Dynamic Programming

PAGE NO.

(Aditya Verma)

(Lecture 1) Introduction to DP

(1) DP is an advanced version of recursion.

DP = Recursion + storage

(2) Identification of DP =>

(1.) choice → recursion
(2.) Optimal.

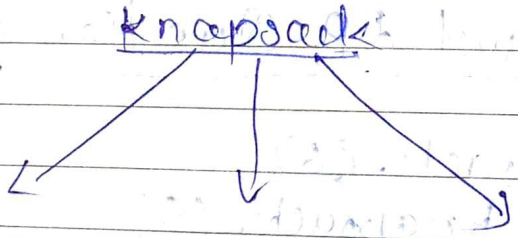
Standard DP Question

(1.) 0-1 knapsack. (6)
(2.) Unbounded knapsack. (5)
(3.) Fibonacci. (7)
(4.) LCS. (15)
(5.) LIS. (10)
(6.) Kadane's Algorithm. (6)
(7.) matrix chain multiplication. (8)
(8.) DP on Trees. (4)
(9.) DP on Grid. (14)
(10.) Others. (5)

(1.) 0-1 knapsack problem.

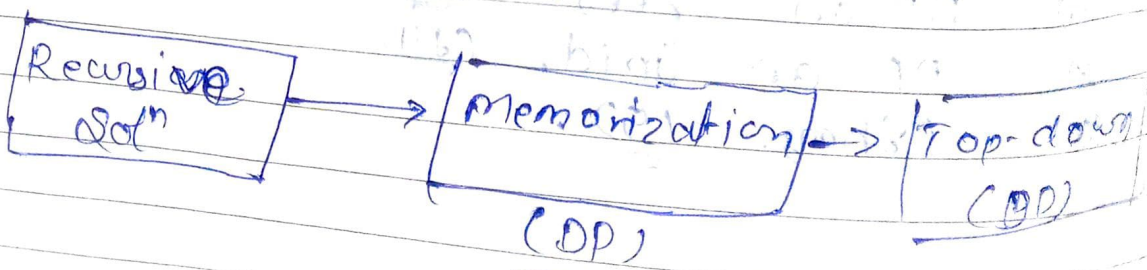
variation problems

(1.) Subset sum.
(2.) Equal sum partition.
(3.) Count of subset sum.
(4.) minimum subset sum diff.
(5.) Target sum.
(6.) # of subset & given d/f.



Method of soluing DP

D.P



0-1 Knapsack Recursive

DP → Recursive

↓

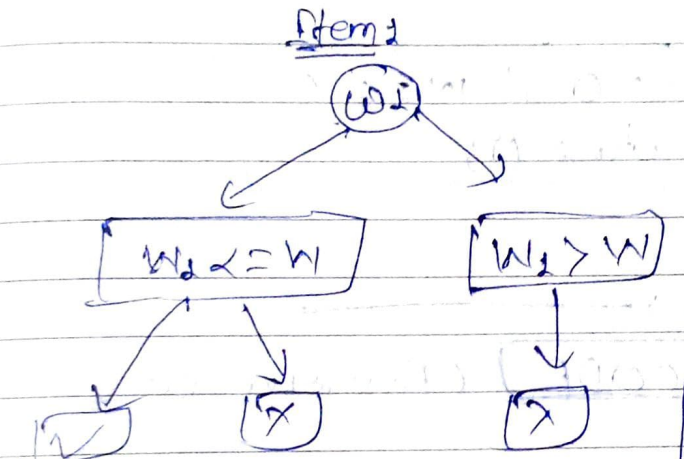
Choice

I/P ⇒ wt[]: [1 | 3 | 4 | 5]

val[]: [2 | 4 | 5 | 7]

W: 7

Choice Diagram



O/P ⇒ 9

Psudo code

I/P

```
int knapsack(int wt[], int val[], int W, int n)
{
    Base Condition
    Choice Diagram
}
```

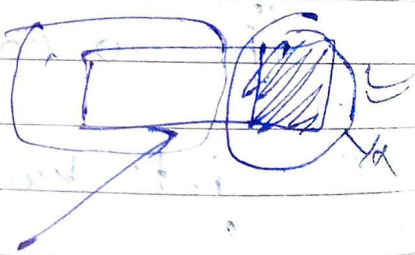
Base condition :)

Base Condition → think of the smallest valid I/P

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

CODE (Recursive code)

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

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



Knapsack memorization (DP)

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

int knapsack(int wt[], int val[], int W, int n)
{
    if (n==0 || w==0)
        return 0;

    if(dp[n][w] != -1)
        return dp[n][w];

    if (wt[n-1] <= w)
        return dp[n][w] = 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 dp[n][w] = knapsack(wt, val, w,
                                   n-1);
}
```

dp[][]

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

→ if it's valid then return it

→ else process it and save the value here.

Knapsack TopDown DP

How to create table()

I/p ⇒

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

n = 4

table[n+1][W+1] ⇒ table[5][8]

① Step 1 ⇒ Initializing 0th row and 0th column ~~as~~ with base condition.

(j) ← wt+1 →

(i) val+1

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

(2) step 2 :) converting choose into code;

```
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]
    }
}
```

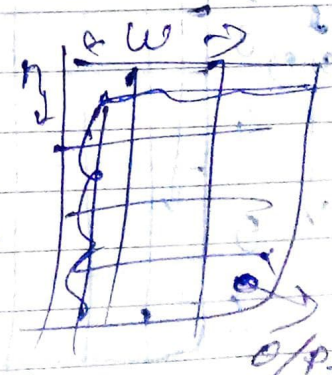
Sum up

(1) create t[n+1][w+1]

(2) Initialization with Base cond.

(3)

(4) output t[n][w]



Subset Sum Problem

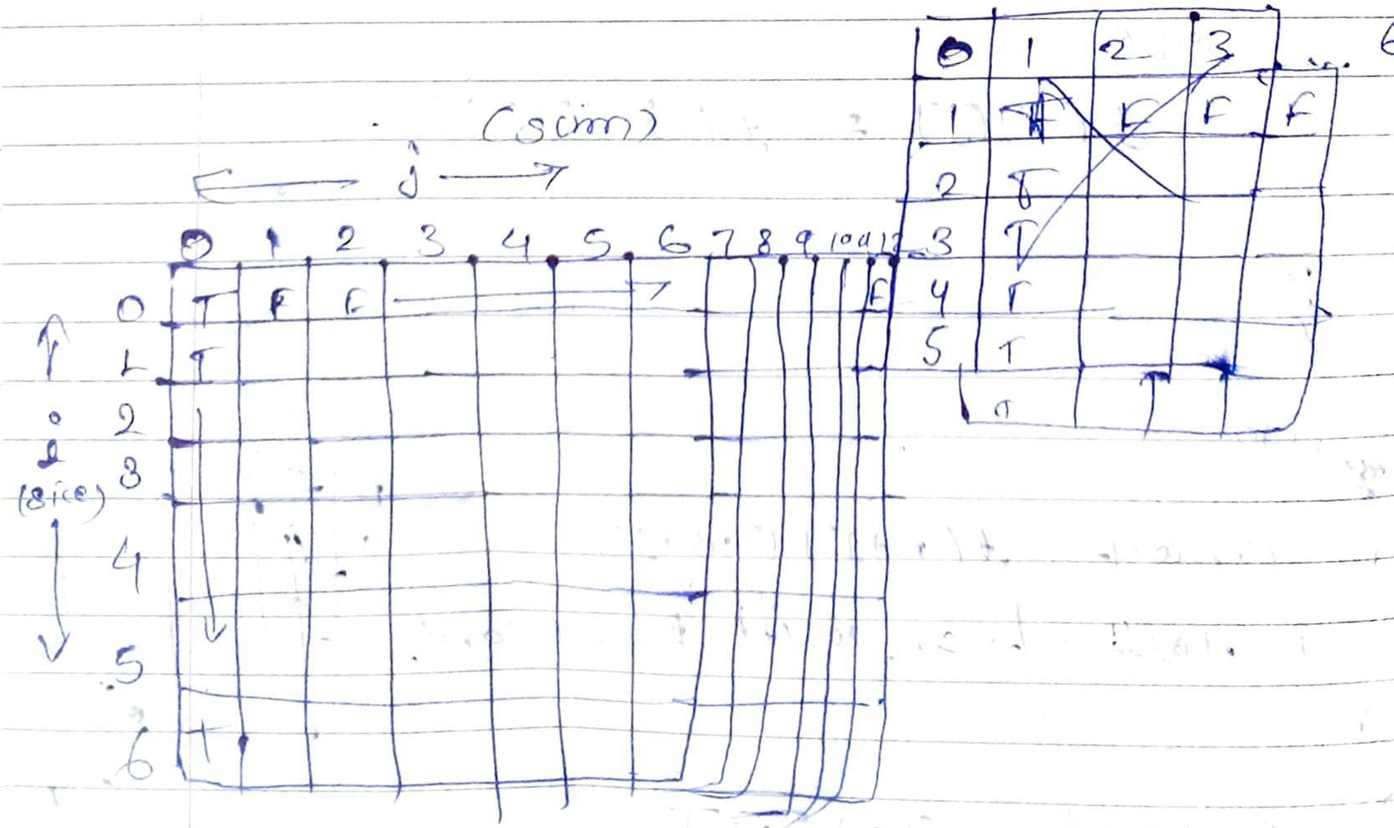
Given,

arr = [2, 3, 7, 8, 10]

sum = 11

↳ check if there is subset of sum equals to given sum.

① dp [arr.size()+1] [sum+1]



PAGE NO

('CODE)

