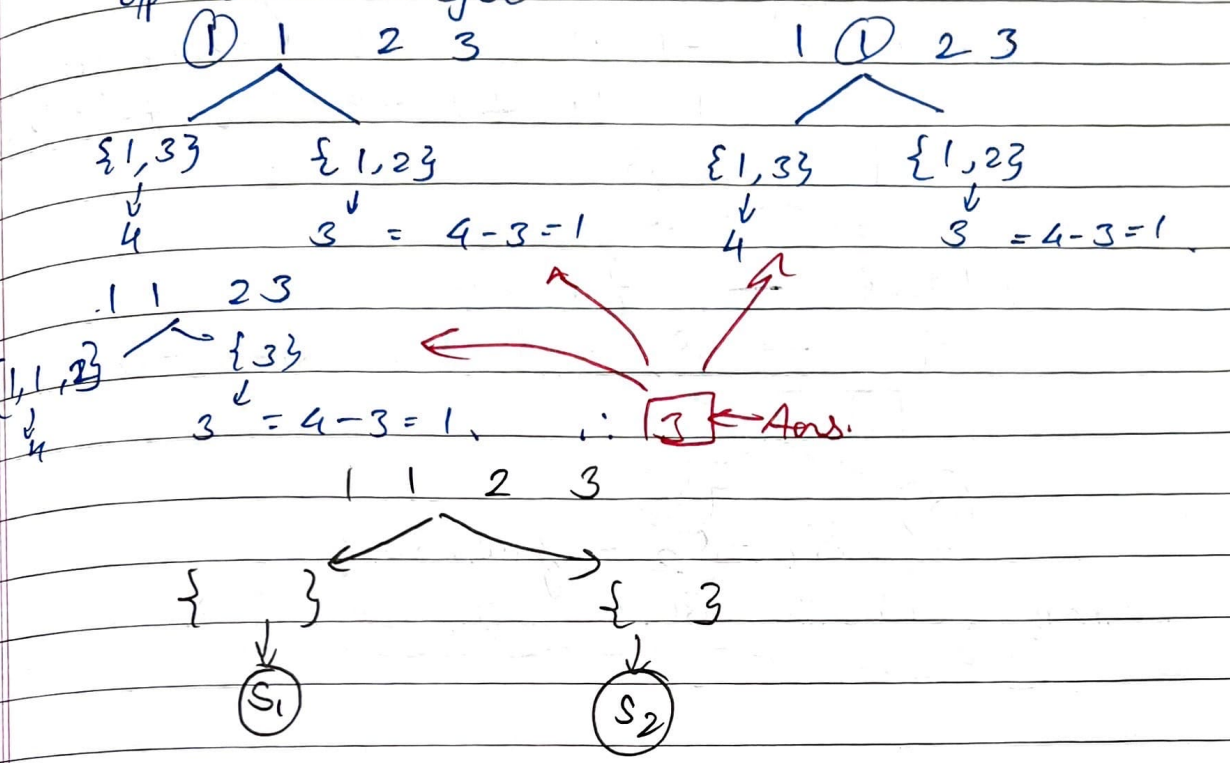
```
boolean SubsetSum (int arr[], int Range)
{
    Generate Matrix.

    Create vector<int> v        | 0 | 1 | 2 | 3 |
    int m = INT_MAX;            ↳ put the sum which are True.
    for(int i=0; i<v.size(); i++) till range/2 as after range/2 to
    { m = min(m, Range - 2v[i]) range; it will be S2 ∴ (Range - S1)
    }
    return m;
```

#1] Count number of Subset with given difference.

Input: arr[] = {1 2 3} (Parent - 0/1 knapsack).
diff = 1.

Find sum of 2 subsets and $S_1 - S_2 = diff$. count number of subsets possible.

o/p = 3 ← integer.



$S_1 - S_2 = diff.$

$+ \quad S_1 + S_2 = sum(arr)$

$2S_1 = diff + sum(arr)$

$$S_1 = \frac{diff + sum(arr)}{2}. \rightarrow \frac{1+7}{2} = 4$$

$\therefore S_1 = 4$

You need to find count.

: Find count of $S_1 = 4$. reduced to.

Count Subset Sum problem[Q3]

Subset Sum Problem

as when $S_1 = 4$; S_2 will automatically be 3 and diff will be 1.

```
int sum = (diff + sumofarr)/2;
return countofSubSetSum(arr, sum);
```

Page No.
Date

VI] Target Sum (Parent → 0/1 Knapsack)

Input: arr[] = 1 1 2 3

Sum = 1.

give + or − to each element such that you should get sum = 1 ; count no of ways.

Eg: 1 1 2 3 → +1 −1 −2 +3
Sum = 1 −1 +1 −2 +3
+1 +1 +2 −3
} → Sum = 1.
Count = 3. ← Ans.

1 1 2 3

{+1 +3} — {1, 2}

$S_1 - S_2 = 1$ ∴ $S_1 - S_2$ = diff

$S_1 - S_2$ = given Sum.

Same ↑ Subset Sum difference (Q5).
Count

rod cutting, coin change I, coin change II, Max ribbon cut

rod sums

Unbounded Knapsack (DP)

0/1 Knapsack → Included ✓ / not Included ✗

Unbounded Knapsack → Include 1 time ✓ → Include 2 times ✓ / Include 3 times ✓ / ✗ Not Include

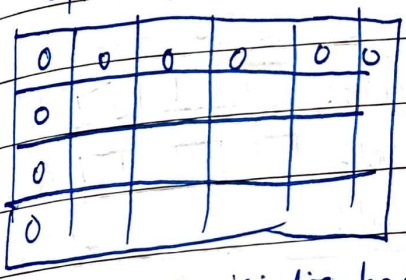
if item is not included it will never be included in future

If item is included it can be included as many times we want.

Multiple occurences allowed ✓

Code diff:

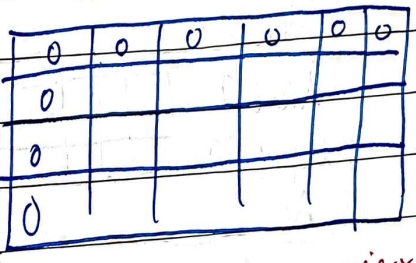
0/1 Knapsack



Same Initialization →

```
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



minor change

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

only this change will make unbounded

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

~~with~~ if item is included you can choose it again.

so you will again call on 'n' & not n-1, as you can include again if it is not included then you will call on n-1

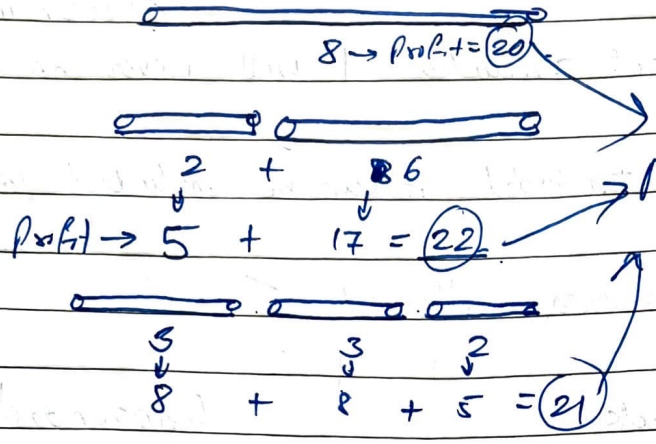
I] ~~Unbounded Knapsack~~ Rod Cutting Problem → (Parent → Unbounded knapsack)

Input: arr: len[] {2 3 4 5 6 7 8.

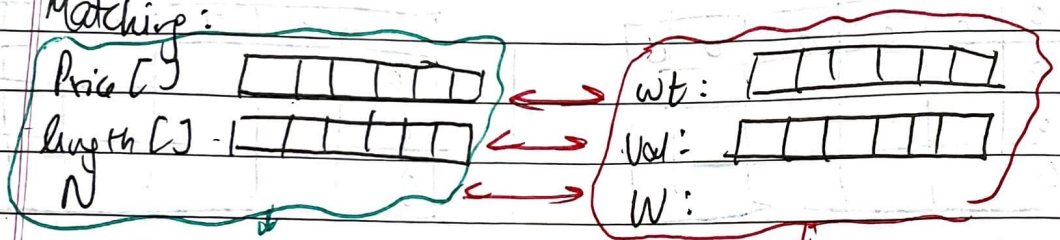
profit[] 1 5 8 9 10 17 17 20

N = 8.

You have to cut rod of length 8 into any number of parts such that profit is max



Matching:



Rod cutting

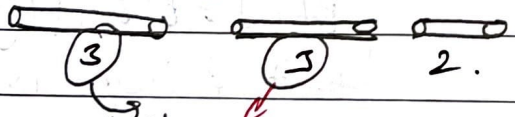
Unbounded knapsack

In both you have to maximize profit.

You have choice to cut in 2 peice or 3 peice etc

In knapsack you have choice to Include wt or not.

Unbounded knapsack as you can have Multiple occurrence of same length.



Multiple occurrences ∴ Unbounded knapsack.

∴ Variation of unbounded knapsack [Full same only framing different]

See code of Unbounded knapsack → t[n+1][W+1] ⟶ t[N+1][length+1]

mostly same.

if [1|2|3|4|5|6|7|8] ~~cancel~~ You have:

Flexibility to choose.

Page No.
Date

II] Coin Change I → Max number of ways.

I/p: coin[]: [1|2|3] → Unlimited supply.

Sum : 5

1 2 3 → 2+3 = 5

- → 1+2+2=5
- → 1+1+3=5
- → 1+1+1+1+1=5
- → 1+1+1+2=5

5 ways ← Ans.

How knapsack? (Unbounded knapsack)

↳ Choice in every coin to include or not.

↳ Multiple occurences of a coin ✓ → Unbounded knapsack.

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

Initialization:

size (i) \ Sum (j)	0	1	2	3	4	5
0	1	0	0	0	0	0
1	1					
2	1					

arr: [ ] = 0 size

Sum will be 0.

arr size = 2 i.e arr: [1, 2]

Sum = 0 → possible? → Yes → { }

Similar to Subset Sum count but it is unbounded

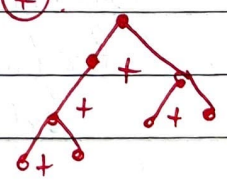
if (coin[i-1] <= j) → unbounded knapsack.

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

if in Q count or no of ways is asked always (+).

else

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



III] Coin Change II → Min number of coins.

I/p: coin[ ] = [1|2|3]

Sum = 5.

2+3 = 5 → 2 coins

1+2+2 = 5 → 3 coins

1+1+2 = 5 → 3

1+1+1+1+1 = 5 → 5

1+1+3 = 5 → 3

} Min no of coins such that Sum = 5 = [2] ← Ans.

Unbounded knapsack ✓

Initialization (Twist). sum → (j) arr: 3 2 0 5

Size of arr (n) (i)	0	1	2	3	4	5
0	INTMAX-1.	INT_MAX ⊗ -1	(INTMAX-1)	INTMAX-1	INTMAX-1	INTMAX-1
1	0			(1)	(INTMAX-1)	
2	0					
3	0					

arr size: 0 = [ ]

Sum = 2 ∴ Infinite : INTMAX-1

IN THIS SUM WE INITIALIZE 2nd ROW ALSO!

arr size = 3 arr size = 1 = [3]

Sum = 3 i.e {3} ✓ Sum = 4 There is no way sum will be 4

∴ (1) ∴ INTMAX - 1.

if j is NOT divisible by arr[0] then INT MAX-1.

INITIALIZATION:

```
for (int i = 0; i <= ~~size~~ +1; i++)
    for( j = 0; j < sum+1; j++)
        if( i == 0 && j != 0)
            t[i][j] = INT_MAX - 1;
        else if( i != 0 && j == 0)
            t[i][j] = 0;
        else t[i][j] = INTMAX-1;
}
}
}
```

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

Page No.
Date

Code:

```
for(i=2; i<n+1; i++)
  for(j=0; j<sum+1; j++)
    if(coin[i-1] <= j)
      t[i][j] = min( t[i][j-coin[i-1]] + 1, t[i-1][j] );
    else
      t[i][j] = t[i-1][j];
```

Coin included so +1

Coin not included

This is why we stored INT_MAX-1 as if that is case -1+1 & INT_MAX will remain.
OR INT cannot store INT_MAX+1 as value in it

Page No.
Date

#3 LONGEST COMMON SUBSEQUENCE.

↳ 14 child Problems.

I/P: X: a b c d g h
Y: a b e d f h r → abdh
↳ 4 length ← Ans.

① Recursive Soln

↳ Base Condtn → Think of smallest valid I/P.
↳ Choice Diagram.
↳ I/P ko small karna.

a b c d g h → ln = m = 6
a b e d f h r → ln = m = 7.

Smallest valid i/p = n = 0 || m = 0
↳ any one become 0 then LCS will be 0 only.

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

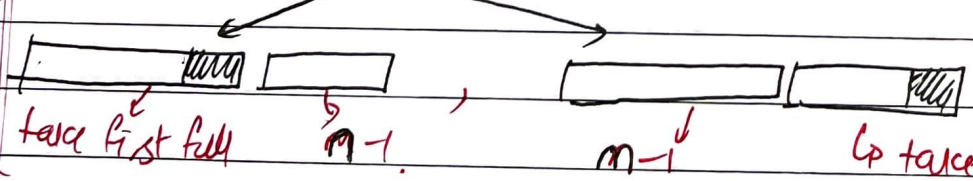
Choice Diagram.

Eg1: a b c d g h
a b e d g h → last matches. then recursive call on n-1 & m-1.

```
if( X[n-1] == Y[m-1])
    Rec call X → n-1
    Y → n-1.
```

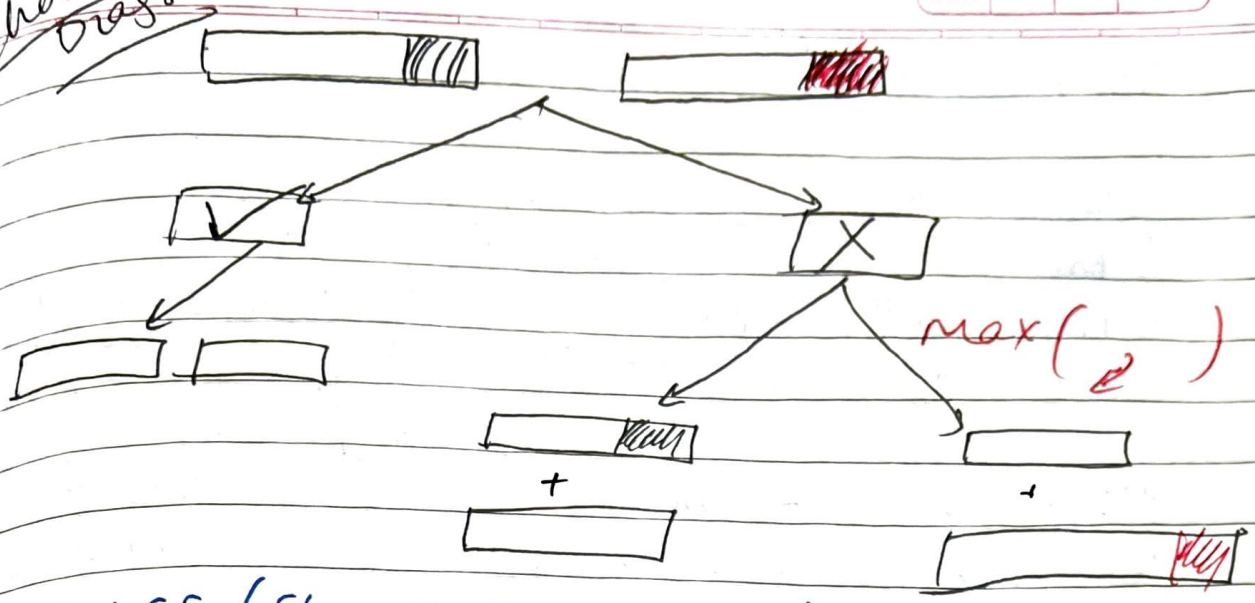
if not matching

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

max (

take first full, n-1, n-1, take second full)

Page No.
Date

Choice diagram:




```
int LCS (String X, string Y, m, n)
{
  if (n==0 || m==0)
      return 0;
  if (X[m-1] == Y[n-1]) {
      return 1 + LCS (X, Y, m-1, n-1);
  }
  else {
      return max ( LCS (X, Y, m-1, n), LCS (X, Y, m, n-1));
  }
```

Base condition

$O(2^n)$ Choice diagram

→ last index same then add 1 + rec call on small i/p on both strings.

↳ matched last index so increase count of LCS by 1

↳ longest chaise. ↳ no need to +1 as match not found.

② MEMOIZATION:

Make matrix t[ ][ ]

in rec call what changes? → m & n

if static t[n+1][m+1] ← declare globally (check constraints
∴ if $n, n \le 10^3$
make t[1001][1001].

initialize all with -1 ← memset (t, -1, sizeof(t));

```
  LCS(X, Y, m, n)
}
LCS( X, Y, m, n) {
  Base condn
  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-1, n),
                        LCS(X, Y, m, n-1).
```

$O(n^2)$

already present in table return no need to do recursion for that call again

How to identify if parent is LCS?

I/P	Q	O/P
a:	LCS	Int
b:		

Some Q:

if 2/3 match then parent is LCS

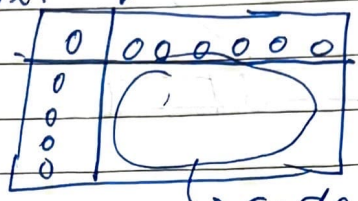
Page
Date

③ Top Down → LCS

Recursive ki base condⁿ

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

→ Top down me initilization.

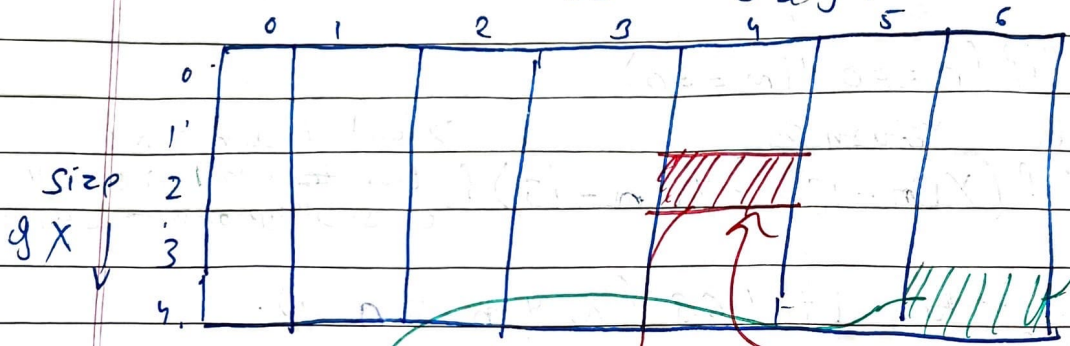


↳ code se bharega.

X: a b c f → n=4

Y: a b c d a f → n=6.

t[m+1][n+1] → t[5][7], size of Y →



X size=2 i.e a b } LCS ki length i.e 2. Store hogi.

Y size=4 i.e a b c d }

X size=4: a b c d }

Y size=6: a b c d a f }

↳ LCS length (Our ANS)

Recursive Choise Diya Code

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

else
return max(LCS(X,Y,m-1,n)
           LCS(X,Y,m,n-1))
```

→ Top down code

```
if (X[m-1]==Y[n-1]){
t[n][m] = 1+ t[m-1][n-1];

else
return max(t[m-1][n],
           t[m][n-1]);
```

Change m & n to i & j.

as this was for one block

```
for(i=1;i<m+1;i++)
  for(j=1;j<n+1;j++){
    t[i][j]=-1
    (replace m with i & n with j)
return t[m][n]
```

Page No.
Date

1] Longest Common Substring. (Parent → Longest Common Subseq wala)

Substring (VS) Subsequence

↓ ↳ can break in betn but order same.

↳ has to be continuous

I/P: a: a b c d e } In subseq ans is: a b c e i.e (4).
b: a b f c e.

↳ ab
↳ c } Longest = ab → length = 2 ← Ans.
↳ e

Similar to LCS (parent Q)

↳ Longest common asked

↳ I/P & O/P same.

Initialization:

Same as LCS →

	0	0	0	0	0
0					
0					
0					
0					

a b c d e

a b f c e → length now 1

↳ discontinuous ∴ length will be 0

∴ when discontinuous you need to make length = 0.

(if not match then)

LCS

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

LC Substring.

```
if (a[i-1] == b[j-1])
  t[i][j] = t[i-1][j-1];
else
  t[i][j] = 0;
```

↳ if not match make 0

Return max value in matrix & not t[m][n]. Traverse thrgh matrix once & return max as Substring can exist anywhere in between.

II] Printing LCS b/w 2 strings → [Parent → LCS].

Ip: a : a c b c f

b : a b c d a f.

o/p → a b c f (print) ← Ans.

t[6][7]

LCS table.

	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

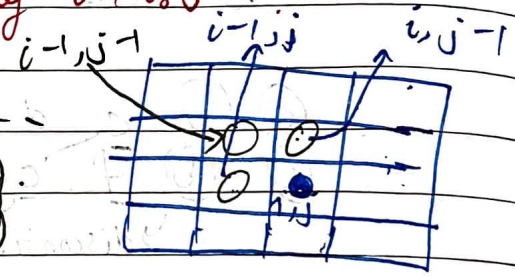
When f & f equal we print 'f c b a'

↳ reverse → a b c f

if not equal we choose max of i-1 or j-1 and go there.

if equal i,j → i--, j--

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



Code:

```
LCS(a, b, m, n) {
  table ✓
int i=m, j=n; String s=""; → starting from last block.
while(i>0 && j>0){
  if(a[i-1]==b[j-1]){ → if equal.
      s.push_back(a[i]); → push to ans.
      i--; j--; → move diagonal
  }
else{ → if not equal go to from where max came.
  if(t[i][j-1] > t[i-1][j]){
    } j--;
  else{
    i--;
}} reverse(s.begin(), s.end())
```

Page No
Date

Shortest Common SuperSequence → (Parent → LCS).

Ip: a: "geek"
b: "eke"

Merge a & b so you get both seq
geekeke. or geeke → Supersequence.
a b ↳ Shortest (≤ ∈ Ans).

a: A G G T A B
b: G X T X A Y B

(A)(G)(G)(T)(X)(A)(T)(A)(Y)(B).
(a) (b) its ok if not continuous.

a: (A)(G)(G) T A B.
b: (G)(X) T X A Y B.

A G G X T X A Y B. → Shortest
↳a ↳b Superseq (length = 9 ∈ Ans).

We are writing common elements one time.
A G (G) X (T) X (A) Y (B)
↳ Common in both strings. so we wrote only once

∴ Common → Write once.
a: A (G) G (T) (A) (B)
b: (G) X (T) X (A) Y (B)

G T A B → Longest common Subseq of a & b.

