Dynamic Programming.

↳ Recursion → 2 calls
↳ then DP

01 Introduction

DP = Enhanced recursion

- Recursion → choice
- Optimal → DP
- ques on optimal.

- Recursive
- ↓
- memoization
- ↓
- top down.

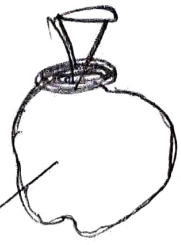
→ Questions on major portion.

1.) 0-1 knapsack
2.) Unbounded knapsack
3.) Fibonacci
4.) LCS
5.) LIS
6.) Kadane's algorithm.
7.) Matrix chain multiplication (MCM)
8.) DP on trees
9.) DP on Grid
10.) Others.

Now 0-1 knapsack problems.

1.) Fractional knapsack
2.) 0-1 knapsack
3.) Unbounded knapsack.

I/P : Wt[] = 1 3 4 5
val[] = 1 4 5 7
W = 7



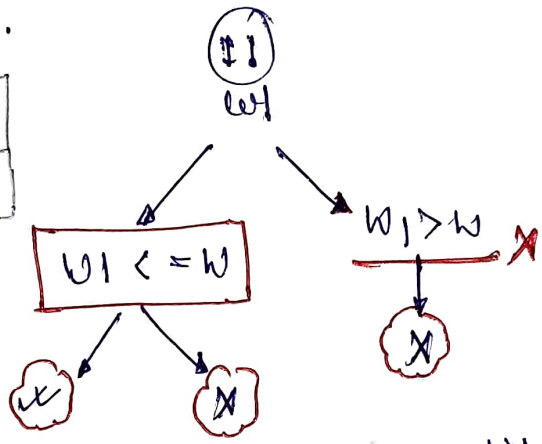
↳ Non item bag me dale to ki max m o/p de. (profit max m).

1st Fractional knapsack → sb cheez ko kaise ko do ya jitna bcha hua hai utna wt dalo → Profit max kro.

2nd Unbounded knapsack → yaha pe multiple occurences dal skte hai.

→ Recursive approach.

wt[] :	1	9	4	8
val[] :	1	4	5	7
W : 7				



int knapSack (int wt[], int val[], int W, int n) {.

Base cond n

choice diagram

}

→ Base cond n

i) think of smallest valid input.

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

→ choice diagram.

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);

Code

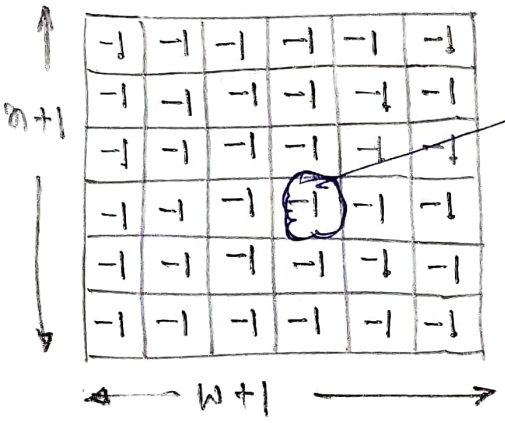
int knap (int wt[], int val[], int w, int n) {.
if (n == 0 || w == 0)
return 0;
if (wt[n-1] <= w)
return max {(val[n-1] + knapsack (wt, val, w-wt[n-1], n-1)), knapsack (wt, val, w, n-1))};
return knapsack (wt, val, w, n-1);
}

1. knapsack memorization.

```
int t[n+1][w+1]
```

→ constraints.

- n <= 100
- w <= 1000



ans store karana hai boxes me.

- isme stack ka size bdhta hai baki space more v theek hai
- time same rehta hai tabulation ke.

```
int t[102][1002]
```

sare ko -1 dena hai.

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

1. knapsack top down approach.

- Yaha pe humlog recursive call ko hi omit kr do (q ki usi usi case me stack overflow se bachata hai)

Recursive
→ memorization
tabulation.

