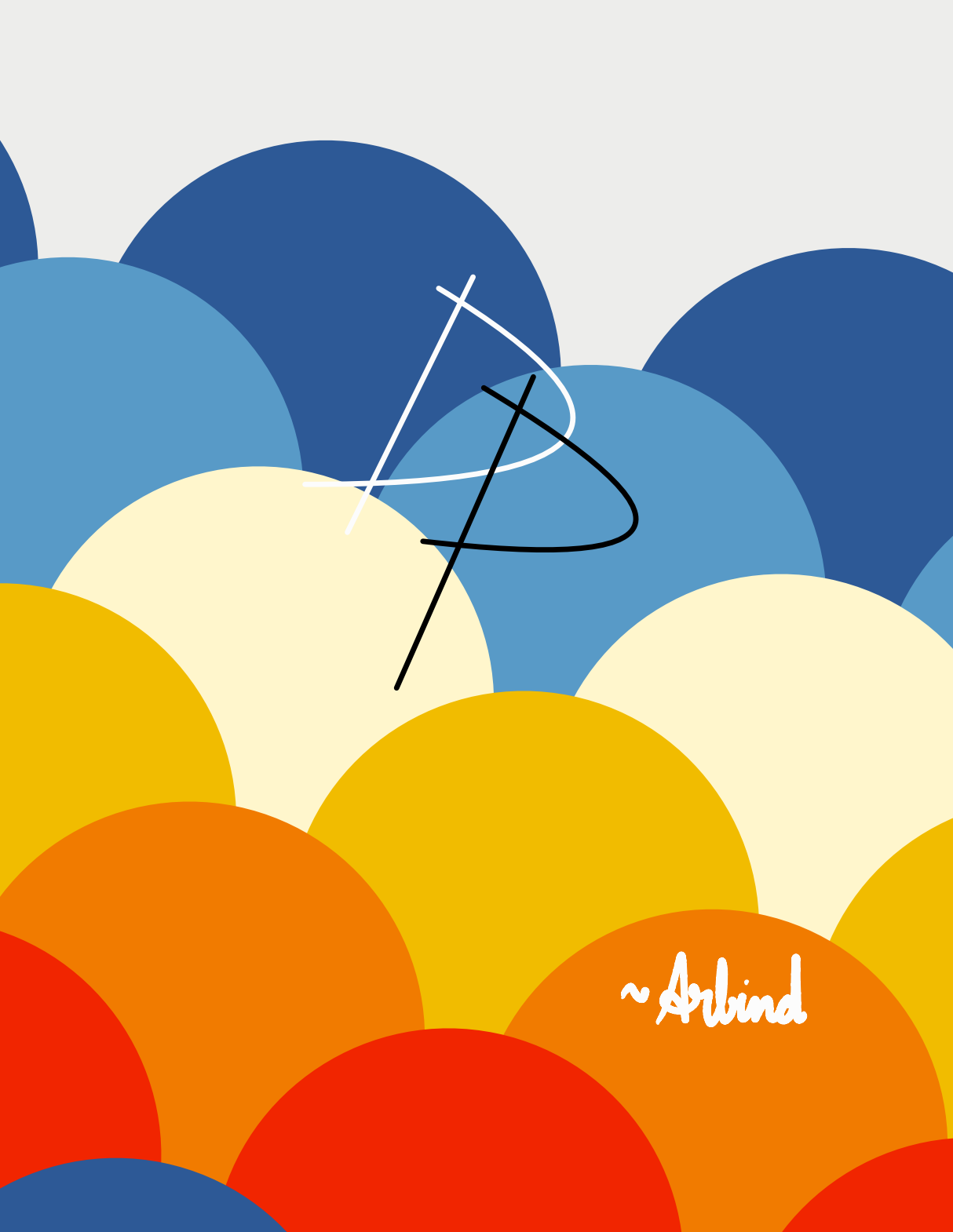

~Arbind

Dynamic Programming

DP = Recursion + Storage

Indentify
→ Enhanced Recursion → Choice

→ Recursive → 2 calls

→ Optimal → min / max / largest

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 (7)
8) DP on Trees (4)
9) DP on Grid (14)
10) Others (5)

0-1 knapsack Problems

1) Subset Sum
2) Equal Sum Partition
3) Counts of Subset Sum
4) Min. Subset Sum Diff
5) Target Sum
6) no. of subset of given diff.

Knapsack Problemm → Unbounded Knapsack (Unlimited supply)

(Greedy) ↓ Fractional Knapsack ↳ (We can select fraction of item)

0/1

	I 1	I 2	I 3	I 4
wt[]:	1	3	4	5
val[]:	1	4	5	7

O/P: Max Profit

w = 7kg

⇒ 0/1 Knapsack Recursive

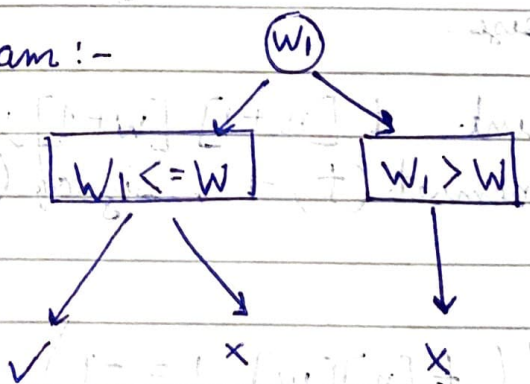
Recursive ↓ Choice

⇒ I/P wt[]: 1 | 3 | 4 | 5

val[]: 1 | 4 | 5 | 7

w = 7

Choice Diagram :-



⇒ Base Condition → Think of the smallest valid I/P

wt[]

val[]

wt:

n → 0

0

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

for (IP)

↓

for (Smaller IP)

```
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
    return knapsack(wt, val, w, n-1);
```

Memorization

n x m

↓ ↳ weight

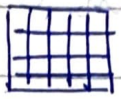
size.

