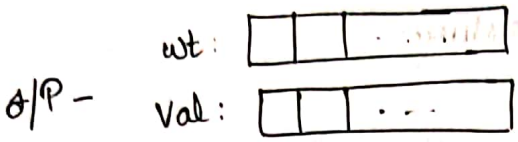
Dynamic Programming

1) 0/1 Recursive Knapsack

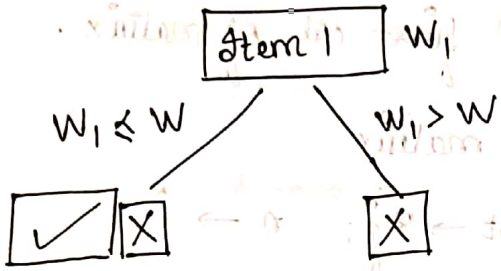


I/P -

O/P → Max Profit

W → wgt. of knapsack

Recursive Code.
↓
Choice Diag.



If wgt > W then we can never include

function

```
int knapsack (int* wt, int * val, int w, int n).
    if (n == 0 || w == 0)          // no items. or W knap = 0.
        return 0.

    if (wt[n-1] ≤ w)                          // not included.
        return max( knapsack(wt, val, w, n-1) ,
                    val[n-1] + knapsack (wt, val, w - wt[n-1], n-1)
                    included + price added.

    if (wt[n-1] > w)      // do not include.
        return knapsack(wt, val, w, n-1).
```

So if included ⇒ val[n-1] + knapsack(wt, val, w - wt[n-1], n-1)

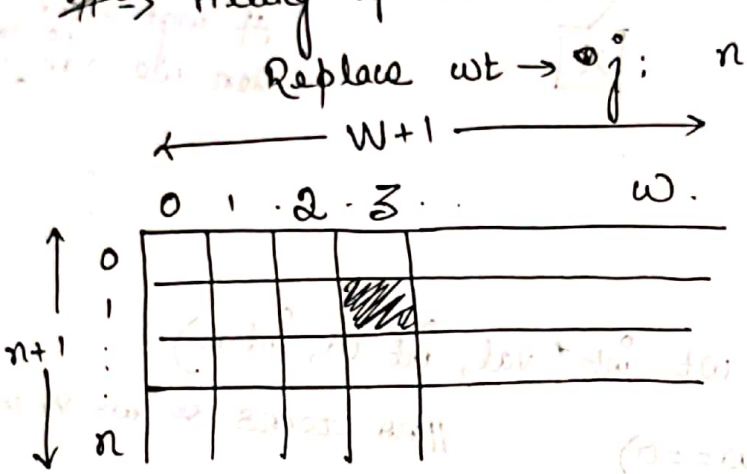
Check Base condit^n.

→ check for the smallest valid i/p.
the result → return

#Top. Down Approach

#=> Base condition turns into initialization of first row and first col. of matrix.

#=> filling up the matrix.



dry random cell is a sub-problem of the given ~~approach~~ problem.

For example → for the highlighted cell.
the sub-problem is

if wgt of the knapsack is 3
and
no. of given items = 1.

then the cell stores the max profit for that sub-problem.

0/1 Knapsack

```
int knapsack(int* wt, int* val, int W, int n)
    int 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)          // Base condition.
                dp[i][j]=0
        }
    }
    // Choice Diagram
    for(int i=1; i<n+1; i++){
        for(int j=1; j<w+1; j++){
            if(wt[i-1]<=j).
                dp[i][j]= max((val[i-1]+dp[i-1][j-wt[i-1]]),
                              dp[i-1][j])
                                  // included
                                  // not included
            else
                dp[i][j]= dp[i-1][j].
        }
    }
    return dp[n][w].
```

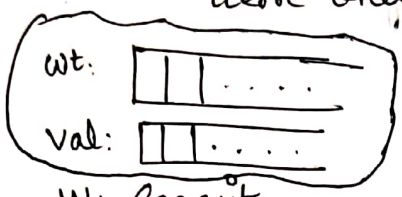
↳ for the given prob.
Max profit.

Variations

How to identify 0/1 knapsack

→ you will have a choice.

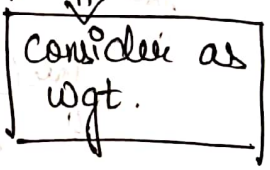
→ You can ~~choice~~ choose an item only once. (if you choose)

I/P ⇒ 

W: Capacity

⇒ Item array with 2 properties

[may be replaced by a single item array]



1. Subset Sum Problem
2. Equal Sum Partition
3. Count of Subset Sum
4. Minimum Subset Sum Difference
5. Target Sum [LeetCode]
6. Count of Subsets ~~for~~ with given Difference.

Subset sum problem

arr = 2 3 7 8 10
Sum = 11

Given an array; find out if there is a subset with the given sum

O/P → True / False.