Worst superseq will be merge both strings
A G G T A B | G X T X A Y B

A G G T A B — (G T A B)
G T A B present in this
G T A B present in this also so we minus one G T A B
LCS of both

Shortest Common supersequence.
= length(a) + length(b) − LCS(a, b)

[IV] Minimum Number of Insertion & Deletion to convert str a to str b (Parent → LCS).

I/p: a: heap
b: pea.

o/p:

a → b

heap pea.

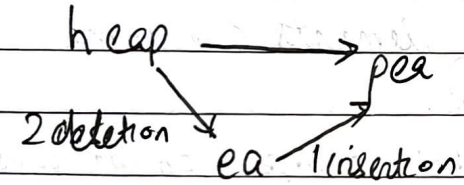
pheap = pea

| Insertion
| 2 deletion → Ans

	I/p	Q	O/p
How LCS Identify?	a:[], b[]	LCS	int
2 Strings given a & b	✓		int ✓
asked to find optimal. (∴ LCS)	2/3 Matching.		
o/p is int			

heap → pea.

ea: untouched (LCS).



First convert heap to ea
convert ea to pea.

no of deletion: length(a) − LCS.

no of insertion: ~~length~~ length(b) − LCS.

Page No.
Date

Longest Palindromic Subsequence (Parent → LCS)

I/P: s: a g b c b a. o/p: 5 ← Ans.

Find all ~~sub~~ palindromic subseq

↳ a b c b a → Longest length = 5 ← Ans.
↳ b c b
↳ b

How LCS?

Matching Algo.

	I/P	@	o/p
LCS	a:	LCS ✓	int ✓
	b:		
Our Q	a:	LPS ✓	int ✓

2/3 : ✓ LCS.

Here only 1 input string given but to apply LCS you need 2.

How you generate 2nd string.

You want palindromic subseq.

a g b c b a.

you compare from start and end

so you can take 2nd substring = reverse(a).

a: a g b c b a

b: a b c b g a.

Find LCS

↳ a b c b a → length = 5 ← Ans.

SUM UP Result.

↳ LPS(a) = LCS(a, reverse(a)).

Page No.
Date

V Minimum No of deletion in string to make palindrome.
I/p: a g b c b a. L(Parent → LCS)
O/p: 1

~~a~~ ~~g~~ b c b ~~a~~ → b c b (3 deletion)
~~a~~ ~~g~~ ~~b~~ ~~c~~ ~~b~~ ~~a~~ → c (5 deletion)
a ~~g~~ b c b a → a b c b a (1 deletion) ← 1 ← Ans.
longest substring palindrome

No of deletion ∝ LPS length.
∴ if you find LPS, you get min no of deletion

Min no of deletion = length(a) − LPS
↓
LCS(a, reverse(a))

~~Print~~.

VI Shortest common Supersequence → (Parent → LCS)
I/p: a: a c b c f
b: a b c d a f.
o/p: a c b c d a f.

Similar to Print LCS.

	∅	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

a b c d a f → same ∴ 1 time
a ~~b~~ b c f fca
not equal
include both

common → write once,
else → write rest of string.

Code:

LCS( ) ✓

table ✓

int i = m, j = n, string s = ""

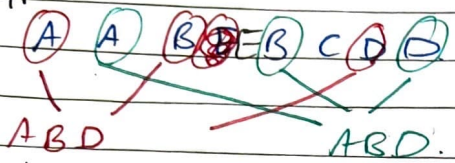
```
while(i > 0 && j > 0)
{
  if(a[i-1] == b[j-1]){
     s.pushback(a[i]);
     i--;
     j--;
  }
  else{
    if(t[i][j-1] > t[i-1][j]){
       s.pushback(b[j-1]);
       j--;
    }
    else if(t[i-1][j] > t[i][j]){
       s.pushback(a[i-1]);
       i--
    }
  }
while(i>0){
  s.pushback(a[i-1])
  i--;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

if i is not 0 & j becomes 0 you need to print a[i-1] bkz for j as we need superseq
Eg: a: ac
b: " "
here i = 2 & j = 0
so our ans will be ac

VII] Longest Repeating Subsequence → (Parent → LCS)

I/p: a = A A B E B C D D .

o/p = 3.



Longest Repeating subseq = B ← Ans.

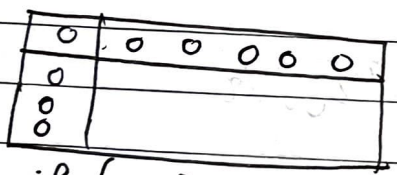
a = A A B E B C D D

b = A A B E B C D D

A → 0, 1	B → 2, 4	E → 3	C → 5	D → 6, 7
A → 0, 1	B → 2, 4	E → 3	C → 5	D → 6, 7

Code → LCS + Restriction → if i == j don't take in LCS

Initialization




```
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-1][j]
                   t[j][i][j-1] );
```

VIII] Seq

a:

b:

is

a:

b

Page No.
Date

Sequence Pattern Matching → (Parent → LCS)

a: A X Y
b: A D X C P Y.
is a subseq of str b.
a: A X Y
b: (A) D (X) C P (Y) → A X Y ⇒ True ∈ Ans (boolean).
Find LCS of a & b.
A X Y
A D X C P Y } → (A X Y) → a (if LCS is a then true)
→ length = 3.
if LCS length & length(a) is same then return true.
a is LCS means entire a is present in b.
Range of LCS can be 0 to min (m, n).
LCS();
if (LCS == a.length())
return true;
else return false.

Min number of insertion to make it palindrome.
Ip: s: a e b c b e d a.

a d e b c b e d a. → 2 insertion
x a d e b c b e d a x. → 4 insertion.
min
2 ← Ans.

No of Insertion = length(s) − LPS := no of deletion

No of deletion = No of Insertion
same code

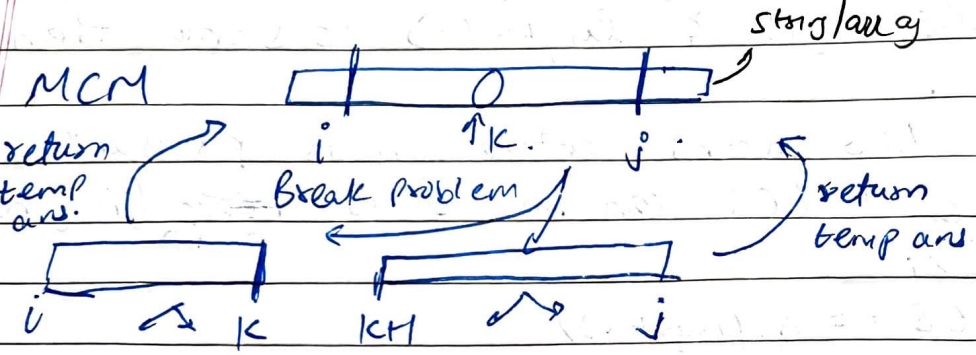
Page No.
Date

Matrix Chain Multiplication.

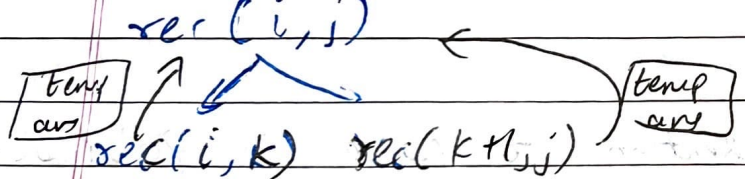
1) MCM.
2) Printing MCM
3) Evaluate Expr to true/boolean
4) min/Max value of Expression
5) Pallindrome Partitioning.
6) Scramble String.
7) Egg dropping Problems

Parenthesis Child Problems.

① Identification + Format.



make kahesd & again solve recursively.



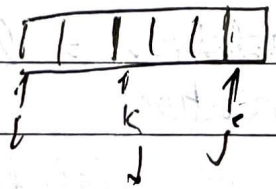
temp ans se ans nikalneka hai

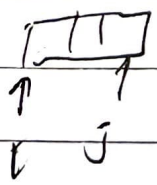
Format Code.

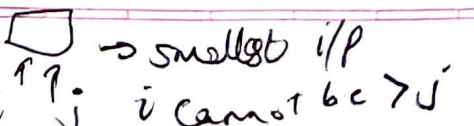
```
int solve (int arr[], int i, int j)
{
```

① Base Condition → Smallest valid i/p
↓ → Invalid I/p.

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



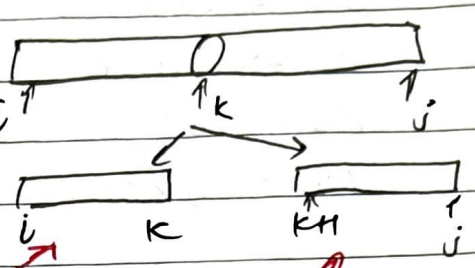




Steps ① Find i & j
② Base Condition
③ value of K loop scheme.
④ Find temp ans recursively ⑤ Calc ans from temp ans.

Format Code: for MCM sums.

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

if(i>=j){
    return 0;
}                                   } Base Condition

for(int k=i ; k<j ; k++){
    // Calc temp ans.
    tempans = solve(arr, i, k) (A1-) solve(arr, k+1, j)

    // Calc ans from tempans.
    ans = some function(temp ans).
}
return ans;
```