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

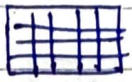
```
{ if( t[n][w] != -1)
      return t[n][w]
{ if(...)
  return t[n][w] = ...
{ else
    return t[n][w] = ...
```

Top Down DP

Memoization → AC+



Top Down →



→ Can give stack overflow (Scrambled String)

⇒ We mak table based on changing value in the recursive function

Step 1 - Initialize

Step 2 - Recursive → Iterative

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

val[ ] = [1 4 5 7]

W = 7

t[n+1] [W+1] t[5] [8]

val	wt		0	1	2	3	4	5	6	7 → W[j]
1	1	0	0	0	0	0	0	0	0	0
4	3	1	0							
5	4	2	0							
7	5	3	0							
		4	0							→ Max profit

n [i]

wt = 1 3 4 val = 1 4 5

Recursive

→ if(n==0||w==0)
return 0;

→ if(wt[n-1]<=w)

→ return max(val[n-1] + knapsack(wt, val, w-wt[n-1], n-1); knapsack(wt, val, w, n-1)

→ else
return knapsack(wt, val, w, n-1)

Top-Down

→ for(int i=0; i<n+1;
for(int j=0; j<w+1;
if [i==0||j==0]
t[i][j]=0

→ if (wt[n-1]<=w)

$$t[n][w] = \max(val[n-1] + t[w-wt[n-1]][n-1], t[n-1][w])$$

$$t[n][w] = t[n-1][w]$$

Code:

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

```
        else
            dp[i][j] = dp[i-1][j];
        }
    }
}
cout << dp[n][w];
```

⇒ wt[ ]:
val[ ]:
element having 2 property
iW : capacity

if only one array is given, we will consider it as wt array.

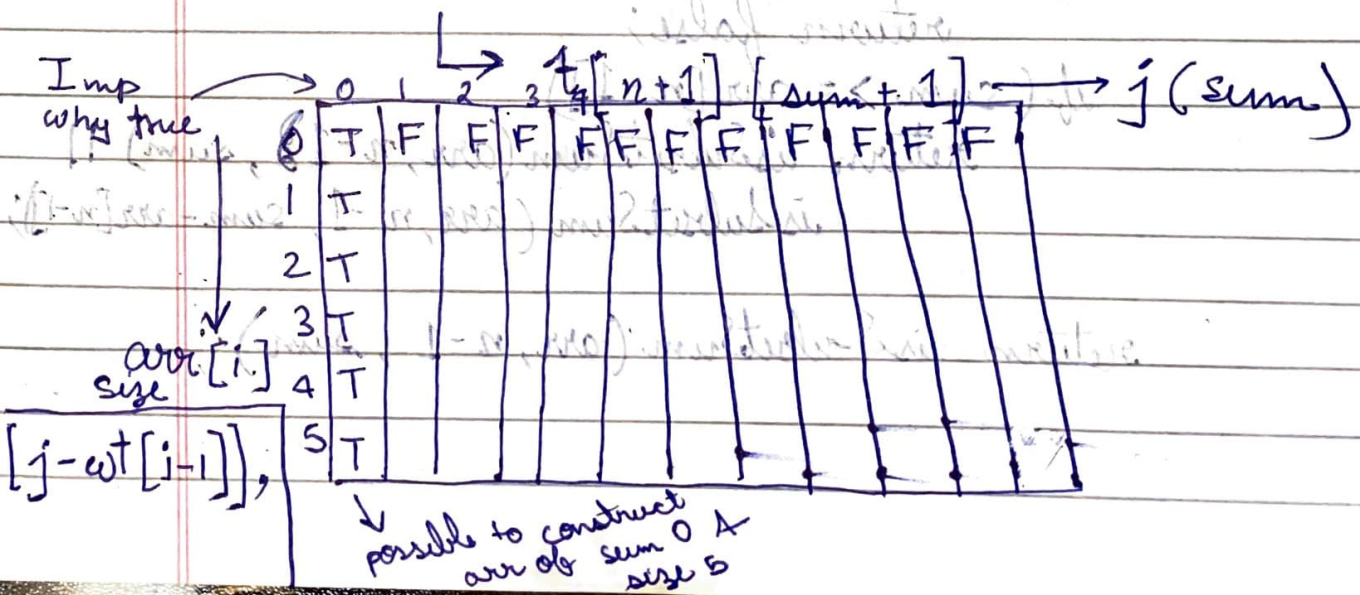
[n-1]

Subset Sum Problemm

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

Logic:

So we have two choices either to take the no. in the array (or) not.



Code:

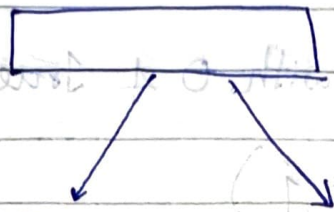
```
bool subset[n+1][sum+1];

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

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

for (int i=1; i<=n ; i++){
    for(int j=1; j<=sum ; j++){
        if (j >= arr[i-1])
            subset[i][j] = subset[i-1][j] ||
                subset[i-1][j-arr[i-1]]
        else
            subset[i][j]=subset[i-1][j];
    }
}

cout subset[n][sum];
```

Recursive Code

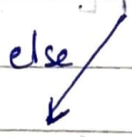
```
if(sum ==0)
    return true;
if(n==0)
    return false;
if( sum >= arr[n-1])
    return isSubsetSum(arr, n-1 , sum) ||
        isSubsetSum(arr, n-1, sum-arr[n-1]);

return isSubsetSum(arr, n-1 , sum);
```