```
bool isSubsetSum(int arr[], int n, int sum)
{
    bool dp[n+1][sum+1];

    for(int i=0; i<=n; i++)
        dp[i][0] = true;

    for(int i=1; i<=sum; i++)
        dp[0][i] = false;

    for(int i=1; i<=n; i++) {
        for(int j=1; j<=sum; j++)
    {
        if(j<arr[i-1])
            dp[i][j] = dp[i-1][j];

        if(j>=arr[i-1])
            dp[i][j] = dp[i-1][j] ||
                       dp[i-1][j-arr[i-1]];

    }}
    return dp[n][sum];
}
```

Equal Sum Partition

given,

arr = [. . .]

sum1 == sum2

CODE

```
int sum = 0;

for(int i = 0; i < arr.size(); i++)
    sum += arr[i];

if( sum % 2 != 0)
    return false;

else
    return subsetSum(arr, sum/2);
```

Lecture 9

Count Subset Problem

Given;

arr = | 2 | 3 | 5 | 6 | 8 | 10 |

sum = 10

Note :- This problem is similar to subset sum problem.

Step 1 :- dp[arr.size()+1][sum+1]

Initialization step is same instead of true, false assign 1, 0 respectively.

Step 2 :-

```
for i:1 ---->
    for j:1 ---->
{
  if( j >= arr[i-1])
  {
    dp[i][j] = dp[i-1][j] + dp[i-1][j-arr[i-1]]
  }
  else
    dp[i][j] = dp[i-1][j]
}
return dp[arr.size()][sum];
```

10

Minimum Subset Sum Difference

Given,

arr = [1 | 2 | 7]

O/P = 4

S_1 S_2

$\Sigma(1,2)$ - $\Sigma(7)$

We have to return minimum diff. between S_1 and S_2.

$\min(\Sigma arr(4) - S_1)$

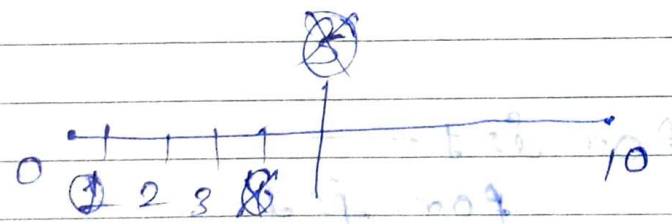
$S_1 - S_2 \Rightarrow \min$

$\Rightarrow S_2 - S_1$

$\Rightarrow (Range - S_1) - S_1$

$Range - 2S_1$

Steps ⇒



S_1 should be in range/2.

So we have, {1, 2, 3}

$Range - 2S_1$

if		
$S_1 = 0$	$10 - 0 = 10$	min = 4
$S_1 = 1$	$10 - 2 = 8$	
$S_1 = 2$	$10 - 4 = 6$	
$S_1 = 3$	$10 - 6 = 4$	

code is on DSA 2nd copy

PAGE NO

Note ⇒ This problem is similar to subset-sum problem.

subset sum code

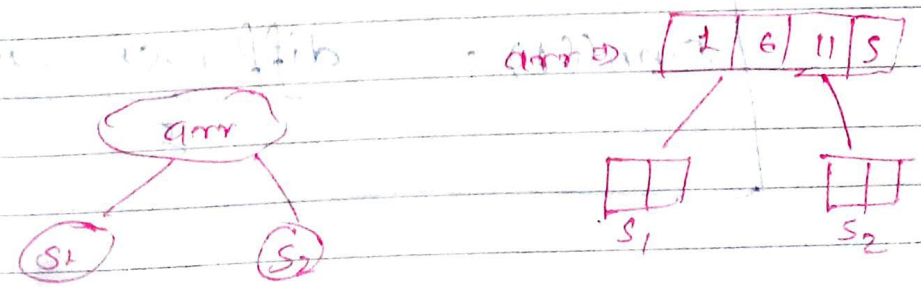
↓

Iterate through the last row of matrix and mid

↓

When we encounter true apply this
Range e - 2S1
and return minimum of all.

Practiced on GFG



abs(S1 - S2) ⇒ min

11

Count the number of subset with a given difference

Given,

arr = [1 | 2 | 4 | 10] diff = 2

$sum(S_1) - sum(S_2) = diff$ — (i)

$sum(S_1) + sum(S_2) = sum(arr)$ — (ii)

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

from eq. (iii),

We have,

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

return countofsubsetsum(arr, sum/2)

12.

Target Sum

Target Sum is same as

← Count the number of subset with a given diff. problem

Here given in,

Target Sum

arr

Sum

Count # of subset

arr.

diff.

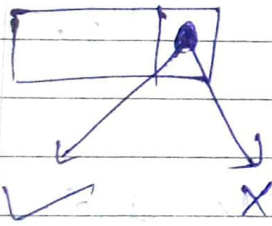
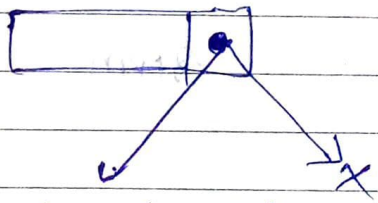
So Actually they are same.

Unbounded knapsack.

Related problems:

(1.) Rod cutting
(2.) Coin change I
(3.) Coin change II
(4.) Maximum ribbon cut.

Difference

Knapsack	Unbounded knapsack
 (processed) (processed)	 (processed) (—)
	We can have multiple occurences of same item.
CODE	CODE
(1) Initilize with 0 for row and col (same as before).	(1) same as it is ←

PAGE NO.

Knapsack

$$dp[i][j] = \max\left(\begin{matrix} val[i-1] + dp[i-1][j-val[i-1]] \\ dp[i-1][j] \end{matrix}\right),$$

else,

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

Unbounded Knapsack.

$$dp[i][j] = \max\left(\begin{matrix} val[i-1] + dp[i][j-val[i-1]] \\ dp[i-1][j] \end{matrix}\right)$$

else

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

14

Rod Cutting Problem

This problem is same as unbounded knapsack.

Given

length() = [| | | |]

price() = [| | | |]

N = 4

CODE

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

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

else

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

Note:

In Kallication is same as unbounded knapsack.

Coin Change 2 : max # of ways

given

coin[] = [1 | 2 | 3]

sum = 5

→ Supply of these coins are unlimited.

o/p : 5

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

CODE

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

Coin Change II ⇒ min # of Coins.

Given,

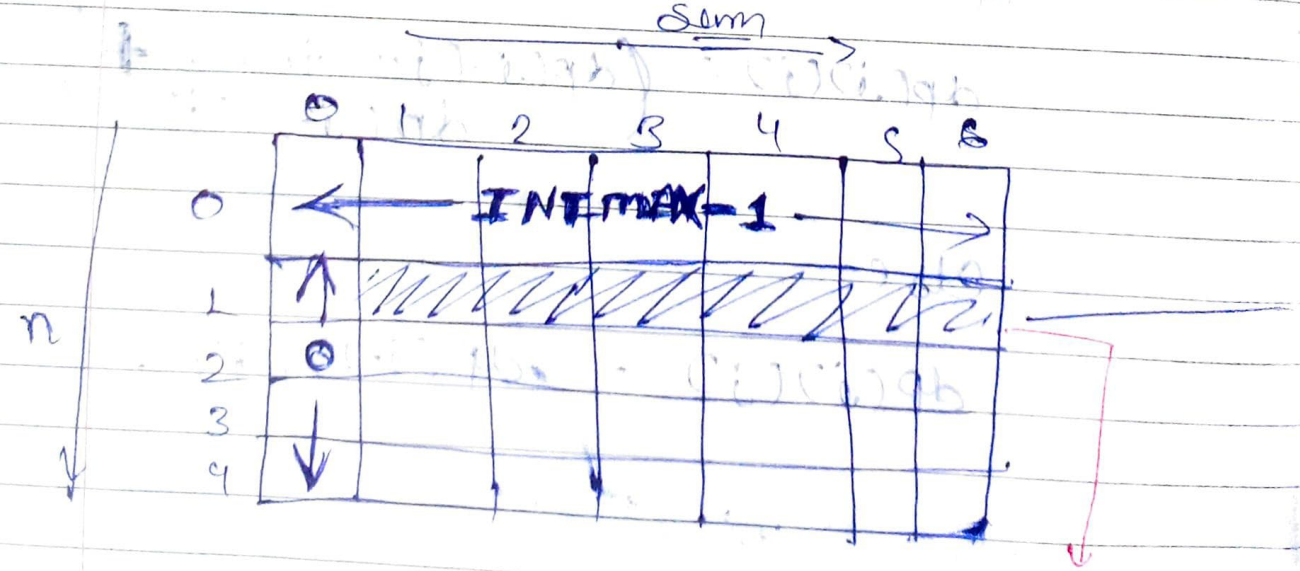
coins[] : [1][2][3]

Sum = 5 ⇒ O/P ⇒ 2.

Steps ⇒ n = coins.size();

⇒ dp[n+1][sum+1] ⇒ dp[4][6]

Initialisation ⇒



PAGE NO:

Initialization (code)