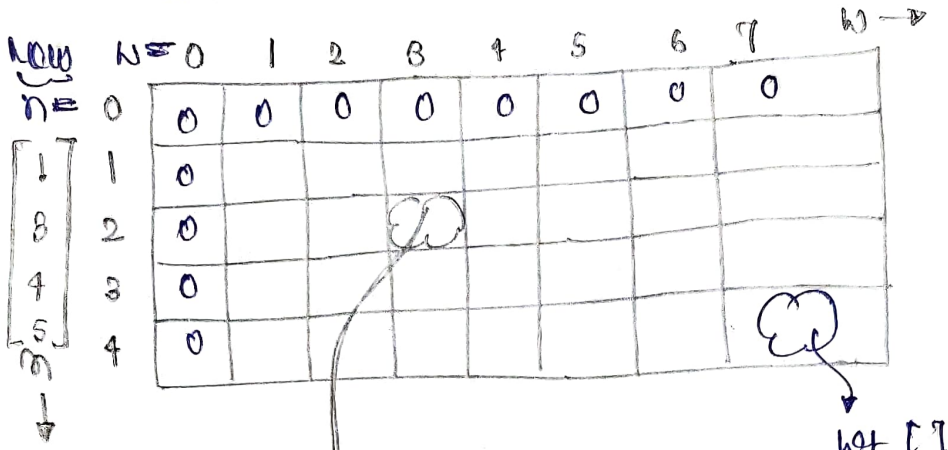
- Put (Base cond n) and initialised dp. matrix.
  → initialization.

Eg. wt[] = [1 3 4 5]

val[] = [1 4 5 7]

W = 7

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



wt[] = 1 3

val[] = 1 4

W = 3.

SUBPROBLEMS.

wt[] = [1 3 4 5]

val[] = [1 4 5 7]

W = 7.

- Yehi final answer hai hamara.

- Base cond^n & dp recursive code.

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

Iska loop kaise lagaoge

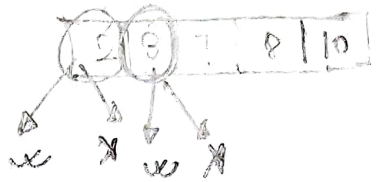
```
for (int i=1; i<n+1; i++) {
    for (int j=1; j<W+1; j++) {
```

→ Subset sum problem.

arr[] = [2 3 7 8 10]

sum = 11 — Yes — {3, 8}

sum = 14 — No



Subset = { }

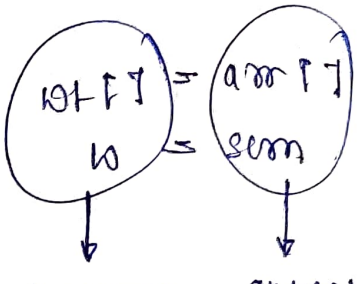
t[n+1][sum+1]

t[6][12].

Base case (initialisation).

	0	1	2	3	4	5	6	7	8	9	10	11
0	T	F	F	F	F	F	F	F	F	F	F	F
1	T											
2	T											
3	T											
4	T											
5	T											

n ↓ W →



Isko for loop me daal do.

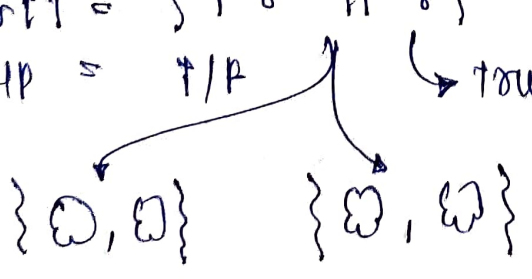
```
if (arr[i-1] <= j)
    t[i][j] = t[i-1][j] || t[i-1][j-arr[i-1]];
else
    t[i][j] = t[i-1][j];
```

→ Equal sum partition. → sum = 1 + 5 + 11 + 5 = 22

arr[] = {1 5 11 5}

op = T/F → true.

req_sum = sum/2; = 11



⇒ sum = S_1 ⇒ sum = S_2

If $S_1 = S_2$ then yes else then NO.

- subset sum chalao jisko sum = 11 ho agar aaya toh done. Yes
- If sum = +ve (odd) not true ever.

→ Count of a subset with a given sum.

arr[] = 2 3 5 6 8 10

sum = 10.

{2 3 5} Count = 3 → o/p

{2 8}

{10}

t[7][11]

t[m+1][sum+1].

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

Base case

POR loop me dal do.

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

→ Minimum subset sum difference.

arr[] = [1 6 11 5]