Equal Sum Partition

arr[] : { 1 5 11 5 }



odd



even

else

one of the subset have sum/2

Code :

input : arr[] , n

```
for(int i=0 ; i<n ; i++)
        sum += arr[i]

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

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

is a subset exists with sum/2 than an another subset exists with same sum

Count of Subset Sum with a Given Sum

arr[]: 2 3 5 6 8 10

sum = 10

⇒ intialize false with 0 & true with 1

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

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

Min. Subset Sum Difference

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

O/P = 1

{1+6+5} - {11}

= 12 - 11 = 1

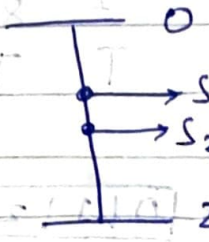
$S_1 - S_2$ = min

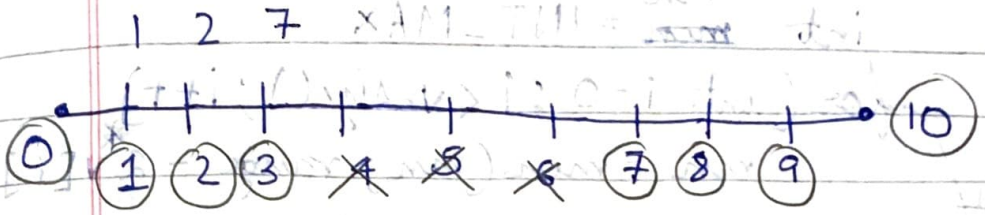
| 1 | 6 | 11 | 5 |

→ { } → $S_1 = 0$

→ {1, 6, 11, 5} → $S_2 = 23$

Range





↳ the number which can be formed by using above subset

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

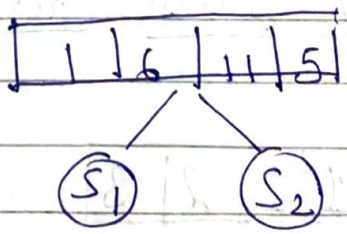
↓

If we know S_1 we can find $S_2 = Range - S_1$

$(S_1 - S_2) \Leftarrow$ Minimise

$|S_2 - S_1| = Range - S_1 - S_1 = \boxed{Range - 2S_1}$ → diff

this must be min



$Range - 2S_1 \Rightarrow$ Minimize

$\Rightarrow S_1$ will lie in range/2

$\Rightarrow$ The row no. in dp[][] tells us the sum when given size $S[1][6]$

size ↓ ↳ sum exists or not

10/2 = 5 (checking)

arr[] : 1 2 7

dp[3][ ]

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

range

Code :

v : [0|1|2|3]

```
int mn = INT_MAX
for(int i=0; i<v.size(); i++)
    mn = min(mn, range - 2*i)
return mn;
```

Efficient

```
for(int i= sum/2; i>=0; i--){
    if(dp[n][i]){
        ans = sum - 2*i;
        break;
    }
}
```

Count the no. of subset with a given difference

arr[] : [1|1|2|3]

Diff = 1

$\Rightarrow$ $S_1 - S_2 = Diff$ — ①

$sum(S_1) + sum(S_2) = sum(arr)$ — ②

① + ②

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

$$sum[s1] = \frac{diff + sum[arr]}{2}$$

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

$$\Rightarrow \boxed{sum[S1] = 4}$$

Target Sum

arr : | 1 | 1 | 2 | 3 | ⇒ assign sign.

sum : 1

OP : 3

+1 -1 -2 +3

+1+3 -1-2

4 -3 = 1

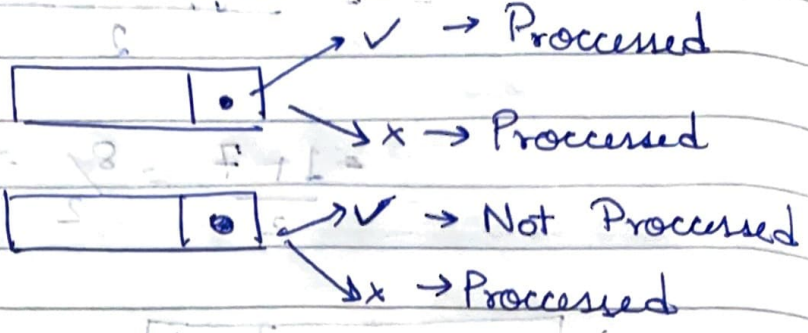
Same prob. as the previous ques.

Unbounded Knapsack Related Problemns

1) Rod Cutting
2) Coin Change I
3) Coin Change II
4) Max. Ribbon Cut

⇒ Multiple occurances of same item

Diff



Code

```
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]
```

Rod Cutting Problem

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

N : 8

We can cut the piece of same length multiple times

wt → length[]
val → Price
W → N

No. of ways we can form the sum

Coin Change I

⇒ where we don't have 2 arrays, ignore val[]

coin[]: | 1 | 2 | 3 |

sum: 5

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

} 5 ways

wt → coin ()
W → sum

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]
```

Min no. of coins Coin Change II

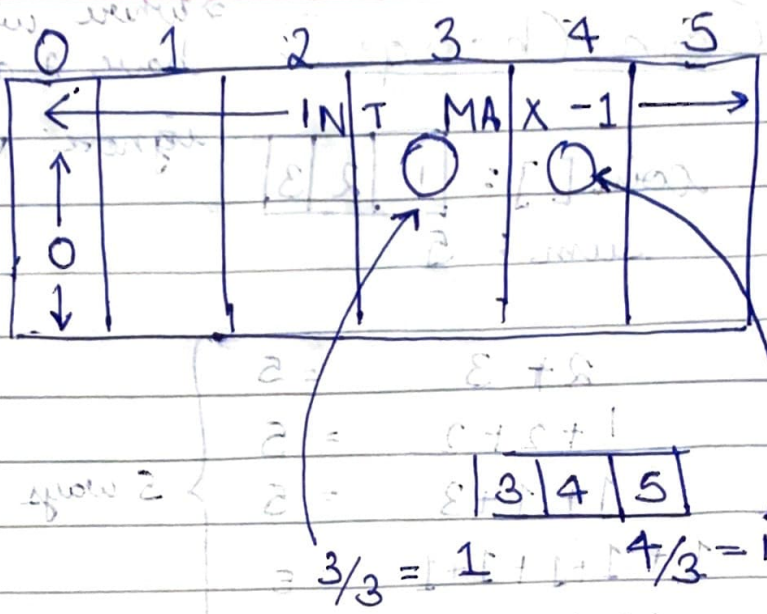
Coin[]: 1 2 3
sum: 5

Note:

→ We have to initialize 2nd row

Note:

⇒ 4 suppose we try to achieve 4 in 2nd row we get 3, 6, 9...