depend on Q.

↳ Eg: if you want to calc max then ans = max(ans, tempans)

I] Matrix Chain Multiplication.

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

Eg: A_1 A_2 A_3 A_4 → Find minimum cost.
2x5 5x3 3x2 2x4.
↳ Number of multiplication

How to find cost?

A_1 same always A_2.
2 x (3) (3) x 6.

Cost = 2 x 3 x 6 = 36 → No of multiplication

$(A_1 (A_2 \cdot A_3)) A_4$
$((A_1 A_2) (A_3 A_4))$
$(A_1 (A_2 (A_3 A_4))$
} which will give min cost is our ans.

arr: 40 20 30 10 30.

Matrix = 4 → (n-1).

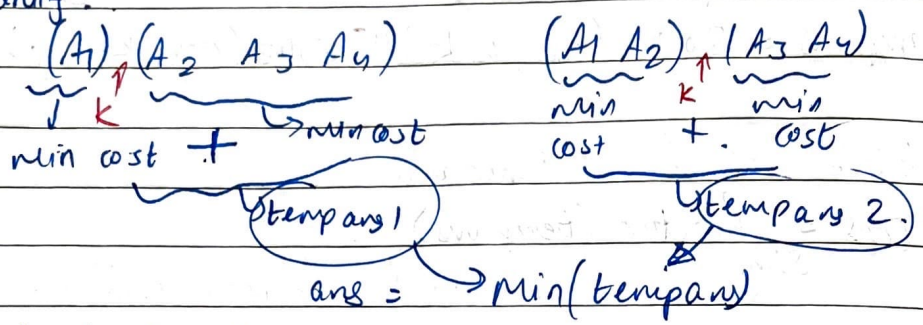
A1 → 40x20 ∴ Dimension

A2 → 20x30 ↳ $A_i \rightarrow arr[i-1] * arr[i]$

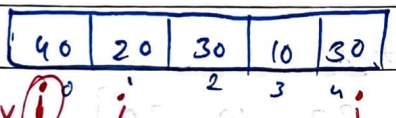
A3 → 30x10

A4 → 10x30.

Identify?



∴ MCM Format.



if i=0

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

$= arr(0-1) * arr(0)$

$= arr(-1) * arr(0)$ ← not Possible

if i=1 ∴ ~~Else~~ check if i from 1 is possible

$A_i = arr[0] * arr[i]$

$A_1 = 40 * 20$ ✓✓ right ∴ i start from 1

$\boxed{i = 1}$

For j

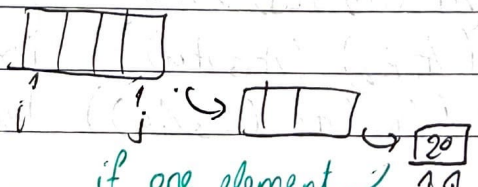
if j=4 $A_j = arr(3) * arr(4)$

$= 10 * 30$ ✓✓ right ∴ $\boxed{j = n-1}$

Base Condt^.

if (i <= j)

return 0.

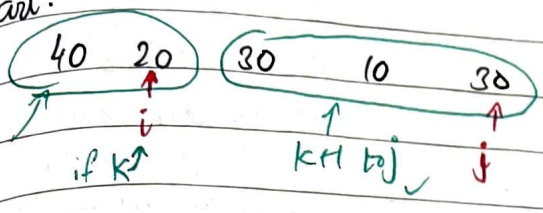


if one element present what will be dimension of matrix? Not possible so i=j not possible.

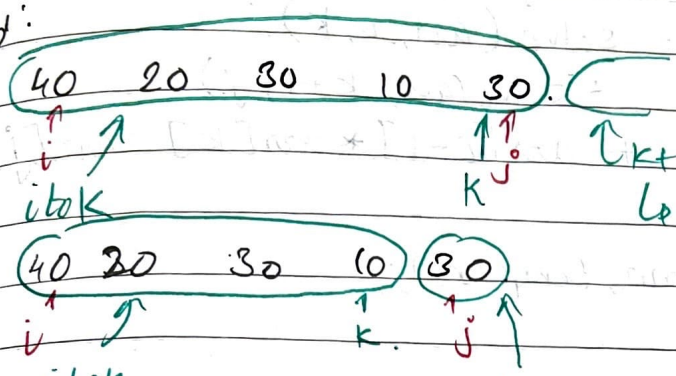
Page No.
Date

Now we have to decide values for k.

start:

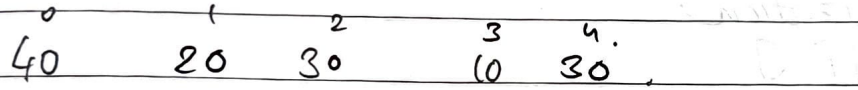


End:



for loop → From to Break

k = i	k = j-1	i to k & k+1 to j.
(or)		
k = i+1	k = j	i to k-1 & k to j.



from i to k

↳ k+1 to j.

40 * 20 20 * 30

30 * 10 10 * 30

C1 → cost = 40 * 20 * 30

cost = 30 * 10 * 30.

C3

solve (i to k)

solve (k+1 to j).

40 * 30 30 * 30.

C2

cost = 40 * 30 * 30.

arr[i-1] * arr[k] * arr[j] → Formula.

C1 + C2 + C3 = temp ans

(A · B) · (C · D)

↳ Result ka multiplication ka cost

Then find min of all temp ans.

Page No.
Date

Recursive Code:-

```
int solve (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];

    mn = min (mn, tempans);
  }
  return mn;
}
int main (){
  solve (arr, 1, n-1);
```

① Find i & j.
② Base Cond
③ K loop scheme
④ temp ans nikalo
⑤ ans nikalo from tempans

initial value of i & j.