```
for i = 0 --> sum
    dp[0][i] = INT_MAX - 1;

for i = 1 --> n
    dp[i][0] = 0;
```

```
for(int j = 1; j <= sum; j++)
  {  if( j % coins[0] == 0)
       dp[1][j] = j / coins[0];

     else
       dp[1][j] = INT_MAX - 2;
  }
```

CODE

```
if( coin[i-1] <= j)
dp[i][j] = min( dp[i][j - coin[i-1]] + 1,
                dp[i-1][j] );

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

Longest Common Subsequence.

variation:

(1.) Longest Common substring.
(2.) Print LCS.
(3.) Shortest Common supersequence.
(4.) Print SCS
(5.) min # of insertion and deletion.
(6.) Longest repeating subsequence.
(7.) length of longest subsequence of a which is a substring is b.
(8.) Subsequence pattern matching.
(9.) Count how many times a appears as ~~substring~~ subsequence in b.
(10.) Longest Palindromic subsequence.
(11.) ~~Count~~ Count of palindromic substring.
(12.) min # of deletion in a string to make it a palindrome.
(14.) min # of Insertion in a string to make it a palindrome.

Longest Common Subsequence / LCS (Recursive)

Given:

I/P:

x : a b c d g h

y : a b e d f h r

O/P → 4

Recursive Approach

- 3 steps

(1.) Base Condition.
(2.) minimizing the I/P.
(3.) Choice Diagrame.

x : [" _ "] n

y : [" _ "] m

(1.) Base Conditions

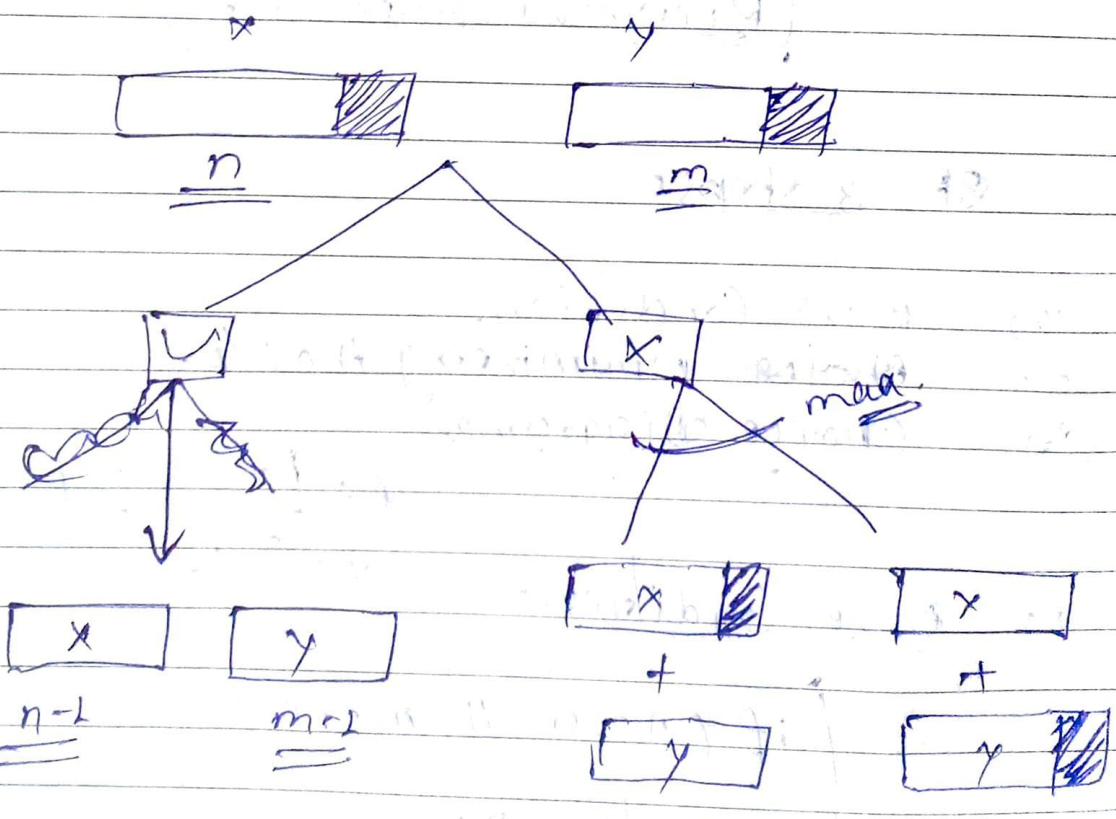
```
if (n==0 || m==0)
    return 0;
```

(2.) Reducing :

x : "[a b c | d]" y : "[g f c | d]".
pick pick.

pick the last item and minimize the length by 1.