Loop for 2nd row

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

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];
```

for this we took INT_MAX-1

Longest Common Subsequence

1) Largest Common Substring
2) Print LCS
3) Shortest Common ... supersequence
4) Print SCS
5) Min. no of insertion & deletion a→b
6) Largest Repeating Subsequence
7) Length of Largest Subsequence of 'a' which is a subsstring in 'b'
8) Subsequence Pattern Matching
9) Count how many times a appears as subsequence in b.
10) Largest Palindromic subsequence
11) Largest Palindromic substring
12) Count of Palindromic Substring
13) min no. of deletion in a string to make it a palindrome
14) min no. of insertion in a string to make it a palindrome.

Largest Common Subsequence

abdh

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

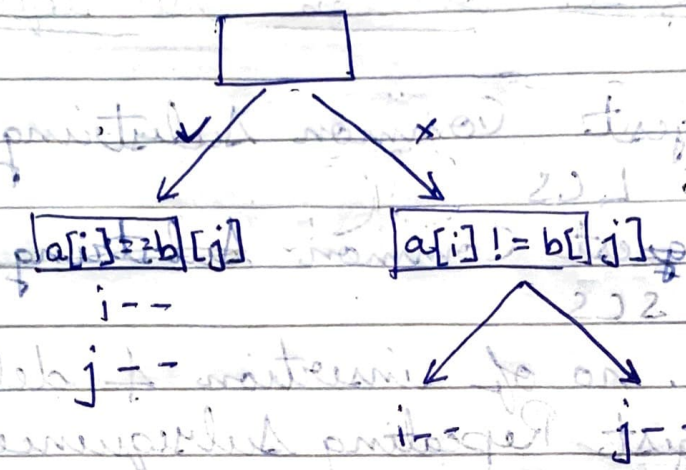
Base Condition

Smallest valid Input $m \to 0$, $n \to 0$

⇒ if (!n || !m)
    return 0;

int func( ){
Base condition
choice Diagram
}

⇒ Recursion




```
int LCS( string X, string Y, int n,
int m) {
    if(!n || !m)
        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));
}
```

Top-Down Approch DP

Code :

```
if(dp[m][n] != -1)
    return dp[m][n];
if(x[m-1] == Y[n-1])
    return dp[m][n] = 1 + LCS(x, y, m-1,
                              n-1);
```

Top-Down DP Approch

X : a b c d e f

Y : a c b c f

LCS ⇒ abcf = 4

⇒ initialize i == 0 j == 0 with 0

Code:

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

```
cout << dp[n][m] << endl;
```

Longest Common Substring

a: a b c d e

b: a b f c c

O/P: continious = 2

if char doesn't match ⇒ make it 0

Code:

```
int mx=0;
    if(a[i-1] == b[j-1])
        dp[i][j] = dp[i-1][j-1] + 1;
    else
        dp[i][j] = 0;
    mx = max(mx, dp[i][j])
```

Print LCS between 2 Strings

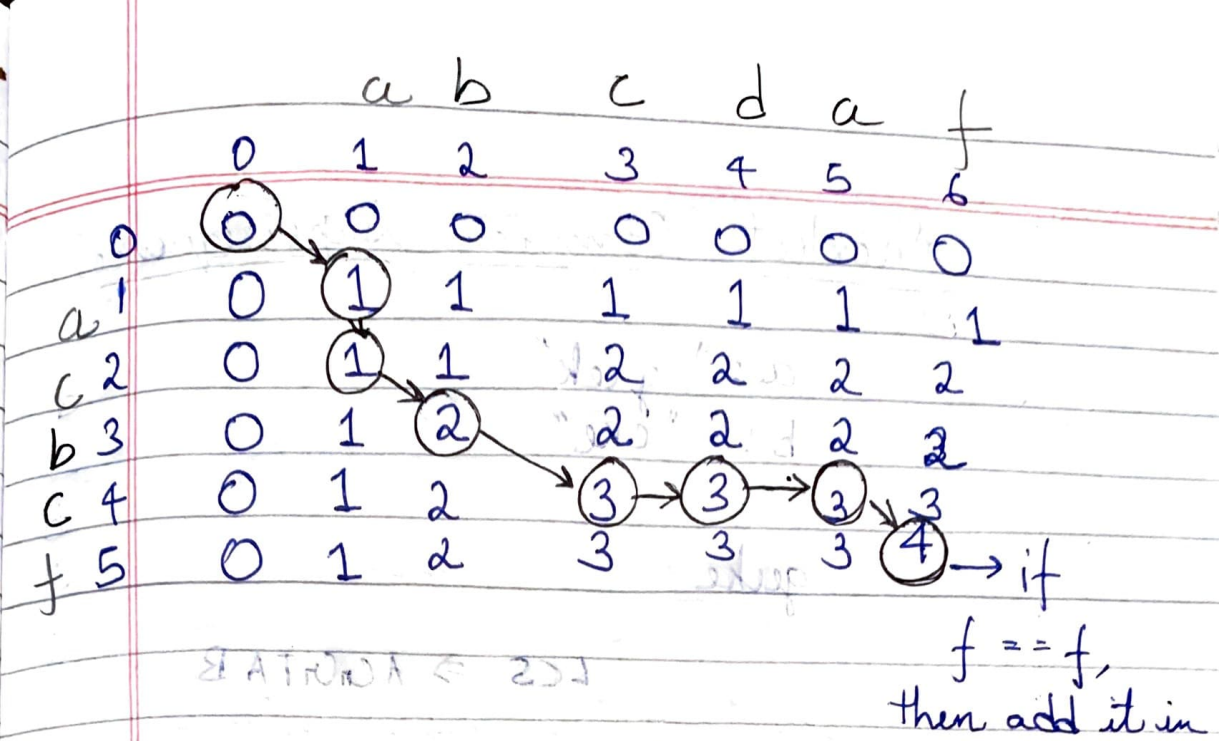
a: @ c b c f → n(5)

b: @ b c d a f → n(6)

t[n+1][n+1] ⇒ t[6][7]

		a	b	c	d	a	f
	0	0	0	0	0	0	0
a	1	0	1	1	1	1	1
c	2	0	1	1	2	2	2
b	3	0	1	2	2	2	2
c	4	0	1	2	3	3	3
f	5	0	1	2	3	3	4

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



"f c b a"

if equal i, j → i--, j--

not equal i, j → max([i-1, j], [i, j-1])

Code LCS +

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

Shortest Common Supersequence

a: "geek"
b: "eke"

geeke

→ LCS ⇒ AGGTAB

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

⇒ A G G X T X A Y B → in the order

→ Total Length = 9

worst case

⇒ $m+n$ - LCS

$m = 6 \quad n = 7 \quad LCS = 4$

$7+6-4 \Rightarrow 13-4 = 9$

Edit Distance ↑ famous question

Min. No of Insertion / Deletion

a: heap
b: pea

a → b

→ p ~~h~~ eap Inser. → 1

Dele. → 2

	I/P	Q	O/P
LIS :	a: , b:	LCS	int
Q :	a: , b:	min.	int

2/3 ⇒ matching → the question is of similar question

(a) heap → LCS

(b) heap → pea → ea

No. of Del = a.Length − LCS

No. of Insert = b.Length − LCS

Longest Palindromic Subsequence

I/P s: agbcba O/P : 5

agbcba ⇒ abcba ⇒ 5

agbcba

↳ LCA

Min. no. of deletion in a string to make it a Palindrome

I/P s: "agbcba" O/P : 1

agbcba

③ bcb (bc, b) ⑤ bcb (c) ① abcba (abcba)

Length of LPS $\propto \frac{1}{\text{no. of Deletion}}$

LPS(s) → LCS(s, reverse(s))

⇒ ans = s.length - LPS

⇒ $O(N^2) + O(N^2)$ time

Longest Palindromic Substring

a a a a b b a a

↘

LPS

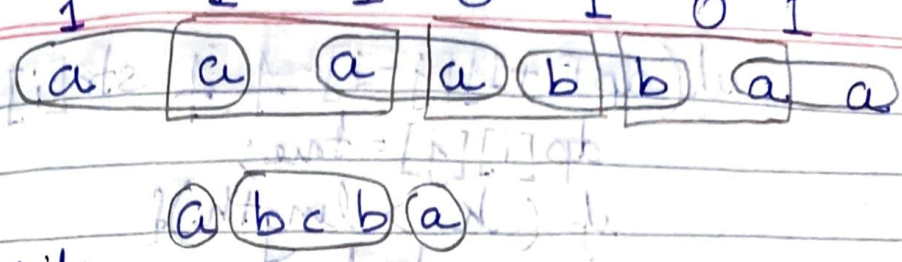
Intialization

→ This can't be included in proper substring

		a ↓	a ↓	a	a	b	b	a	a
		0	1	2	3	4	5	6	7
a	0	1	1	1	1	0	0	0	0
a	1		1	1	1	0	0	0	0
a	2			1	1	0	0	0	1
a	3				1	0	0	1	0
b	4					1	1	0	0
b	5						1	0	0
a	6							1	1
a	7								1

↘ always true

① ②



Condition

i) s[start] == s[end]

ii) non-boundary substring should be palindrome

$O(N^2)$

⇒ str[i] == str[j] && dp[i+1][j-1] == 1

↳ Note: first two diagonals are filled because this condition is not valid for str of len '1' & '2'.

Code:

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

for(int i=0; i<n-1; i++){
    if(str[i] == str[i+1]){
        dp[i][i+1] = true;
② start = i;
        length = 2;
    }
}
```

↳ length of the string

```
for(int k=3; k<=n; k++){        → starting index
    for(int i=0; i<=n-k+1; i++){
ending index  int j = i+k-1;
```

```
if(dp[i+1][j-1] && str[i]==str[j]){
    dp[i][j] = true;
    if (k > length){
        start = i;
        length = k;
    }
}
}
}
}
```

```
cout << length << endl;
```

Print SCS

I/P
a : a c b c f
b : a b c d a f

O/P

	∅	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

f a d c b c a

Code

```
while(i>0 && j>0)
{
  if(a[i-1] == b[j-1])
  {
    s.push_back(a[i]);
    i--;
    j--;
  }
  else{
    if(dp[i][j-1] > dp[i-1][j]){
      s.push_back(str2[j-1]);
      j--;
    }
    else if(dp[i-1][j] > dp[i][j-1]){
      s.push_back(str1[i-1]);
      i--;
    }
  }
}
while(i>0){
  s.push_back(a[i-1]);
  i--;
}
while(j>0){
  s.push_back(b[j-1]);
  j--;
}

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

Longest Repeating Substring

str = "AABEBCDD"

(A)(A)(B)E(B)C(D)(D)

O/P: ABD

S = "A A B E B C D D"

C, E occured once. don't take

So, A → 0 1
A → 0 1

⇒ i == j
don't take

Code: change in LCA code

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

Sequence Pattern Matching

a = "AXY"
b = "(A)D(X)CP(Y)"

subsequence → order
↘ discontinuous

LCS range ⇒ 0 to min(m, n)