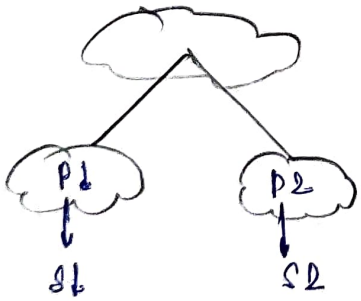
o/p = 1

↳ Eg {1+6+5} - {11}

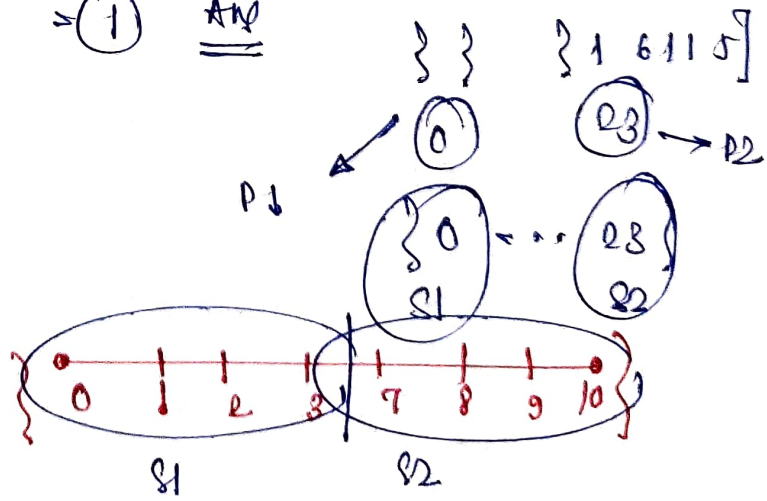
12 - 11

→ (1) Ans

→ So, we find range where the difference lie.



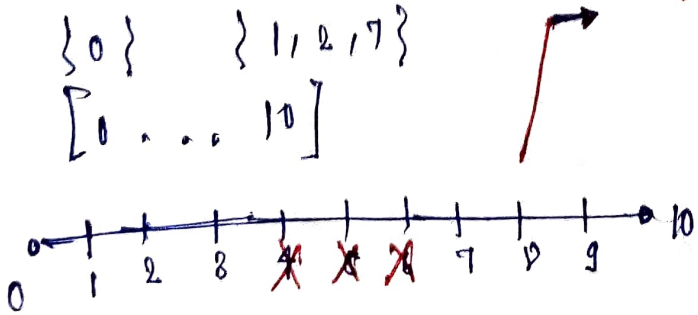
abs(S1 - S2) = min ∞



Eg. arr[] = {1, 2, 7}.

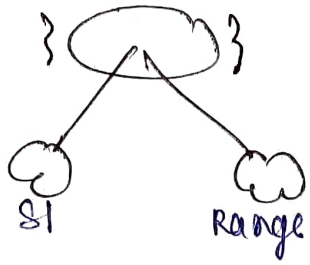
{0} {1, 2, 7}

[0 ... 10]

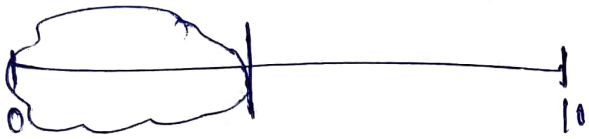


- S1 mil gya to S2 automatic mil jayega.

Idea



So, Range - S1 - S1 ≠ minimize
(Range - 2S1) min.



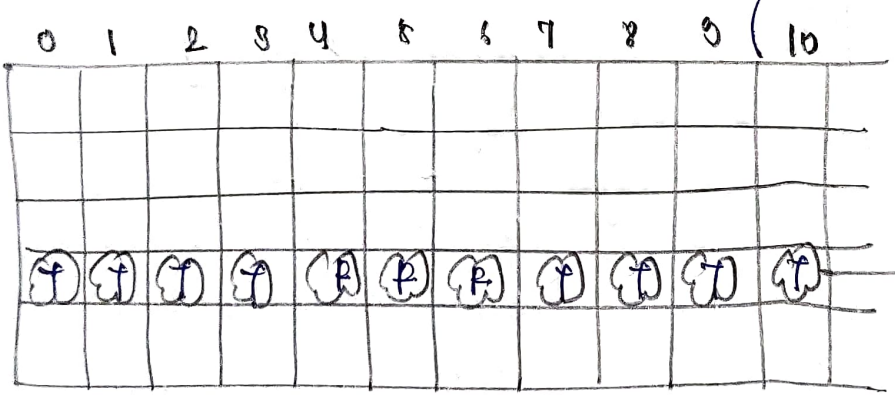
S1 Range - S1

Hence, Range - 2S1 ≈ min

Range - 2S1

S1 = 1 | 10 - 2 = 8
S2 = 2 | 10 - 4 = 6
S1 = 3 | 10 - 6 = 4

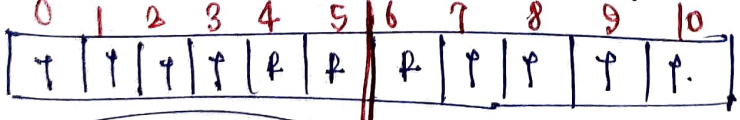
min = 4 Ans



Yeh code subset sum ka code hai.

Yeh wala row ye dega ki array ka size 8 hai to sum [0 — 10] me kon kon possible hai

Now ab loop kaise tak chalayenge,



Reg value of S1 → 0 | 1 | 2 | 3

Range = 10

Si wali value lega jisme ye row me true hoga.