(3.) Choice Diagram :



Recursive CODE

```
int LCS (string x, string y, int n, int m)
{
    if (n==0 || m==0)
        return 0;
```

→ Base Condition

```
    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) );
}
```

← choice diagram implementation

20.

LCS [memorized] (Bottom UP (DP))

Note:- 3 ways to solve any DP Q.

(1.) Recursive.
(2.) table + Recursion ⇒ memorized
(3.) table ⇒ top-down

⇒ We create table for those variables whose value's changing.

CODE

Global declaration of table ⇒

```
int static t[1009][1001];
```

Inside main function intiallize it with -1. ⇒

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

```
int LCS(string x, string y, int m, int n)
{
    if(n==0 || m==0)
        return 0;
```

→ Base condn

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

→  (checking if the value is already present)

```
    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) );
```

storing ← the value in the table

```
    return(m][n]);
}
```

← Ans

T.C. → $O(n^2)$
S.C. → $O(n^2)$

21.

LCS : Top-Down (DP)

PAGE NO.:

Why top-down?

In memorization we can have stack overflow. That's why it is safer to use Top-down.

x = "abcf" → m = 4

y = "abcdaf" → n = 6

} O/P = 4

t[4+1][6+1] =

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1
2	0	1	2	2	2	2	2
3	0	1	2	3	3	3	3
4	0	1	2	3	3	3	4 ← ans

Note: First row and first column will be initialized with base condition.

~~for(i)(2) = "a" ? (5) ; .~~

"a" /

table creation ← `int t[m+1][n+1];`

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

Initialization

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

filling the table By Choice Diagram ←

Any ← `return t[m][n]`

22.

Longest common Subsstring

Given,

a : abcde } m = 5

b : abfce } n = 5.

Strings are,

→ ab
→ c } longest (ab)
→ e.

O/P ⇒ 2

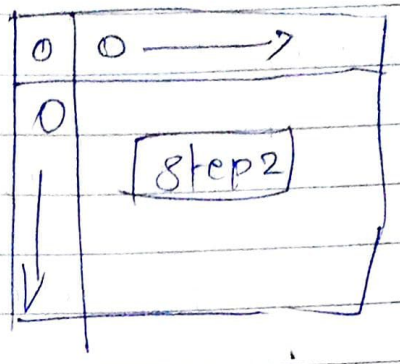
Note ⇒ It should not be discontinuous.

Step 1⇒ Initillization is same as LCS.

PAGE NO.

t[m+1][n+1];

Step 2 :)



for i : 1 ⟶

for j = 1 ⟶

if(a[i-1] == b[j-1])

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

else

t[i][j] = 0;

28. **Printing Longest Common subsequence**

Given,

I/p ⇒

a: a c b g f

b: a b c d a f

O/p ⇒

"abcf"

table of given strings.

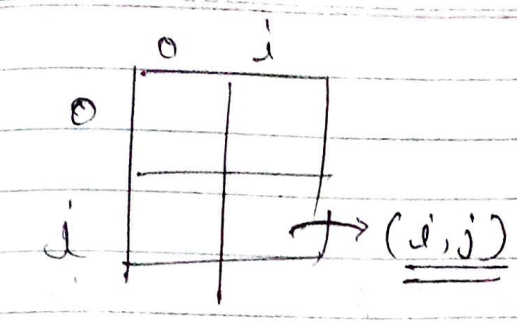
b: a b c d a f ⇒ m = 6

a: a c b c f ⇒ m = 5

	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	2	2	3	4

ans

Note




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

then,

(1.) Start from t[m][n]

Check

```
if ( i==j)
  i--; j--; [store the char]
else
  max ( i-1, j
        i, j-1 )
```

(1.) create table
(2.) Call LCS and fill the table.
then,

CODE

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

24. **Shortest Common Supersequence (SCS)**

Given

a = AGGTAB m = 6

b = GXTXAYB n = 7

Shortest Supersequence length,

O/P ⇒ 9

Explain ⇒

a + b

Supersequences ⇒ AGGTAB GXTXAYB (13)

a b

LCS (common elements) ⇒ GTAB (4)

We have to return shortest supersequence length.

return m+n − LCS(a, b, m, n)

Ans ↳ AGGXTXAYB O/P ⇒ 9

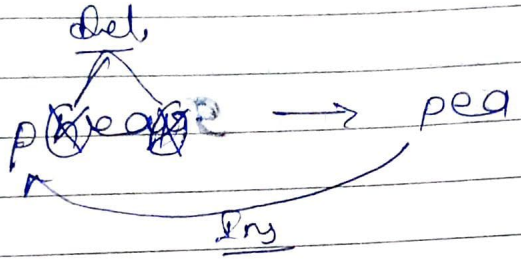
Minimum Number of Insertion and Deletion to convert string a to b

Given,

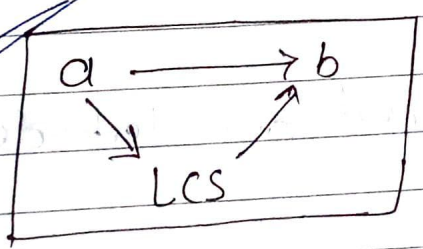
I/P ⇒ a = heap
b = pea

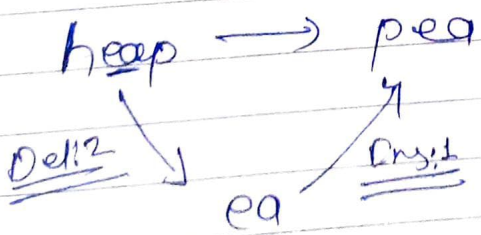
O/P ⇒ Ins = 1
Del = 2

Explain



Solution ⇒





# of Ins. = b.length() − LCS
# of Dels = a.length() − LCS

26.

Longest Palindromic Subsequence (LPS)

I/P ⇒ s = agbcba O/P ⇒ 5

(abcba)

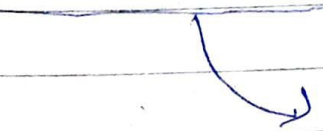
LPS(s) ≡ LCS(s, reverse(s))

Practiced over leetcode.

27.

Minimum number of deletion in a string to make it a palindrom

I/p ⇒ s: "agbcba" | O/p → 1



⇒ length of LPS $\propto \frac{1}{\#\text{ of deletion}}$

LPS (s) = LCS (s, reverse(s))

return s.length() − LPS

28.

Print Shortest Common Supersequence.