```
if( LCS. length == min(m, n))
        return true
else
        return false;
```

Min no. of insertion in a string to make it a palindrome

S: "a e b c b d a"

d e

a	e	b c b	d	a

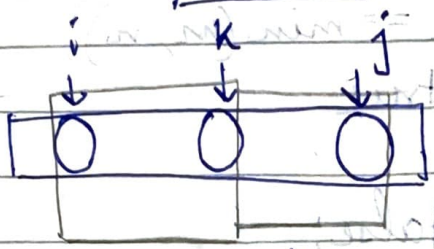
∴ No. of insertion = No. of deletion

Matrix Chain Multiplication

1) MCM
2) Print MCM
3) Evaluate Expr. to True / Boolean Parenthesization
4) Min / Max. value of an Expr.
5) Palindrome Partitioning
6) Scramble String
7) Egg Dropping Problem

Identification + Format

MCM → String (or) arr



temp ans

Base Condition

- the smallest valid inputs
- think of invalid I/P

we will change k to another point & again do the above stem

temp ans
temp ans
temp ans
→ f = ans

Format

```
int solve(int arr[], int i, int j){
    if(i > j) → diff based on question
        return 0;
    for(int k=i; k<j; k++){ → can change
      //calculate temp ans
      tempans = solve(arr, i, k) + solve
                (arr, k+1, j)
      ans ← func(tempans)
    }
    return ans;
}
```

Cost = 2 [3 3] 6 — 2×3 3×6

= 2 3 6 — Matrix Size

Matrix Chain Multiplication

arr[] = {40 20 30 10 30}

A → 10×30
B → 30×5
C → 5×60

1. Find i, j
2. Find Base Condition
3. Find K loop schema
4. calculate ans from temp_ans

(AB)C → 10 × [30 30] × 5

→ 10 × [5 × 5] × 60

→ 10 × 5 × 60

10 * 30 * 5 + 10 * 5 * 60

= 4500

A(BC) → 30 × [5 5] × 60

30 × 5 × 60

10 × [30 30] × 60 + → 2700 MIN

= 10 × 30 × 60

A_i ⟶ arr[i-1] * arr[i]

A_1 → 40×20
A_2 → 20×30
A_3 → 30×10
A_4 → 10×30

we can't choose i from here

40	20	30	10	30
0	1	2	3	4

i = 1

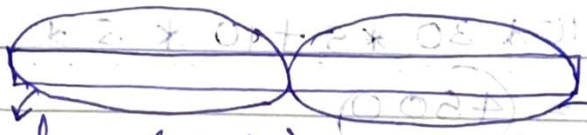
j = n - 1

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

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

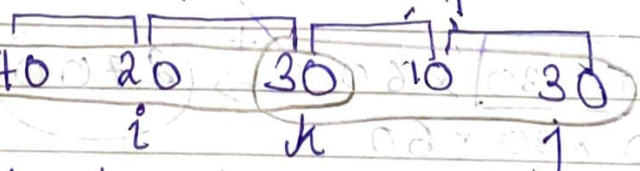
Scheme 1 ⇒ k = i to k = j-1 (i to k) (k+1 to j)

Scheme 2 ⇒ k = i+1 to k = j (i to k-1) (k to j)



solve (i to k) (AB)

solve (k+1 to j) (CD)



⇒

i to k

40 × 20 20 × 30

40 × 20 × 30

k+1 to j

30 × 0 10 × 30

30 × 10 × 30

Dimension 40 × 30 30 × 30

40 20 30 10 30

40 x 30 30 x 30

= 40 x 30 x 30

temp ans arr[i-1] * arr[k] * arr[j]

Code

```
int tempans = solve(arr, i, k) +
              solve(arr, k+1, j)
              + arr[i-1] * arr[k] *
              arr[j]
                    (cost)

if (temp < mn)
{
  mn = temp
}
}
return mn;
```

---

40 20 30 10 30

i i=k ✓ j j=k → Solve will start for k+1 ✗

30x10

K

40 20 30 10 30 ⇒ i to k-1

⇒ k to j

i=k ✗ Not valid

j=k ✓

Matrix Chain Multiplication Memorization

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

int solve(){
    if(i>=j)
        return 0;

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

    return t[i][j]=min;
}

int main(){
    memset(dp,-1,sizeof dp);
    solve( );
}
```

Palindrome Partitioning :

S: nitin → 0 ✓

n|i+i|n → 2

(n) (iti) (n)

i=0
j = size(1) - 1

i to k
k+1 to j

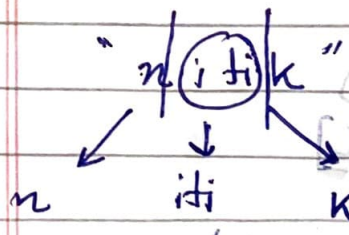
```
int solve (                                    ) {
    if (i >= j)
        return 0;

    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;
}
```

Code:

```
if (is palindrome (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 (temp, mn);
}
return mn;
```

Top Down

Palindrome Partitioning



```
bool isPalindrome (string s,
int i, int j){
    if(i >= j)
        return true
    while (i <= j){
        if(s[i] != s[j]) return false;
        i++, j--;
    }
return true;
}
```

Code

```
int solve (
    .
    .
    .
    if (isPalindrome (s, i, j))
        return 0;
    if ( dp[i][j] != -1
    .
    .
    .
    .
    return dp[i][j] = mn ;
}
```

Optimization

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

⇒ in this case we only check if [i to j] is computed before so now we will check for [i to k] & [k+1 to j] also

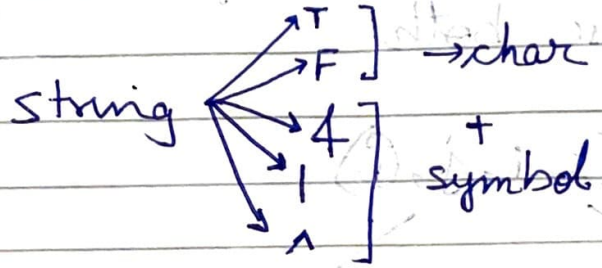
Code:

```
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 = l + left + right;
```