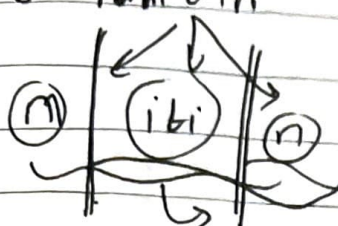
Memoization

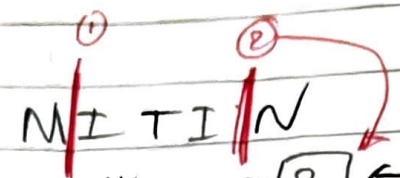
t[ ][ ]
what are changing? → i & j ∴ t[size+1][size+1]
i.e max val of i & j will be
size of matrix
Initilize entire with -1
memset (t, -1, sizeof(t)).
t[1001][1001]
memset (t, -1, sizeof(t))

```
int solve (arr, i, j){
  if (i >= j) return 0;
  if (t[i][j] != 0){
    return t[i][j];
  }
  for(          )
    :
    :
  return t[i][j] = mn;
}
```

II] Pallindrome Partitioning → (Parent → MCM) (Calc no of minimum partitions)

I/P: S: mitin





→ No of partition = 2 ← Ans

All pallindrome

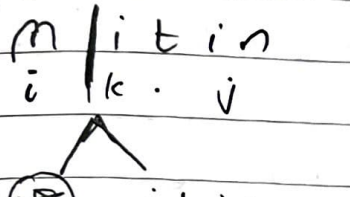
m|i|t|i|n → 4 partition

Worst Case → n−1 (5−1)=4

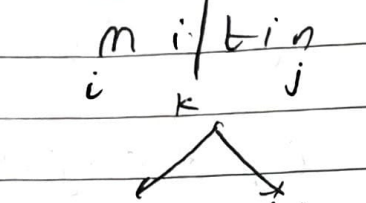
m|iti|n → 2 partition

minimize no of partition.

Identify? MCM Format.



temp ans. 1



temp ans 2.

Steps

① Find i & j value

② Base condition.

③ value of k for loop scheme.

④ temp ans calc.

① Value of i & j

M I T I N
i j ∴ i = 0 & j = size − 1

② Base Condition:

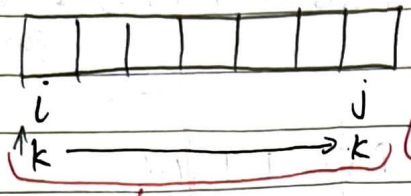
if i == j i.e arr size = 1

No partition reqd.
∴ return 0.
Same for i > j

```
if (i >= j)
    return 0.
if (ispallindrome(S, i, j) == true)
    return 0;
```

If string is already pallindrom no partition ∴ return 0

```
bool ispallindrome(S, i, j)
    if (i >= j) return true.
    while (i < j) {
        if [S[i] != S[j]) return false;
        i++; j--;
    }
    return true;
```



↳ k+1 to j ✗ Not possible.

$k = i$ to $k = j-1$ ∴ Break → i to k ∴ k+1 to j.

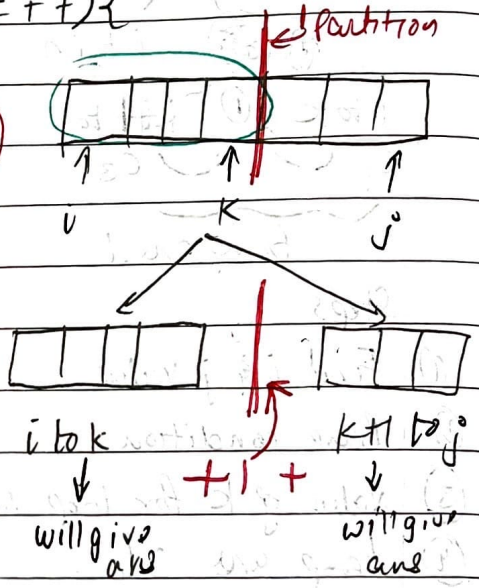
(or)

$k = i+1$ to $k = j$ ∴ Break → i to k-1 ∴ k to j.

∴ for(int k = i; k <= j-1; k++){

temp = solve(arr, i, k)
+ solve(arr, k+1, j) + 1

ans = min(ans, temp)
}
return ans.



② Memoization

Matrix t[ ][ ]

↳ what changes in rec calls → i ∴ j.

i ∴ j max value = string size. ∴ t[size+1][size+1]

initilize all with -1. if 1000 then t[1001][1001]

~~t~~ t[1001][1001].

memset(t, -1, sizeof(t))

Solve(arr, i, j)

Base Cond" ✓

if(t[i][j] == -1){

return t[i][j];

Code

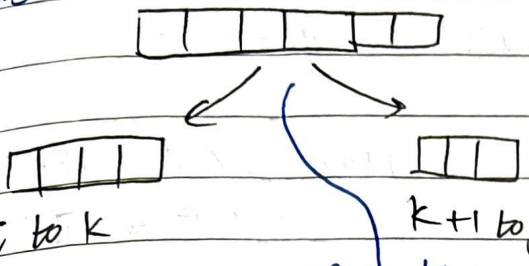
⋮

return t[i][j] = ans;

}

Page No.
Date

Optimised Memoization for Palindrome Partitioning



we used to check if this is already solved or not.
and we used to call rec call on i to k & k+1 to j
but what if those are already solved too?.
so store them & check ... instead of every time calling.

Replace

temp = 1 + solve (arr, i, k) + solve (arr, k+1, j);

↳ Why solve these everytime even if thy are already solved?

with. for( ){.

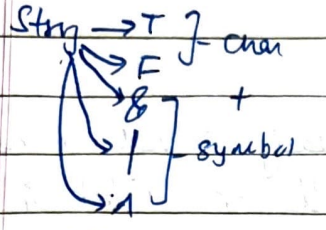
```
if (t[i][k] != -1)
int left = t[i][k];
else {
  left = solve (s, i, k)
  t[i][k] = left;
}
if (t[k+1][j] != -1)
int right = t[k+1][j];
else {
  right = solve (s, k+1, j)
  t[k+1][j] = right);
}
int temp = 1 + left + right.

  ans = min(ans, temp)
}
return t[i][j] = ans.
}
```

Page No.
Date

Evaluate Expression to true Boolean Parenthesization

I/p: string: T xor F and T o/p = 2.

T ∧ F & T.



Note: T ∧ T = F

T ∧ F = T

F ∧ T = T

F ∧ F = F

T ∧ F & T

(T ∧ F) & T T ∧ (F & T)

T & T = True T ∧ F = True

number of ways.? 2 ways ← Ans.

How MCM.?

(T | F) & (T ∧ F). (A1 A2 A3)(A4)

Bracket dekhne hai

(T) | (F & T ∧ F)

k k

k = k+2 hoga → (T | F) & (T ∧ F)

k

① Find i & j.

T | F & T ∧ F

i j

no prob here ∴ i = 0 j = size − 1

② Base Condition

Page No.
Date

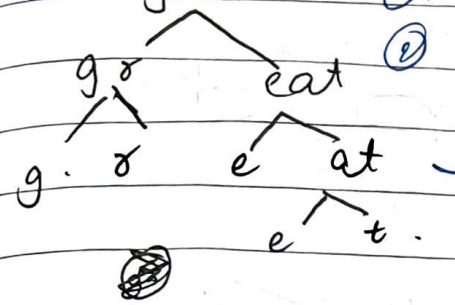
Scrambled String → Parent: (MCM)

I/p: a: "great"
b: "rgeat"

o/p → True → Ans (boolean)