I/p ⇒ a = "acbcf" m=5

b = "abcdaf" n=6

O/p ⇒ "acbcdaf"

Note ⇒ This question is same as Printing LCS. (minor diff)

CODE

Step1 ⇒ table creation of t[m+1][n+1];

Step2 ⇒ Initializing it 0th row and 0th colmn. with 0.

Step3 ⇒ fill the table same as LCS.

These steps are same as LCS. (dikto)

mainsteps CODE

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

29. # Longest Repeating Subsequence.

Given,

str = "a a b e b c d d"

O/P = 3

⇒ a b d is repeating and so its length is 3

Note:- This Q. is same as LCS, just one restriction is that we can't include same index.

CODE

```
String s = "a a b e b c d d"
String a = s
String b = s

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] );

return t[m][n];
```

30.

Sequence Pattern Matching

Given,

I/P ⇒

```
a = "axy"
b = "adxcpy"
```

O/P ⇒ True

⇒

```
LCS(a, b)

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

Steps ⇒

(1) Write LCS code.

(2) then add a check.

That's it ☺ ✓.

81.

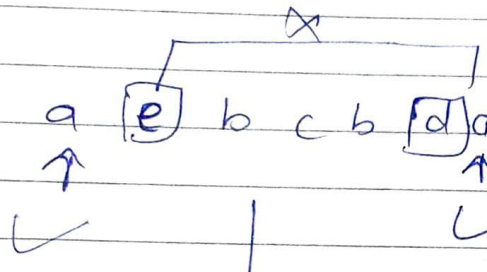
Minimum Number of Insertion in a string to make it palindrom.

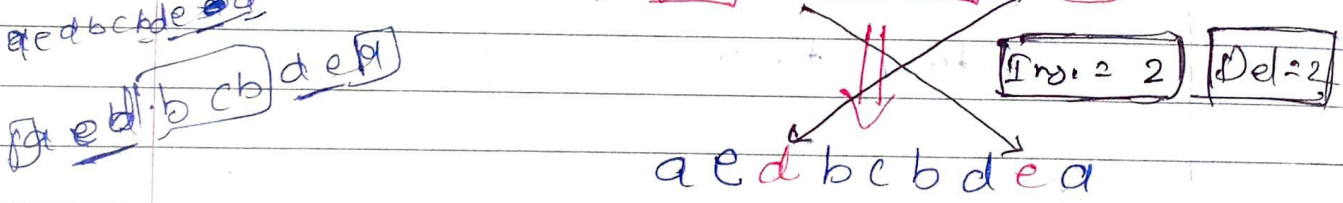
Given

S = " a e b c b d a "

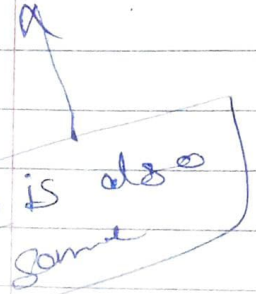
O/p => 2.

Note





Note => This Q. is exactly same as # of deletion



return S.length - LPS(S, reverse(S))

(abcba)

7 - 5 = 2

Matrix Chain multiplication (mcm)

→ knapsack
→ LCS
→ mcm

32.

(1.) mcm ↔ Recursive
memorication

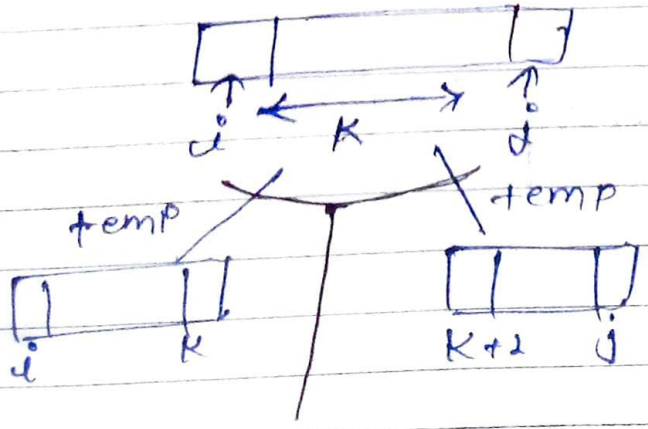
[variation]

(2.) Printing mcm
(3.) Evaluate Expression to True/Boolean Parenthesization
(4.) Min/Max value of an Exp.
(5.) Palindrome Partitioning.
(6.) Scramble String.
(7.) Egg dropping problem.

[mcm] ↔ [Identification & General Form]

Identification:

Will be given an Array or String



return ans = f(min, max)

General Format

```
int solve( int arr[], int i, int j )
{
    if ( i > j )
        return 0;

    int ans = INT_MAX;

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

        ans = fun.(ans, temp);
              (min, max)
    }

    return ans;
}
```

Base Condition ← `if ( i > j ) return 0;` → d/f.

main Break points ← `int temp = solve(arr, i, k) + solve(arr, k+1, j);`

Optimal ← `ans = fun.(ans, temp);` (min, max)

Best of all ← `return ans;`

Matrix Chain multiplication (mcm)

(Recursive)

General Concept ⇒

$[\,]_{a\times b} \times [\,]_{b\times c} \Rightarrow [\,]_{a\times c}$

It should become

cost ⇒ $a\times b\times c$

Eg ⇒ [A] 10×30 [B] 30×5 [C] 5×60

2 combinations.

A(BC)

$(10\times 30)\times(30\times 5\times 5\times 60)$

~~$10\times 30\times 30\times 5\times 60$~~

~~$10\times 30\times 5\times 60\times 60$~~

$30\times 5\times 60 + 10\times 30\times 60$

⇒ 27000 ✓

(AB)C

$(10\times 30\times 30\times 5)\times(5\times 60)$

$\Rightarrow 10\times 30\times 5 + 10\times 5\times 60$

⇒ 4500 ✓

Given,

arr[] = (40, 20, 30, 10, 30)

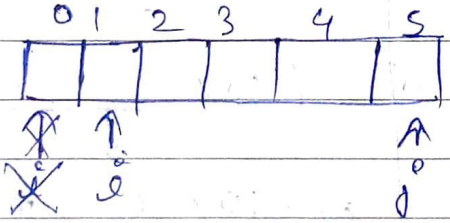
then,

$A_1 = 40 \times 20$
$A_2 = 20 \times 30$
$A_3 = 30 \times 10$
$A_4 = 10 \times 30$

formula

$A_i = arr(i-1) \otimes arr(i)$

Step 1 ⇒ Find i and j values.



not valid ✗ $A_0 = arr(0-1) \otimes arr(0)$

So,

smallest valid index ✓ $A_1 = arr[1-1] \otimes arr[1]$ $i = 1$

✓ $A_5 = arr[5-1] \otimes arr[5]$ $j = n-1$

Solve(arr, i, j)

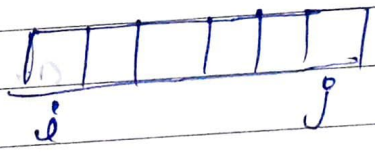
Step 2 ⊙ Finding the Base Condition.

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

Step 3 ⊙ Finding the range of K

Two Scheme

a)	K = i , K = j-1	i to K , K+1 to j
b)	K = i+1 , K = j	i to K-1 , K to j



Total steps ⊙

(1.) Find i, j.
(2.) Find BC.
(3.) Find K loop ⊙ scheme.
(4.) Calculate ans from temporary.

(Max Product Array)

CODE (Recursive)