Initialisation:

i \ Sum	0	1	2	3	4 . . .	11
0	T	F	F	F	F . . .	F
1	T					
2	T					
.	.					
5	T					

subsetSum(arr, sum - arr[i-1], n-1) || subsetSum(arr, sum, n-1)

Sum needed 1, 2 . . . 11
Number of elements in array 0 } False

Sum needed 0;
Take empty subset } True

If True or False; use OR.

```
bool subsetSum(int* arr, int sum, int N){
    bool dp[N+1][sum+1];
    for(int i=0; i<N+1; i++){
        for(int j=0; j<sum+1; j++){
            if(i==0)
                dp[i][j] = false;
            if(j==0)
                dp[i][j] = true;
        }
    }
    for(int i=1; i<N+1; i++){
        for(int j=1; j<sum+1; j++){
            if(arr[i-1] <= j)   // If true
                dp[i][j] = dp[i-1][j-arr[i-1]] || dp[i-1][j];
            else
                dp[i][j] = dp[i-1][j];
        }
    }
    return dp[N][sum];
}
```

#Count Subsets with a given sum

Simple extension of the subset sum problem.

Count subsets;
So return type = int
Instead of "||" use "+" that counts subsets.

Initialisation remains same.

```
int countSubsetSum(int *arr, int N, int sum) :-
    if (arr[i-1] <= j).          only change.
        dp[i][j] = dp[i-1][j - arr[i-1]] + dp[i-1][j].
    else
        dp[i][j] = dp[i-1][j].

return dp[n][sum].
```

Equal Sum Partition

Given an array of size N, check if it can be partitioned into 2 subsets such that sum of elements are same.

O/P → True / False

— x —

Total Sum = S.

Partition into 2 subsets → S_1 + S_2

Now ATP; $\#S_1 = \#S_2$.

$\because S_1 + S_2 = S$

$\Rightarrow 2S_1 = S$

$\Rightarrow S_1 = S/2$.

$\therefore$ Total Sum has to be even; as $S/2$ ← integer.

Target Sum for $S_1 = S/2$

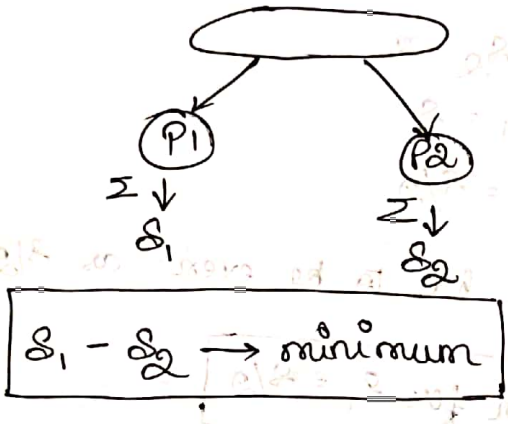
So, we need to find only a ~~sum~~ subset where target sum $= S/2$.

```
bool equalSumPartition(int* arr, int N){
    int sum = 0
    for(int i=0; i<N; i++)
        sum += arr[i];

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

    int target = sum/2;
    else if(sum%2 == 0)
        return subsetSum(arr, target, N);
}
```

Minimum Subset Sum Difference ***

Given an int arr of size N; the task is to divide it into 2 sets S_1 and S_2 such that the abs. diff b/w their sum is minimum.

O/P → Print the minimum difference.

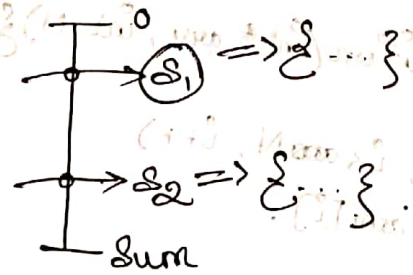


We cannot find the exact values of S_1, S_2.

However we can very easily find the range of S_1, S_2

$S_1, S_2 \in [0, sum]$.

range = sum



Sum = Sum of all elements of the array.

So, we find all subsets possible for the given array b/w [0, sum].

	0	1	2	3	.....	sum
0	T	F	F	F		
1	T					
2						
N	T	F	T	T	.....	T

The last row of the matrix is the sub-problem of the given particular problem.

It shows for the given problem; array of N what all subsets are possible b/w [0, sum].

$S_1 + S_2$ = Range (Sum of all elements in array)

$\Rightarrow S_2 = \underline{\text{Range} - S_1}$

$\Longrightarrow$ So we need to find only S_1.

Restrict range of $S_1 \in [0, \text{Range}/2)$.

$$|S_1 - S_2| = S_2 - S_1$$
$$= \text{Range} - S_1 - S_1$$
$$= \text{Range} - 2S_1.$$

∴ We actually minimize = $\boxed{(\text{Range} - 2S_1)}$

Take out all values b/w [0 and Range/2] that are True and store in a vector.