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

→ Count the number of subset with given difference.

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

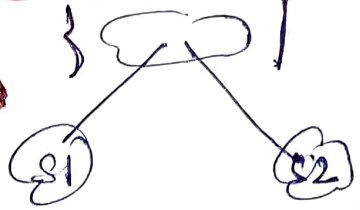
diff = 1

O/p = 3.

Eg. {3,1} {2,1}

{1,1,2} {3}

LOGIC



S1 - S2 = diff

So, S1 - S2 = D

S1 + S2 = S

2S1 = D + S.

$S1 = \frac{D+S}{2} = \frac{1+7}{2} = 4$

known

Now, S1 = 4

→ Target sum.

arr[] = [1,1,2,3]

sum = 1

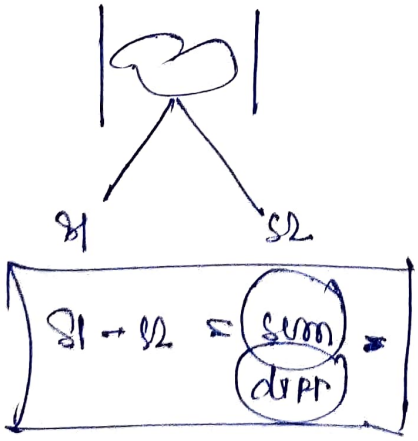
O/p = 3.

count.

1st	+1	-1	-2	+3 = 1
2nd	-1	+1	-2	+3 = 1
3rd	+1	+1	+2	+3 = 1

- Simply do subset me divide kr do. Jiska diff given ka sum ke barabar ho.

Simple



- Same qus as above qus

- Subset sum problem where count to find karna hai. Jaha sum = 4 ho.

done & dusted.

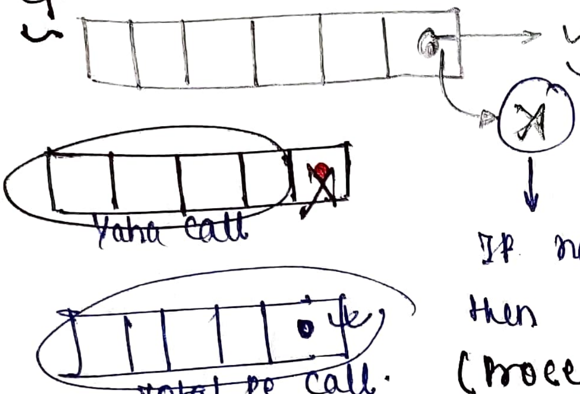
Unbounded knapsack.

↳ multiple occurrences ko allow kera hai.

→ 

01 knapsack.

Eg.

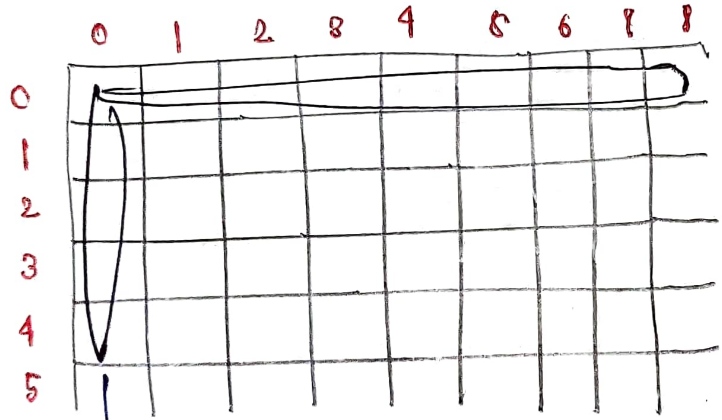


→ ✓ ✓ ✓ ✓ — If true then multiple times it occur. (again, again & again).

✗ → If not then waste. (processed).

Now to recursive code

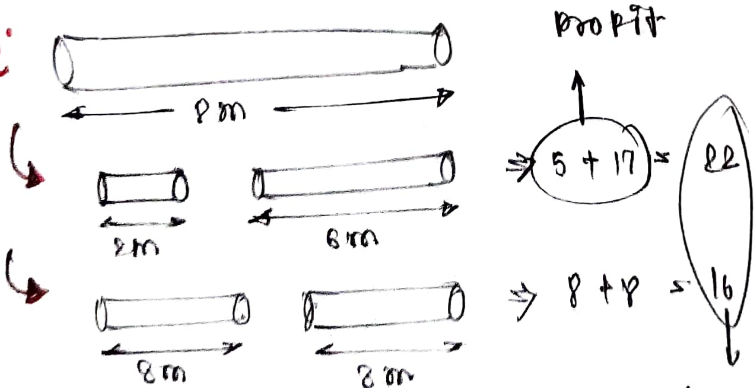
```
if (wt[i-1] <= W) {
   t[i][j] = max { val[i-1] +
                   t[i][j - wt[i-1]],
                   t[i-1][j] };
}
else {
   t[i][j] = t[i-1][j];
}
```



↳ Base case same apply Hoga.