```
int solve( int arr[] ,int i, int j)
{
    if ( i >= j)
        return 0;

    int ans = 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];

        if (temp < ans)
            ans= temp;
    }

    return ans;
```

B.C ← if (i >= j) return 0;

Initialization ← int ans = INT_MAX;

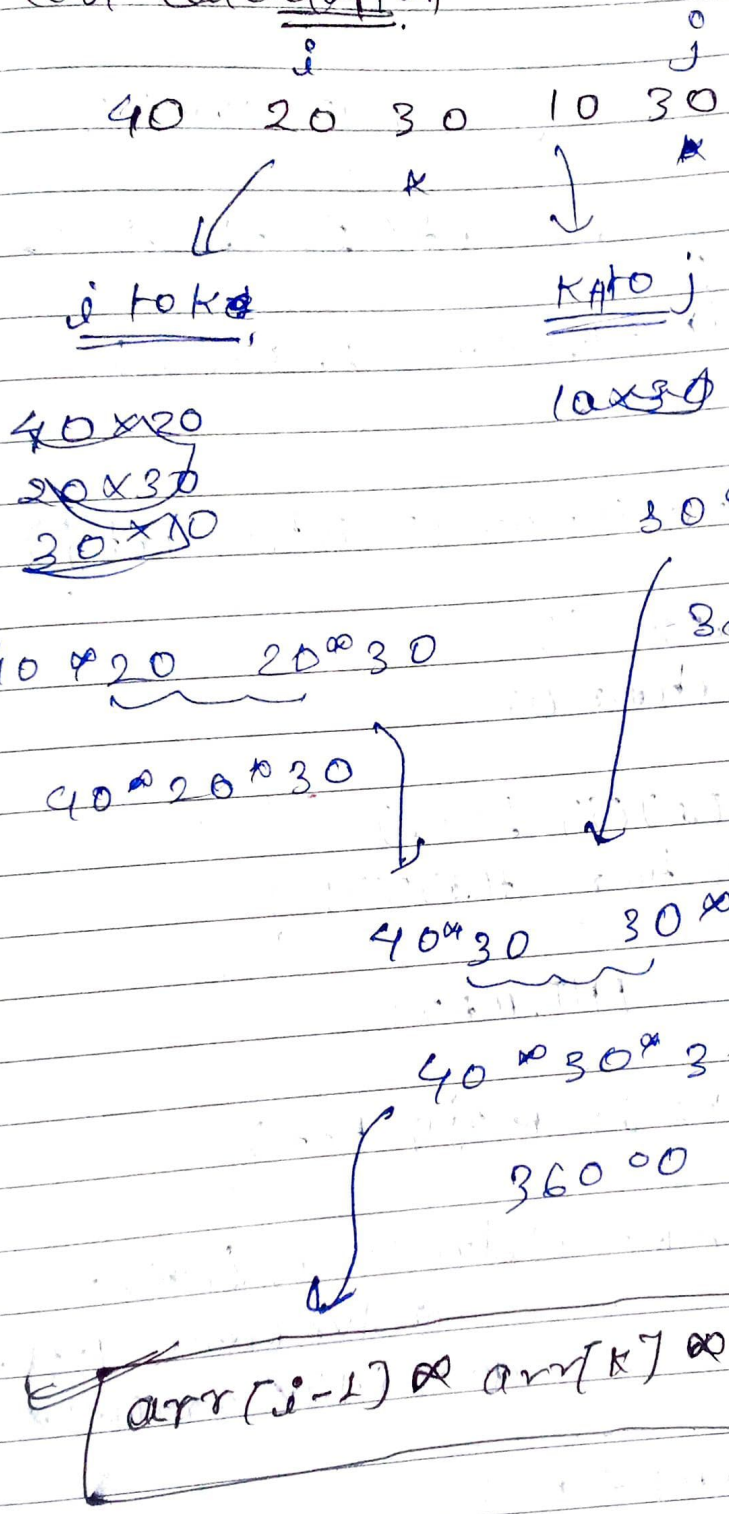
Loop ← for(int k= i; k<= j-1; k++)

Main ← int temp= solve(arr, i, k) + solve(arr, k+1, j) + arr[i-1] * arr[k] * arr[j];

min ← if (temp < ans) ans= temp;

2nd Scheme i to j-1

Extra Cost Calculation



84.

mcm memorization

Step 1 ⇒

Global ⇒ `t[1001][1001];`

main ⇒ `memset(t, -1, sizeof(t));`

↳ `Solve(arr, 1, n-1);`

```
int solve(int arr[], int i, int j)
{
    if(i >= j)
        return 0;

    if(t[i][j] != -1)                    // check-point
        return t[i][j];

    int ans = 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];

        if(temp < ans)
            ans = temp;
    }

    return t[i][j] = ans;                // storing ans in table
}
```

mcm

A = 10 × 30 , B = 30 × 5 , C = 5 × 60

Give the minmum no. of operation for ABC = ?

Cost

1) (A)(BC) ⇒ (10×30)((30×5)(5×60)) ⇒ 30×5×60 = 9000

= (10×30)(30×60) ⇒ 10×30×60 = 18000

= 10×60 ⇒ = 0

27000

2) (AB)(C) ⇒ ((10×30)(30×5))(5×60)

⇒ 10×30×5 = 1500

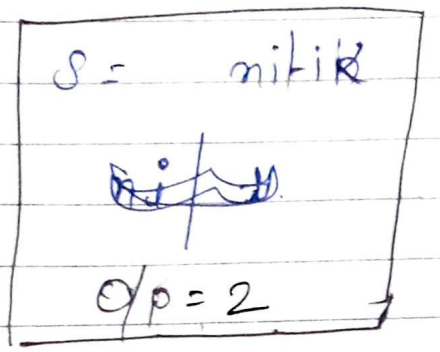
= (10×5)(5×60) ⇒ 10×5×60 = 3000

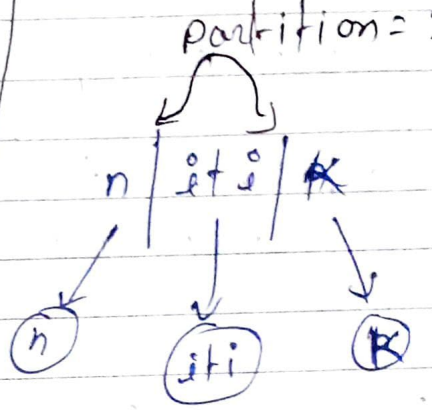
= 10×60 ⇒ 0 =

4500

35. **Palindrome Partitioning** (Recursive) (mcm-~~format~~ variation)

Given,





Steps :-

(1.) find i and j
(2.) BC.
(3.) k loop range.
(4.) min of temps.

From main function,

```
Solve(s, 0, n-1)
```

CODE (Recursive)

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

    if(isPalindromic(s, i, j))
        return 0;

    int ans = INT_MAX;

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

      if( temp < ans )
         ans = temp;

    }

    return ans;
```

empty or single char ← `if( i >= j ) return 0;`

check if it is already palindromic ← `if(isPalindromic(s, i, j)) return 0;`

`int ans = INT_MAX;` → This should be outside of this fn.

partitions ← `for(int k=i; k<=j-1; k++)` { `int temp = 1 + solve(s, i, k) + solve(s, k+1, j);` `if( temp < ans ) ans = temp;` }

Palindrome Partitioning

(IB)

36.) Memorization (Bottom Up)

DP

Recursive + table

Global ⇒ `int static t[1001][1001];`

main ⇒
```
memset(t, -1, sizeof(t));
solve(s, 0, n-1);
```

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

    if(isPalindromic(s, i, j))
        return 0;