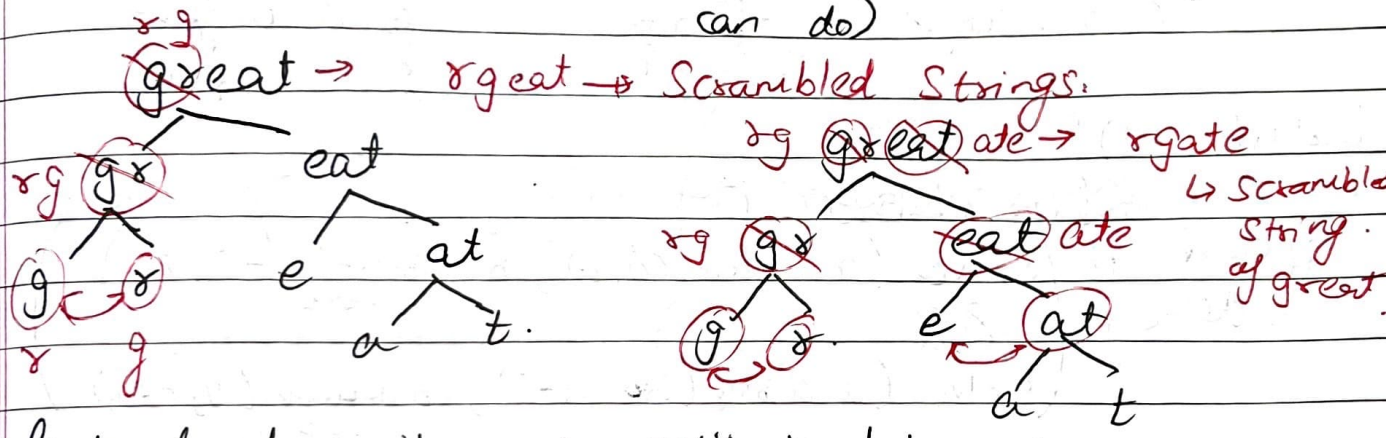
if Scrambled → True
↳ if not → or False

a → great → constraints: ① Make binary Tree (2 child)
② No child empty → great → X ("great")

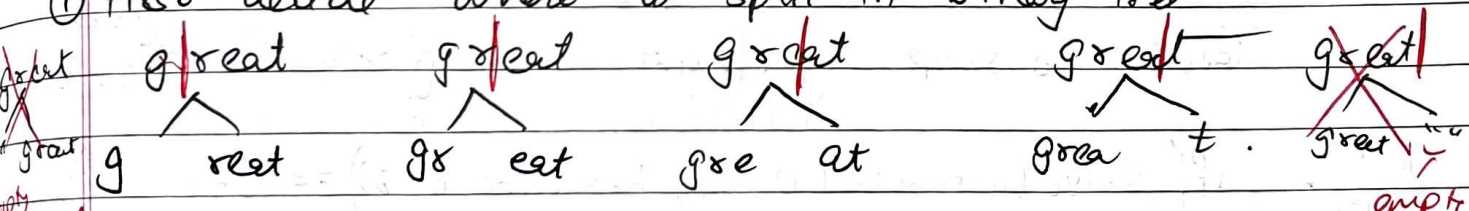


→ Can be more binary trees.

Non leaf node → We can swap. (Zero or More swapping we can do)



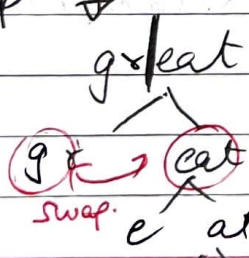
① first decide where to split in binary tree



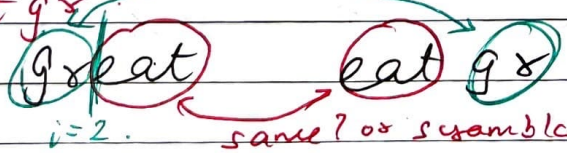
breaking on every k ∴ MCM.
∴ i = 1 to i = n-1

For each string → 2 opts → swap ✓ → will be same string
→ No swap X → will be scrambled string

if Swap →



∴ eat gr

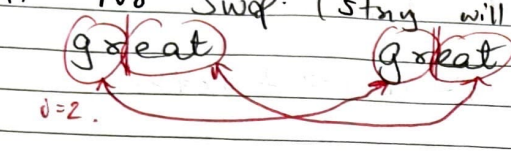


same?

i=2 ∴ first 2 of a & last 2 of b same? or Scrambled ↳ call recursively

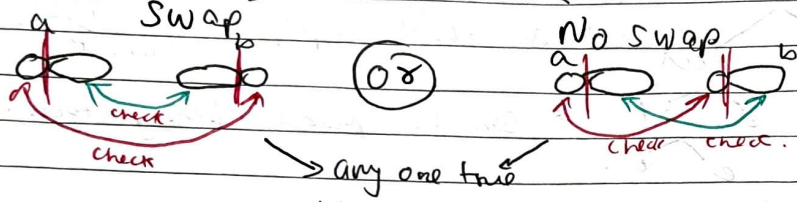
Page No.
Date

if No Swap. (string will stay same)



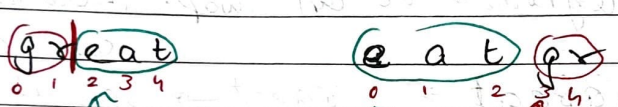
Scrambled string

for i = 1 to n-1



→ any one true then Scrambled string ✓ ∴ (or).

Case I : Swap ✓



eg: i = 2.

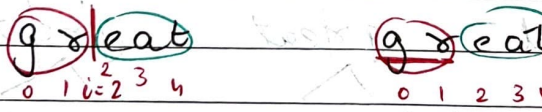
```
bool solve(a, b){
  if(a.substr(0, i)
```

start length

length.

if Swap is done → if (solve (a.substr(0, i), b.substr(n-i, i)) == true && solve (a.substr(i, n-i), b.substr(0, n-i)) == true.)

Case II : no swap



or grate.

if no Swap done → if (solve (a.substr(0, i), b.substr(0, i)) == true && solve (a.substr(i, n-i), b.substr(i, n-i)) == True)

(Case I || Case II) == true → Scrambled ✓

(or)

Page No.
Date

Base condition + Smallest input ?.

→ if len(a) != b(len) greed greetws. → No scramble possible

① if (a.length != b(length) return False.

② if (both empty) return true.

③ if (a == b) (or a.compare(b) == 0). return true

④ if any string is empty return False.
if (a.length <= 1 or b.length <= 0) return False.

```
bool solve (string a, string b){
  if (a.compare(b)==0)
    return True;
  if (a.length <= 1)
    return ~~true~~ False;
  int n = a.length.
  bool flag = False;
  for (i=1 to n-1; i++){
    if ( case① || case② ){
      flag = true;
      break.
    }.
  }
  return flag.
}
int main(){.
  if (a.length != b.length)
    return False.
  if (both Empty) → True;
  solve (a,b).
```

Memoization:

```
String key = a;
key.Push_back(" "); → a" "
key.append(b) → a" "b
if ( ~~key~~ mp.find(key) != mp.end()){
    return mp[key];
}
```

→ return mp[key] = flag.

unordered_map <String, ~~int~~ bool> mp;

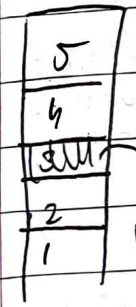
key : str = "a" + " " + "b"

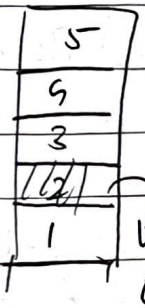
Variables that are changing i.e a & b

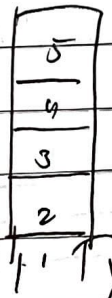
Egg Dropping Problem. → (Parent: MCM)

I/p: e = 3 (no of eggs) o/p = 3.
f = 5 (no of floor.)
↳ Minimize no of attempts in worst case.

egg break.
Threshhold floor / Critical floor. Minimum No of attempts needed to find critical floor
↳ Ques & we have for worst case. limited eggs.
→ no egs break.