→ Rod cutting problem.

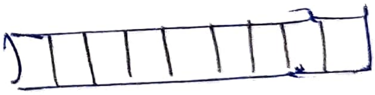
Eg.



Isko maximise karna hai.

length = 1 to N.

not necessary it will be given.

price[]: 

o/p : ( ) → output.

matching	
W	W
price	val
length	wt

Eg.

length [ ] =	1	2	3	4
price [ ] =	5	6	8	8

↳ • same as that of unbounded knapsack. **[CODE]**

→ Coin change (Infinite supply).
→ max^m no. of ways.

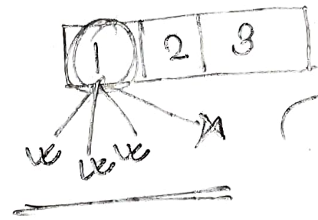
coin [ ] = 1 2 3
sum = 5.

Total ways = (5) / o/p.

Ans
2+3
1+1+1+1+1
1+1+3
1+1+1+2
1+2+2

} n ways

wt [ ] = coin [ ]
W = sum
→ knapsack unbounded



→ Recall subset sum questions.

1 2 3 5 → T = true
sum = 8 → F   (1 2 5)

So, if (arr [i-1] <= j) {
t [i] [j] = t [i-1] [j] +
val [i-1] + t [i-1] [j - arr [i-1]]

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

Now for coin change problem

dp [n+1] [sum+1]
↑
Size of array

sum →

0	1	2	3	4	5
	0	0	0	0	0
↓					
↓					
↓					
↓					

↓ n

- count kuch nhi hai toh (+) kr diye.
  ↳ count / no of ways kuch Jaye toh chote diagram ko sum kete chale Jao.

if (coin [i-1] <= j) {
t [i] [j] = (t [i] [j - coin [i-1]] + t [i-1] [j]);

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

done.

→ Coin change ii. (return no. of coins)

Coins = | 1 | 2 | 3 |

Sum = 5.

o/p = (2)

2 + 3 = 5

→ only 2 coins.

Now t[n+1][w+1]

↓

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

↳ sum.

now | 1 | 2 | 3 | t[4][6]

• Initialization done.

sum = 5

sum →

size	0	1	2	3	4	5
0	MAX	MAX-1	MAX-1	MAX-1	MAX-1	MAX-1
1	0			3		
2	0			•		
3	0					

• at [0,3]

arr[] = [1]

sum = 3

coins needed = 3.

↳ 1+1+1

• at [0,1] = coin[] = 0, sum = 1 } → ∞ - 1

• at [0,0] = coin[] = 1, sum = 0 } → 0

set [] = coins []

set = sum.

• Kbi pe aisa case aaya like

arr[] = 3, sum = 4 } → ∞ - 1. ↳ Impossible case like that.

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

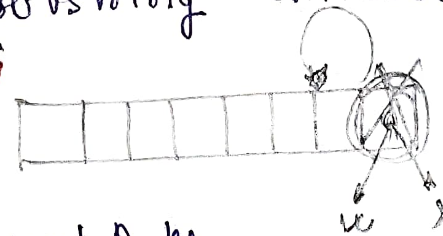
→ **Longest common subsequence.** (subsequence me string ho char char ke ke v le sakte hai).

(substring continuous hoga).

i/p : x : abcdgh
y : abedfgha

o/p : abdgh

length = 5 only length batana hai.



→ logic for choice diagram.

Now **approach.**

- Recursive
1) Base cond^n
2) choice diagram
3) I/p smaller.

Base cond^n ke — Uske valid smallest input.

n = 0
m = 0
Res = 0

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

Now (c-1)
x : abcdgh
y : abedfh

Yaha n-1, and m-1 ke uye chala do.