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

    for(int k=i; k<=j-1; k++)
    {
        int temp = 1 + solve(s, i, k) + solve(s, k+1, j);
        if(temp < ans)
            ans = temp;
    }
    return t[i][j] = ans;
}
```

Check if i,j is already processed.

Storing the processed item

(VVIP) 28. Evaluate Expression to True Boolean Parenthesizing Recursive.

Given,

S = "T | F & T ^ F (True)

O/p = ~~Total~~ min# of ways it is true. (2)

This Q. is same as mcm.

Note

T ^ T → false
F ^ F → false
T ^ F → True
F ^ F → True.

CODE (Recursive)

from main() {

solve(s, 0, n-1, true);

memorization code

int solve(string s, int i, int j, boolean isRequired)

{

if (i > j)
return 0;

empty string ←

→ Base condition

```
if(i==j)
{
    if(isRequired == 'T')
      return true;

    else
      return false;
}
```

```
if (i==j)
{
   if(isRequried == true)
   {
      if(S[i]=='T')
         return true;
      else
        return false;
   }
  else if (isRequried == false)
   {
     if (S[i]=='F')
         return true;
     else
         return false;
   }
}
```

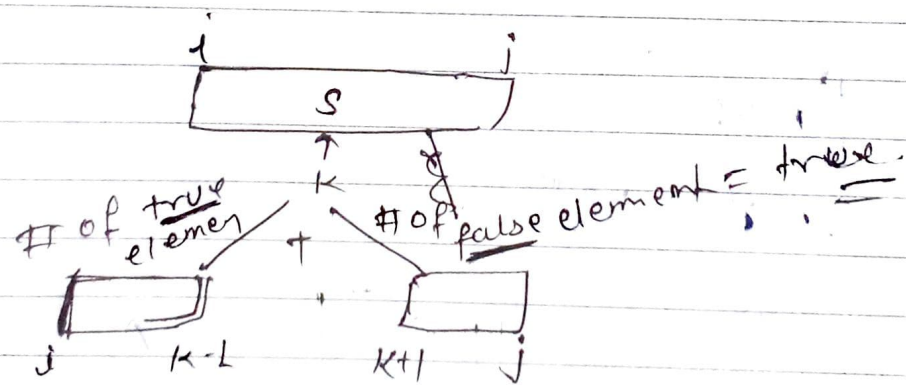
→ Single element.

(memorization code) →

```
int ans = 0;

for(int k = i+1; k<=j-1; k=k+2)
{

  int lt, lf, rt, rf;

  lt = solve(s, i, k-1, true);
  lf = solve(s, i, k-1, false);

  rt = solve(s, k+1, j, true);
  rf = solve(s, k+1, j, false);


  if(s[k] == '^')
  {
    if(isRequired == true)
      ans += lt * rf + lf * rt;

    else
      ans += lt * rt + lf * rf;
  }
  else if(s[k] == '&')
  {
    if(isRequired == true)
      ans += lt * rt;

    else
      ans += lt * rf + lf * rt + lf * rf;
  }
```

```
else if (s[k] == '|')
{
    if (isRequired == true)
        ans += lt * rf + lf * rt + lt * rt;
    else
        ans += lf * rf;
}
}
return ans;
```

memorization code



DP. memorization

array as an matrix

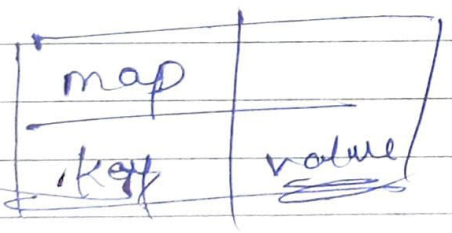
Dimensions → Number of variables that are changing in function call

In this problem,

dimention of matrix → i × j × is required.

```
int t[1001][1001][2];
```

Better way



&&	||	^
0 && 1 = false	0 || 0 = false	1 ^ 1 = false
1 && 0 = false	1 || 1 = true	0 ^ 0 = false
0 && 0 = false	1 || 0 = true	1 ^ 0 = true
1 && 1 = true.	0 || 1 = true.	0 ^ 1 = true

39.

memorization code

Global ⇒ `unordered_map<string, int> mp;`

main ⇒

```
mp.clear();
solve(_____);
```

code variation

string

```
i + " " + j + isTrue
```

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

making string

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

check

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

storing new items

40.

Scrambled string

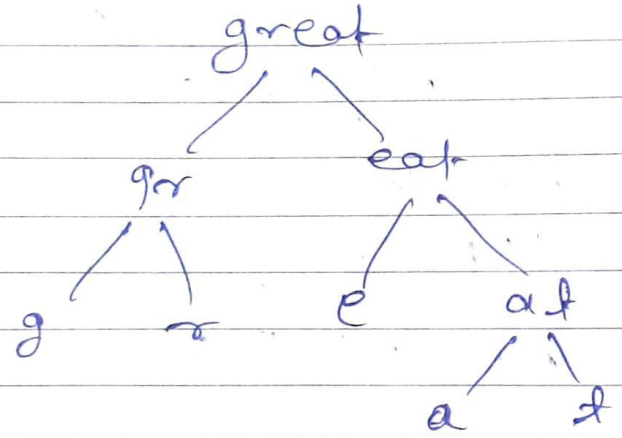
Recursive

Given two strings

s1 = great
s2 = rgreat.

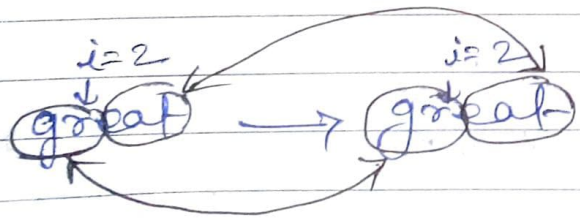
o/p ⇒ True

We can convert it into Binary tree.



We have to swap it's non-leaf nodes.

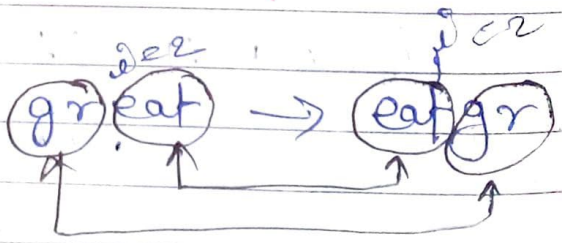
Condition I



Same String

↳ Scrambled

Condition II



Non-similar scrambled string

Condition II

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

Condition I

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

Base Case

(main function)

Non-similar string ←

Empty string ←

Same string ←

```
if(a.length() != b.length())
        return false

int n = a.length();
if(n==0)
        return true;

if(a==b)
        return true;

Solve(a,b);
```

```
bool flag = false;
bool Solve(string a, string b)
{
    for(int i=1; i<n; i++)
    {
        if(condition I || condition II)
        {
            flag = true;
            break;
        }
    }
    return flag;
}
```

41.

memorization (DP)

Global ⇒ `unordered_map<string, bool> mp;`

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

Base Condition

creating string ←

```
string temp = a;
temp.push_back(' ');
temp.append(b);
```

checking ←

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