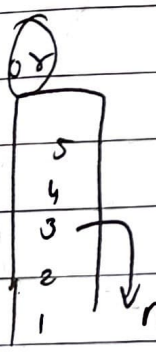


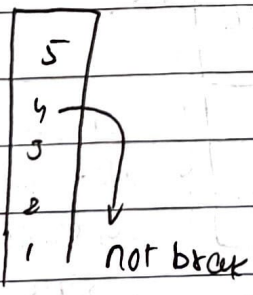


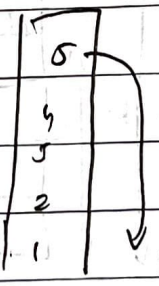
worst case. = 3 attempt.

0 0 ~~0~~

0 ~~0~~



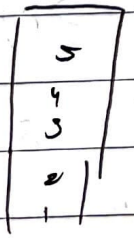




0 0 0

worst case: 3 attempts

Identify MCM?
Given: Floors = 5

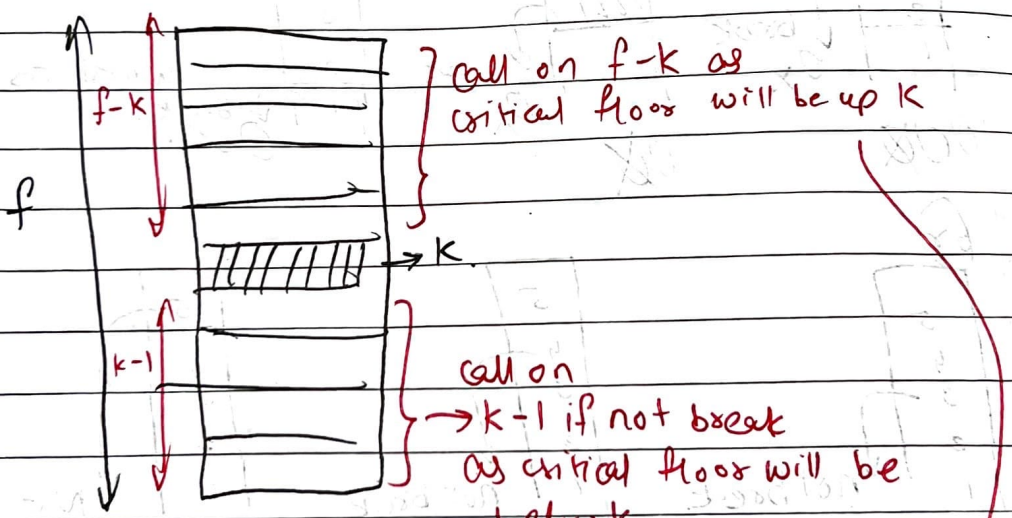


→ Similar to array ✓

5	← j ✓
4	k
3	k
2	k
1	← i ✓

i = 0 j = f.
k = 1 to j ✓, k++.

Base Condtn:
e = 0 / 1
f = 0 / 1
if (e = 1)
  return f; (you start from floor 1 to critical floor)
if (f == 1 or f == 0)
  return f; (if 1 floor [1] ✓ - only 1 attempt it will break or not)



Solve (e, f)

Break ↓ Solve (e-1, k-1)

Not break. Solve (e, f-k).

You need to find Min no of attempts in WORST Case ∴ Maximum of (Break, not break)

```
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;
}
```

Worst case

Min of worst case.

Memoization:

```
int t[e+1][f+1];
memset(t, -1, sizeof(t));
if (t[e][f] != -1) {
    return t[e][f];
return t[e][f] = mn;
```

Optimization in Memoization

We are calling recursive on e-1, k-1 & e, f-k every time so we can store this also in t[][] & check before calling if already present:

```
if (t[e-1][k-1] != -1) return t[e-1][k-1];
else {
    int low = solve(e-1, k-1);
    t[e-1][k-1] = low;
}
if (t[e][f-k] != -1) return t[e][k-1];
else {
    int high = solve(e, f-k)
    t[e][f-k] = high;
}
int temp = 1+max(low, high);
```

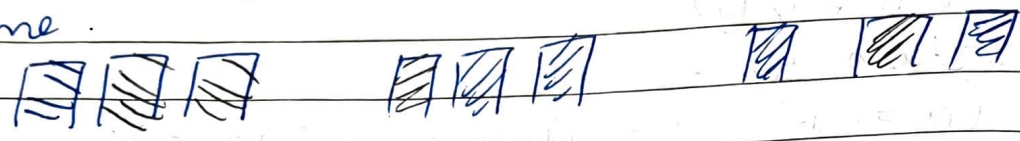
Optimized Memoization

Page No.
Date

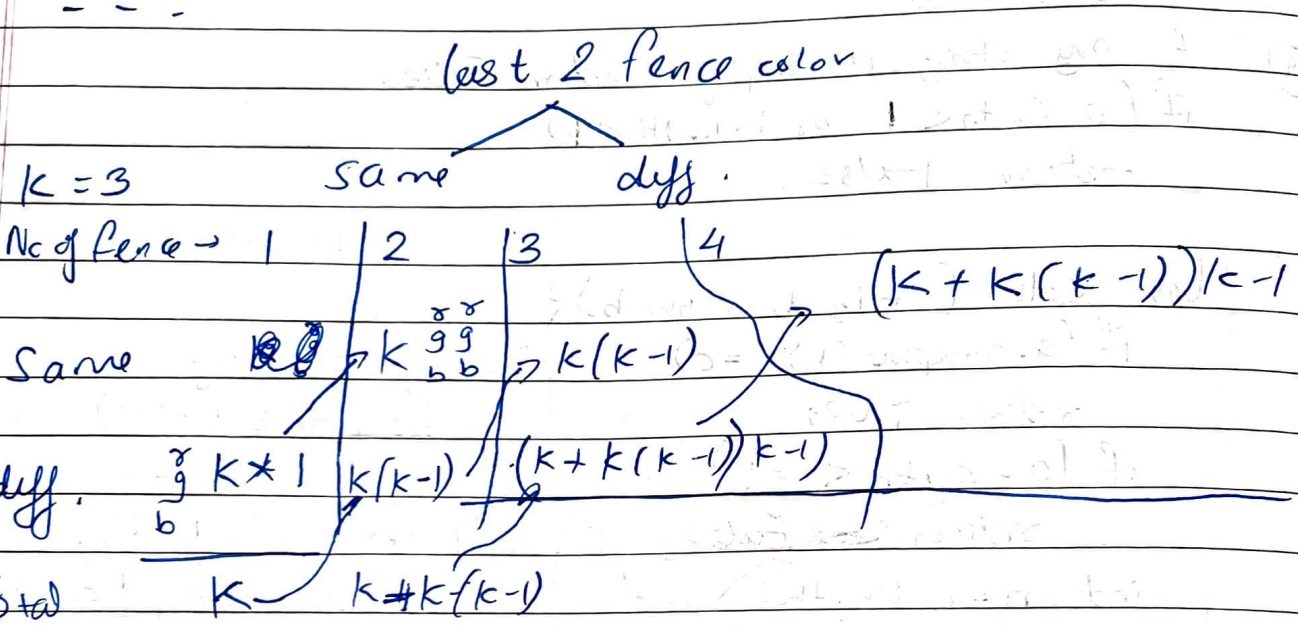
Painting Fence → (Parent: None)

n posts & k colours

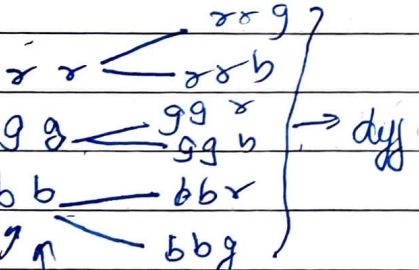
number of ways you can paint such that ~~atmost~~ adjacent fence have atmost 2 colors same.



- - -



When n = 2 & k = 3



each k-1 i.e k(k-1).

same = k ; diff = k(k-1)

∴ for (i = 3; i < n; i++)

same = diff

diff = (total) * k - 1

total = same + diff

}

return total.