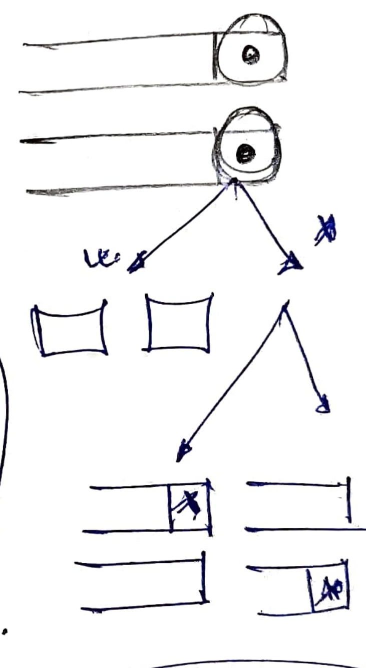
if match.

Now (c-2)
x : abcdgh
y : abedfhr

don't match.

max (n-1 or m pe chalao , n or m-1 pe chalao)

do case bnega yaha.



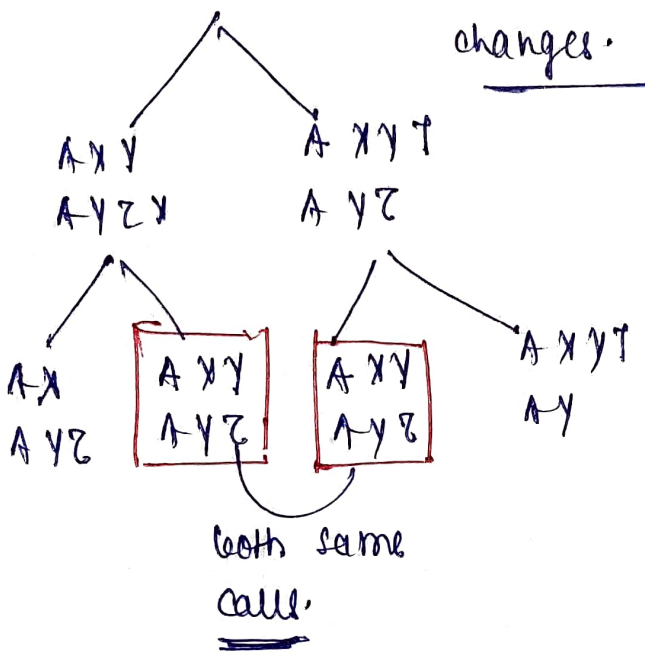
max m nikal lo

Code.

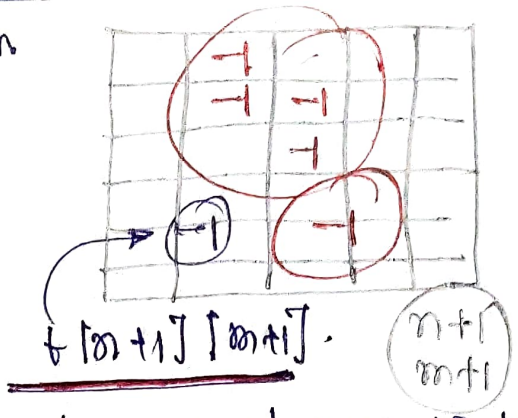
```
int lcs (string x,
         string y) {
  if (n==0 || m==0) return 0;
  if ( x[n-1] == y[m-1] )
  return 1 + lcs( x, y, n-1, m-1);
  else
  return max ((lcs(x, y, n, m-1), lcs(x, y, n-1, m)) ;
```

⟹ Recursive way.

eg



→ table depends on which variable changes.



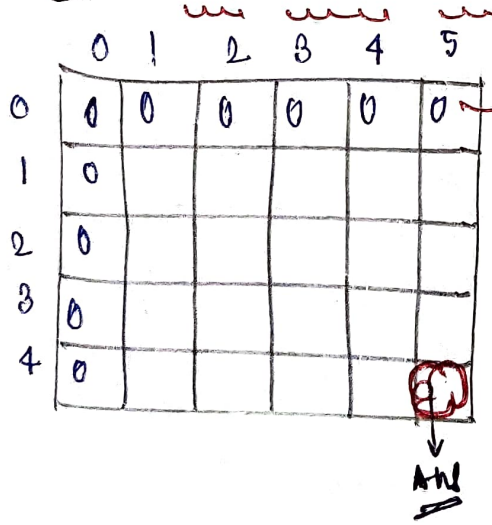
now do normal memorization.

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

both same calls.

So we need memorization approach

Now top down approach.



initialization.

```
for (i=1 ; i<n+1 ; i++) {
    for (j=1 ; j<m+1 ; j++) {
        if (text1[i-1] == text2[j-1])
            dp[i][j] = 1 + dp[i-1][j-1];
        else
            dp[i][j] = max (dp[i-1][j],
                            dp[i][j-1]);
    }
}
return dp[n][m];
```

done

→ Longest common substring

a: abcde

b: abfce

o/p: ab
↳ length = 2

dp [n+1] [m+1].

x : a → same
y : b   as LCS.
longest common.

o/p : int

→ n

0	0	0	0	0	0
0					
0					
0					
0					

↓ m

- Yaha sirf LCS wale code me else wale part ko zero se assign kr dena.