Evaluate Exp To True

String : T xor F and T



or — |
and — &
xor — ^

T ^ F & T

Nor. of ways to make it true

((T ^ F) & T) (T ^ (F & T))

T & T → True T ^ F → True

⇒ (T | F &) T (^ F)

↳ as this case is not valid we must take k = k+2

1. Find i & j

```
i = 0;
j = s.length() - 1;
```

2. Base Condition

i (T or F and T) k xor (F) j

i to k-1 | k+1 to j

⇒ Exp1 xor Exp2

No. of true = lT * rF + lF * rT

⇒ in 'xor' we must find subproblems for true & false both

for (s, i, j, T)

for(s, i, k, T) for(s, k+1, j, F)

⇒ if(i > j) return false;

if(i == j){

if(isTrue){

return s[i] == 'T';

}

else

return s[i] == 'F';

3. K Loop

```
int ans = 0;
for (int k = i+1 ; k <= j-1 ; k = k+2){
    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, F)
}
```

4. lT lF (T) XOr (T) rT rF

```
ans += lT * rF + lF * rT
```

Symbol ⇒ 4 | i | T

```
if (s[k] == '4'){
    if(isTrue)
        ans += lT * rT
    else
        ans += lF * rT + lF * rF + lT * rF
}
else if (s[k] == '1'){
    if(isTrue)
        ans += lT * rT + lT * rF + lF * rT
    else
        ans += lF * rF
}
```

```
else if (s[k] == '^') {
    if (isTrue)
        ans += lF * rT + lT * rF
    else ans += rT * lT + rF * lF
}
return ans
```

(both the expression must be same to make it false)

Memorization

changing values ⇒ i, j & isTrue

dimension of matrix ⇒ i x j x isTrue

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

⇒ we can also use map

i+j+isTrue ⇒ "50 90 F" → 9

```
unordered_map<string, int> ump;

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

before for loop

```
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];
```

⋮

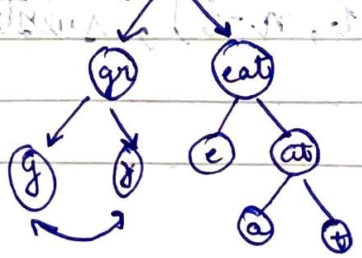
```
return mp[temp] = ans;
```

Scrambled String

a : "great" O/P : True
b : "rgeat"

Constrain :-

1. Binary Tree
2. Child must not be empty
3. Leaf node can be swapped

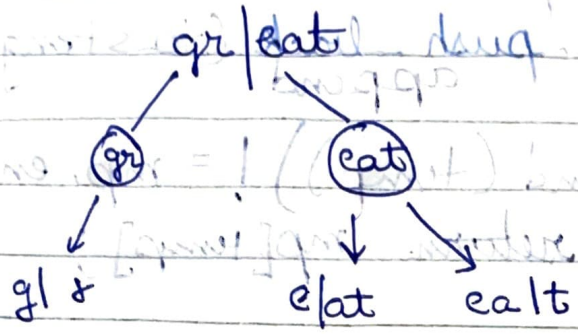


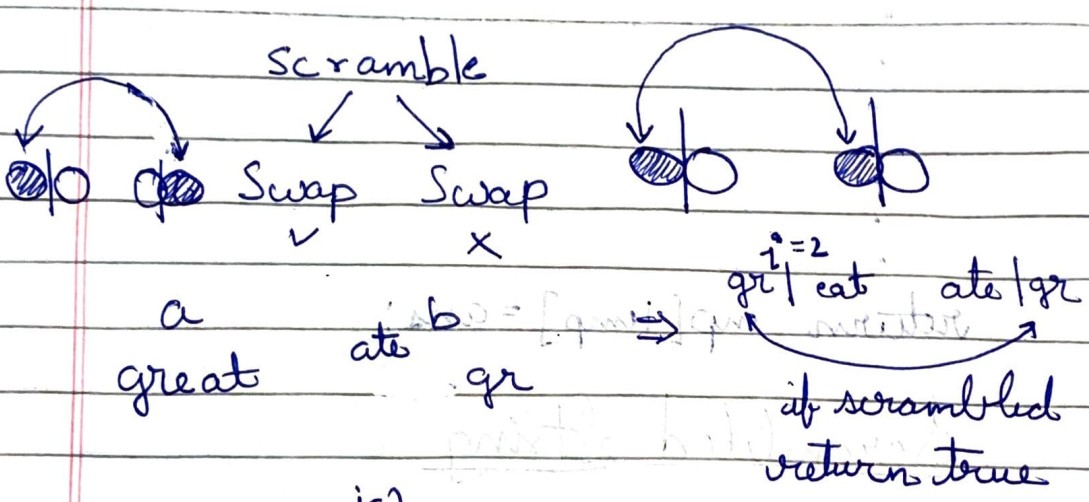
rgeat

great

i=1 i=2 i=3 i=4

i = 1
i = n-1





Case I: Swap | gr|eat cat|gr (i=2)

for the above case ↑ last 2

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

Case II: gr|eat gr|ate

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

swapped ↩ ↪ not swapped

If (Case I || Case II)
then true

⇒ **Base Conditions**

✓ if (a.length() != b.length())
return false;

if (!a.length() && !b.length())
return true;

⇒ ✓ if (a.compare(b) == 0) ← if a single char is left & they are not equal return false
return true; "abc" "a"

we always take a & b of equal len. ✓ if (a.length() <= 1)
return false;

Code

```
bool solve(string a, string b){
    if(a.compare(b) == 0)
        return true;
    if (a.length() <= 1)
        return false;
    int n = a.length();
    bool flag = false;
    for(int i = 1; i <= n-1; i++){
        if (condition 1 || condition 2)
        { flag = true;
          break;
        }
    }
return flag };
```

,i))

Memorized

if not declared pass it by refrence '&'

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

variables which are changing