```
for i → n-1
    condition
```

storing ←

```
return mp[temp] = flag;
```

}

42.

Egg Dropping Problem

Recursive

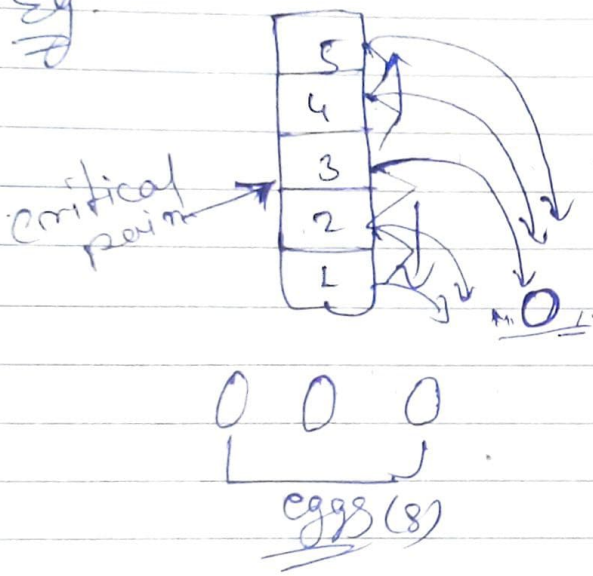
Given,

e = 3
f = 5

O/P
3

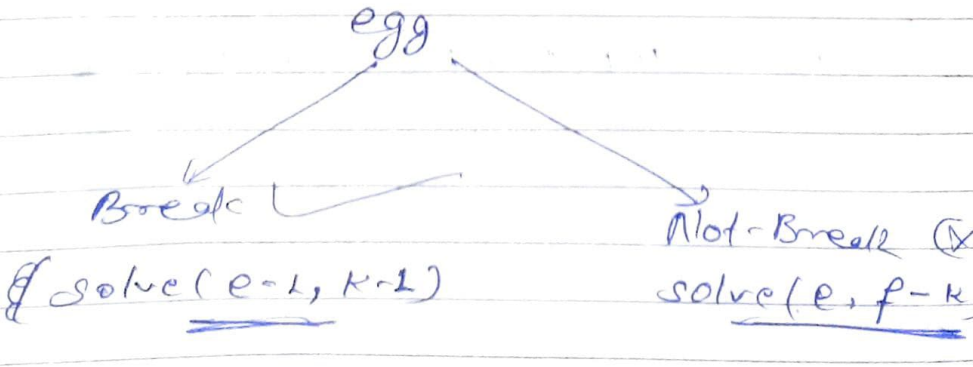
min # of attempts
↓ in
worst case

Eg:



2 condition
1) if ~~break~~ then go down 1
2) else go up ↑

P.S. It's MCM variation.



CODE (Recursive)

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

    if(e==1)
        returnf;
```

memorization

```
    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;
```

memorization

44

Memorization (DP)

Globally ⇒ `int t[101][101];`

main ⇒
```
memset(t, -1, sizeof(t));
solve(e, f);
```

Check ⇒
```
if(t[e][f] != -1)
    return t[e][f];
```

Store ⇒
```
return t[e][f] = mn
```

45

Dynamic Programming on trees

Content:

- General Syntax.
- How DP can be applied to trees (Identification)
- Diameter of a Binary Tree.
- Maximum path sum from any node to any.
- Maximum path sum from leaf to leaf.
- Diameter of N-ary trees.

General Syntax

```
int fun ( Int )
{
    Base Condition
    Hypothesis
    Induction
}
```

4.6

General Syntax

```
int solve( Node* root, int &res )
{
```

Base Cond. ←

```
    if (root == NULL)
        return 0;
```

Hypothesis ←

```
int l = solve( root->left, res );
int r = solve( root->right, res );
```

Induction ←

```
int temp = 1 + max(l, r);
int ans = max( temp, relation );
                      ↳ (1+l+r)

res = max(res, ans);
```

```
return temp;
```

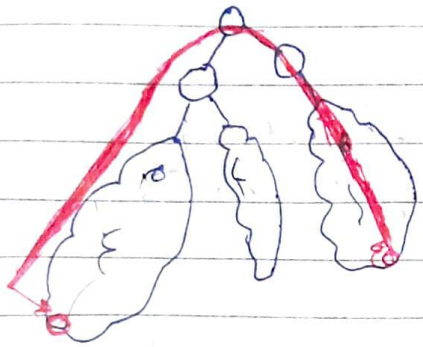
```
main()
{
    int res = INT_MIN;
    solve( root, res );
    return res;
}
```

47

Diameter of a Binary Tree

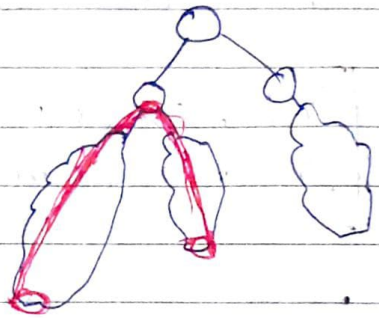
maximum distance between two leaf nodes.

Case I



passing through root

Case II



passing through arbitary root

CODE

```
int solve(Node* root, int& res)
{
    if(root == NULL)
        return 0;

    int l = solve(root->left, res);
    int r = solve(root->right, res);

    int temp = max(l, r) + 1;
    int ans = max(temp, l+r+1);
    res = max(res, ans);

    return temp;
}
```

48. # Maximum Path Sum

From any node to any node

CODE

```
int solve(Node* root, int& res)
{
```

B.C. ←
```
if(root == NULL)
    return 0;
```

Hypothesis ←
```
int l = solve(root->left, res);
int r = solve(root->right, res);
```

Induction ←
```
int temp = max(max(l,r) + root->val,
               root->val);

int ans = max(temp, l+r+root->val);

res = max(res, ans);
```

moving ahead ←
```
return temp;
}
```

```
main()
{
    int res = INT_MIN;
    solve(root, res);
    return res;
}
```

Maximum Path Sum

49. From leaf node to leaf node

CODE

```
int solve(Node* root, int& res)
{
    if(root == NULL)
        return 0;

    int l = solve(root->left, res);
    int r = solve(root->right, res);

    int temp = max(l,r) + root->val;

        if(root->left == NULL && root->right==NULL)
            temp = max(temp, root->val);

    int ans = max(temp, root->val + l + r)

    res = max(res, ans);

    return temp;
}

main()
{
    int res = INT_MIN;
    Solve(root, res);
    return res;
}
```

Basic

Hypothesis

Inductive step

Moving ahead

Main func

50. Minimum Edit Distance

Given,

Eg1 s1 = abc O/P = 2

s2 = bcd abc(a)

We have to convert s1 to s2 in min # of steps

Operations : i) Insert.
ii) Remove.
iii) Replace.

Eg 2

s1 = S u n d a y

s2 = S a t u r d a y

a t u r r

a t u r

O/P = 3

dp[4][4]

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

ans

a b c

b c d

(a) → (b)

(a) → ab

(a) → abc

CODE

finding min of three #

```
int min(int x, int y, int z)
    return min(min(x,y), z);
```

```
int editDistDP(string s1, string s2, int m, int n)
{
```

creating table

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

```
for(int i=0; i<=m; i++)
{
    for(int j=0; j<=n; j++)
{
```

if first string is empty then second is any

```
if (i==0)
    dp[i][j] = j;
```

A vice-versa

```
else if(j==0)
    dp[i][j] = i;
```

If last chars are same then ignore the last char

```
else if (s1[i-1] == s2[j-1])
    dp[i][j] = dp[i-1][j-1];
```

else

Insert

dp[i][j] = 1 + min (dp[i][j-1] , dp[i-1][j] , dp[i-1][j-1]);

Remove

Replace

If last chars are d/f then compare b/w given operations

}

}

cmy ← return dp[m][n];

}

T.C = O(MN)

S.C = O(MN)