Jaha pe flag hua break kr do.

→ Print longest common subsequence.

a : acbcf → m (5)

b : abcdaf → n (6).

abcf

		0	a 1	b 2	c 3	d 4	a 5	f 6
	0	0	0	0	0	0	0	0
a	1	0	1	1	1	1	1	1
c	2	0	1	1	2	2	2	2
b	3	0	1	2	2	2	2	2
c	4	0	1	2	3	3	3	3
f	5	0	1	2	3	3	3	4

Ye LCS ka code leke table bhar liye.

↳ Last column se start krna hai or us index ke a or b string me check krna hai equal hai toh diagonal pe move jayenge.

(3) ↖ (4)

or agar equal nhi hai toh jo max hoga udhar jayenge or jaha pe equal hote gaya waha usko record kr lenge. us index ka. jaise

f c b a

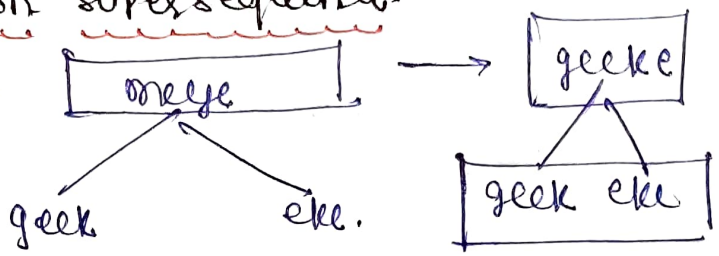
reverse kr do

abcf

done.

→ shortest common supersequence.

a : "geek"
b : "eke"



Subsequence nikale toh mil jaye.

Eg. a : A G G T A B → G T A B
b : G X T X A Y B. 4

A G G T B X A B T X A Y B ⟷ 13 length.

sequence means to order always in order.

A G G X T X A Y B → 9 length very less

→ Has event continuous nhi hona chahye but order me hi hona chahye.

Continuous → not necessary
order → yes always

a : A G G T A B → 6 − 4 = 2
b : G X T X A Y B → 7 − 4 = 3

So, (A + B) = 9 always.

A → LCS length
B → rest

or, 6 + 7 − (4) = 9 always.

Total sum → LCS length.

→ minimum number of insertion and deletion to convert string a to b.

a : heap → LCS = ea = 2
b : pea.

Total = 4 + 3 − 2(2)
= 7 − 4
= (3) operations.

a → b.

heap → pea.

Total = 3.

→ Delete operations for two strings

Eg:
w1 : sea
w2 : eat
lcs = 2

Length n = 3
m = 3
total − lcs
6 − 2(2)
= (4)

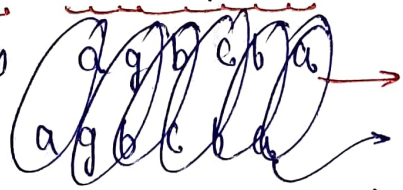
Eg
w1 : leetcode
w2 : etco
lcs = 4

total one = 8 + 4 − 4x2
= (4).

→ longest palindrome subsequence.

s : a g b c b a
o/p : abcba
(5)

Now



a g b c b a
a b c b g a

Now
reverse

BR bar string ko reverse
kro and lcs ka length print
kr do.

→ minimum number of deletions to string make it as palindrome.

i/p : s = "a g b c b a" → abcba

o/p : 1

Now,

a g b c b a
a b c b g a

lcs (5).

Now, length − s
length (l) − lcs
= (1)

↳ same for minimum number of insertion to string make it as palindrome.

→ longest repeating subsequence.

ip : "AABEBCDD"

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

	0	1
A	0	1
A	0	1

i = 3
j = 3

op: ABD
↳ 3 length.

only one condition
in if case

```
if (text1[i-1] == text2[j-1]
    && i!=j) {
}
else {
}
```

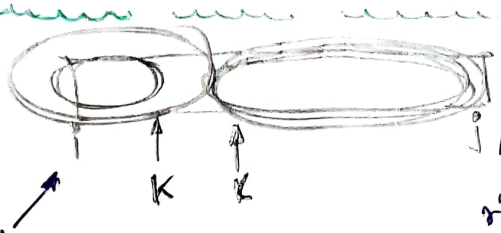
→ sequence pattern matching

a : "AXY"
b : "ADXCPY"

is 'a' a subsequence of B.

kya B me A hai ya nhi.
agar hai toh theek nhi hai
toh false.