```
key = a + "_" + b
```

Store value correspondingly

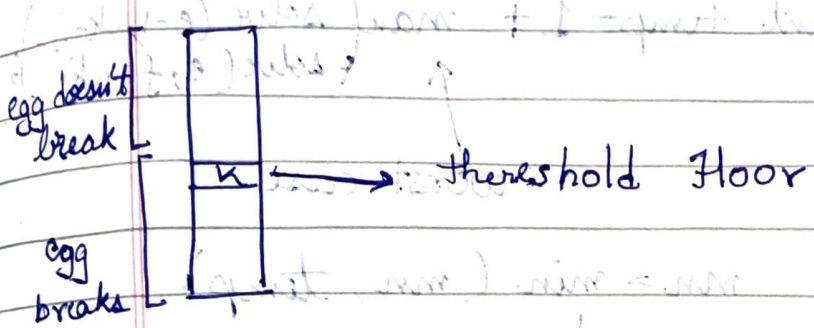
Code

```
bool solve(string a, string b) {
  if (a.compare(b) == 0)
      return true;

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

  string key = a;
  key.push_back('_');
  key.append(b);

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

  return mp[key] = flag;
}
```

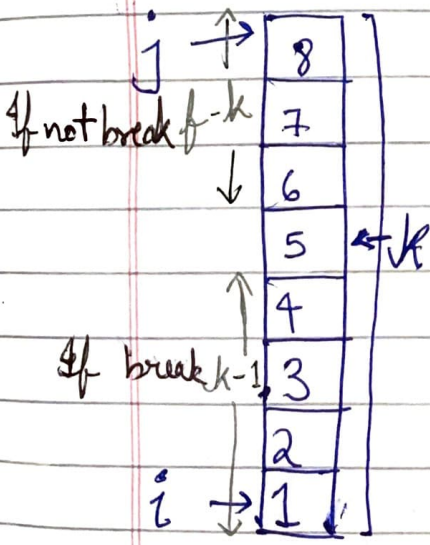
Egg Dropping

I/P | e = 3, f = 5 | O/P | 3

no. of floor



⇒ Min. no. of attempts to find the critical floor




```
for (k=1; k<=j; k++)
```

not valid → Min input Check

e = 0 / 1 → return (f);

f = 0 / 1 → return (f);

solve (e, f)

Break ✓ → solve(e-1, k-1)

Break ✗ → solve(_ , f-k)

Code :

```
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) );
                    ↑ worst case
        mn = min (mn, temp)
             ↳ to select min from all worst case answer
    }
    return min;
```

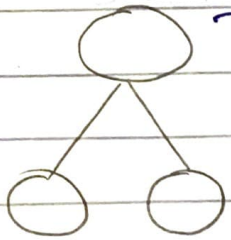
Memorization

```
int static t[11][51];

int solve ( ... ){
    .
    .
    .
if[t[e][f] != -1)
    return t[e][f];
    .
    .
    .
```

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

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

Optimization

⇒ We can also memorize the subproblem



Code:

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

DP on Tree

→ General Syntax
→ How DP can be applied on Tree (Identification)
→ Diameter of a Binary Tree
→ Max. Path Sum from any node to any node
→ Max. Path sum from leaf to leaf
→ Diameter of N-ary Tree

Tree Traversal → O(n)
Node → O(n)
↓
O(n²) ⇒ dp

DP on Tree - General Syntax

```
int fun (← I/P →) {
    Base Condition
    Hypothesis  → gets ans from left & right subtree
    Induction   → Induce the ans from left & right subtree for the root
}
```

Code:

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

Base Condition (80% of cases)

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

Hypothesis (same)

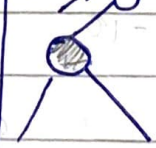
```
int l = solve (root → left, res);
int r = solve (root → right, res);
```

Induction

```
int temp = 1 + max (l + r)
                ↳ calc temp ans.
int ans = max(temp, relation)
                ↳ 1+l+r
res = max (res, ans)
```

```
return temp;
```

temp → ○ Two cases



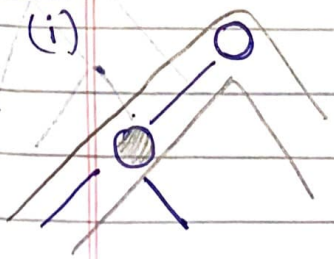
i) temp - passes the value to other node

ii) when the node itself become a part of diameter

Diameter of Binary Tree

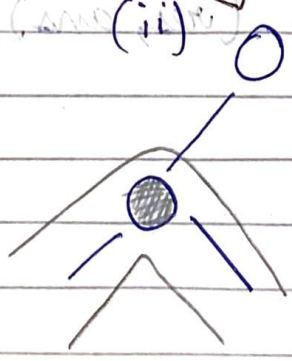
Cases

(i)



$max(l, r) + 1$

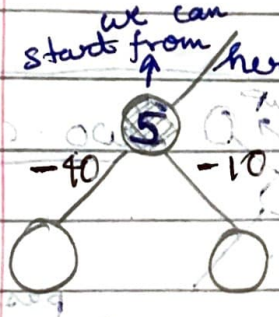
(ii)



$l + r + 1$

```
int solve (Node * root , int &res)
{
    .
    .
    .
    int temp = max(l,r)+1 ;
    int ans = max (temp, 1+l+r);
    res = max (res, ans) ;
}

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

Max Path Sum from Any Node to Any



Case I

(max (l, r) + root→val, root→val)

Case II

l + r + root→val

Code:

```
① int temp = max (max (l,r) + root→val, root→val);
   int ans = max(temp, l+r+root→val);
   res = max (res, ans);
```

Max Path Sum Leaf Node

```
int temp = max(l, r) + root->value
if (!root->left && !root->right)
    temp = max(temp, root->val);
int ans = max(temp, l+r+root->val)
```