Matrix chain multiplication.



left end index

right hand index

→ (i to k) (k+1 to j)

Here k values from Here to Here.

Now qu

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

arr ke size 5 hai to matrix 4 hai.

Now A1 — 40 × 20
A2 — 20 × 30
A3 — 30 × 10
A4 — 10 × 30

So, Ai → A[i-1] × A[i].

0	1	2	3	4
40	20	30	10	30
	i			j

↳ Base conditions

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

```
int solve (int arr[], int i,
           int j) {
  if (i > j)
  return 0;
  for (int k = i ; k < j ; k++) {
    // calculate temp ans
    tempans = solve (arr, i, k)
            + solve (arr, k+1, j);
    ans = min (tempans);
  }
  return ans;
```

Eg. A → 10 × 30
B → 30 × 5
C → 5 × 60.

⇒ 10 × 30 30 × 5 5 × 60.

10 × 5 5 × 60

10 × 60

So, 10 × 30 × 5 + 10 × 5 × 60
= 1500 + 3000 = 4500

⇒ 10 × 30 30 × 5 5 × 60

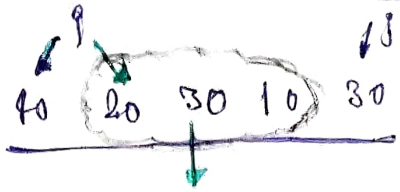
10 × 30 30 × 60

10 × 60.

ans will be 2700

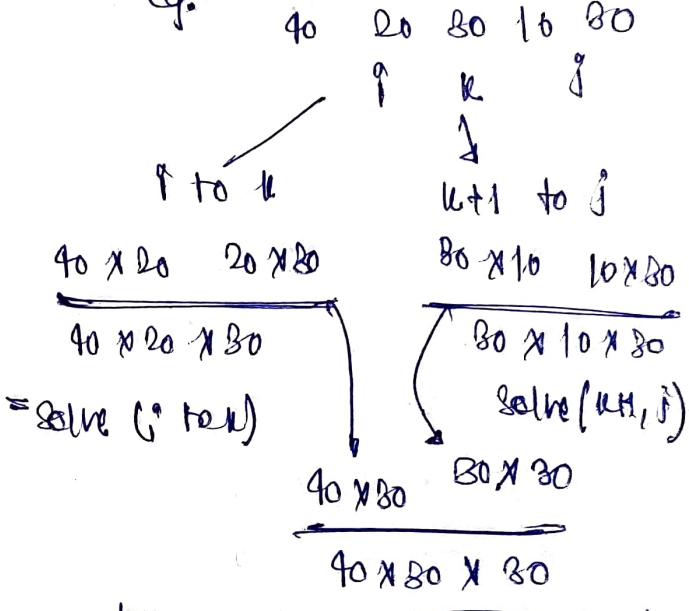
So, 10 × 30 × 60 + 30 × 5 × 60
= 2700.

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



	1st	2nd
k =	i	i+1
k =	j-1	j
	i to k	k+1 to j
	i to k-1	k to j

Eg.



Final recursive code.

Note: dp[100][100].

↳ (-1).

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

arr[i-1] x arr[k] x arr[j]

↳ Extra cost.

↳ memoization code (Bottom up).

Palindrome Partitioning.

Eg. nitin

n | iti | n

2 Partition

total 3 Palindrome. ✗

only one whole is Palindrome

no cuts required

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

int mem (string ts, int i, int j, vector<int> dp){
    if (i >= j || ispalindrome (s, i, j))
        return 0;
    if (dp[i] != -1){
        return dp[i];
    }
    int mn = INT_MAX;
    for (int k=i; k=j-1; k++){
        if (ispalindrome (s, i, k)){
            int temp = mem (s, k+1, j, dp) + 1;
            mn = min (mn, temp);
        }
    }
    return dp[i] = mn;
}

int minCut (string s){
    int n = s.length();
    vector<int> dp (n+1, -1);
    return mem (s, 0, n-1, dp);
}
```

→ most optimised Code

→ **Evaluate expressions to true / Boolean Parenthesization.**

string → T / F / & / | / ^
(T / F: two symbol; & / | / ^: characters.)

eg T | F & T ^ F
(↑ k: operator; expressions on either side)

Now (T | F & T) ^ (F)
(i … j; k points to the operator ^)

(i to k−1) → Exp 1 | operator (k) | (k+1 to j) → Exp 2

for XOR: T ^ F = T, F ^ T = T

So, LT * RF + LF * RT

• Yaha pe Func ^ me true or false v change hoga.

( ) XOR ( )
left → lT, lF
right → rT, rF

Base condition.

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

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