// Code ***

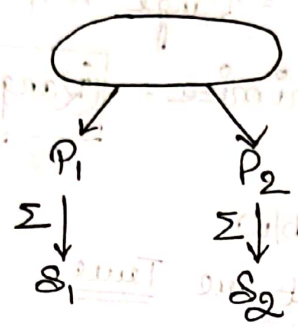
```
int minDifference(int* arr, int N){
    int range = 0;
    for(int i=0; i<N; i++)
        range += arr[i];
    bool dp[N+1][range+1];
    // Write the code for Subset Sum problem  Sum = Range.
    int minDiff = INT_MAX;
    vector<int> vals;
    // Iterate through last row and store trues.  range/2
    for(int j=0; j<=range/2; j++)
        if(dp[n][j] == true)
            vals.push_back(j);
    // Finding MinDiff → Range - 2*S1
    for(int i=0; i<vals.size(); i++)
        minDiff = min(minDiff, range - 2*vals[i]);

    return minDiff;
}
```

Count the number of subsets with a given difference.

Given an int arr with size N; the task is to divide into 2 subsets S such the difference b/w the sums of elements of each subset equals the given difference.

o/p → Count (int)

— x —



$S_1 - S_2$ = difference (given)

$S_1 + S_2$ = sum(arr) = S.

$$S_1 = \frac{\text{sum(arr)} + \text{difference}}{2}$$

So we need to count number of subsets with target/given sum = $\frac{\text{sum(arr)} + \text{difference}}{2}$

target.

// return countSubsetSum (arr, N, $\frac{\text{sum(arr)} + \text{difference}}{2}$)

Target Sum (Leet Code)

Given a list of non-negative int and a target S.
You have 2 symbols + and - for each integer you should choose one from + and -.
Find out how many ways to assign symbol to make sum equal to target S.

O/P → count(int)

— x —

I/P

arr[]: 1, 1, 2, 3.
sum: 1

+1 -1 -2 +3

+1+3 S_1

-2-3 = -(2+3) S_2

$S_1 = \{1, 3\}$
$S_2 = \{2, 1\}$.

$S_1 - S_2 = sum$

∴ becomes exactly similar to previous question.

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

*** Catch

① If zeroes (0) are present we can assign both +/- with it effectively increasing count
So while returning multiple $2^{cnt\ of\ zeroes}$.

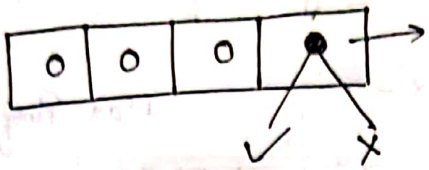
return pow(2, cnt) * dp[n][S_1].

② Also check the condition (target + sum(arr)) % 2 == 0.
If not divisible there cannot be any subset possible

return 0;

Unbounded Knapsack

Recursive approach :-



Once you include - you can include or reject it next time.
If you do not include (i.e, reject) - then you will not include it next time.
↳ (already processed - will not come back)

Initialisation

	0	1	2	3	...	W
0	0	0	0	0	...	
1	0					
2	0					
⋮						
n						

Changes -

only change.

passing n

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

Recursive

```
if (wt[n-1] <= W)                    if you choose (pass n)
    return max(val[i-1] + knapsack(wt, val, W - wt[n-1], n),
               knapsack(wt, val, W, n-1))

else
    return knapsack(wt, val, W, n-1)
```

Rod Cutting Problem

Given a rod of length n inches and an array of prices that contains prices of all pieces less than n. Determine the maximum value obtained by cutting the rod and selling pcs.

—— × ——

Max Profit = 22

2 + 6

length -	1	2	3	4	5	6	7	8
price -	1	5	8	9	10	17	17	20

N: - 8.

This is exactly similar to Unbounded Knapsack.

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

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

else

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

small pcs available → N

dp[n+1][(length of rod) + 1].

length = N in most problems.

n = number of smaller pcs.

If some problems n < length (constraint on smaller pcs).

0	0 0 . . . .
0	
0	
.	
.	
.	

Coin Change I - number of ways

Given a value Sum, find the number of ways to make change for Sum cents; if we have infinite supply of each of Coin = {Coin$_1$, Coin$_2$, ... Coin$_M$}.

— x —

I/P:

Coin: | 1 | 2 | 3 |

Sum: 5

Types of coins (size)
m = 3
↳ Each coin available infinite number of time

Initialisation:

	0	1	2	3	....	Sum.
0	1	0	0	0	0 ...	0
1	1					
2	1					
:						
m	1					

∞ have ∞ coins and have to make sum = 0.
1 way = { }
↑ Empty subset

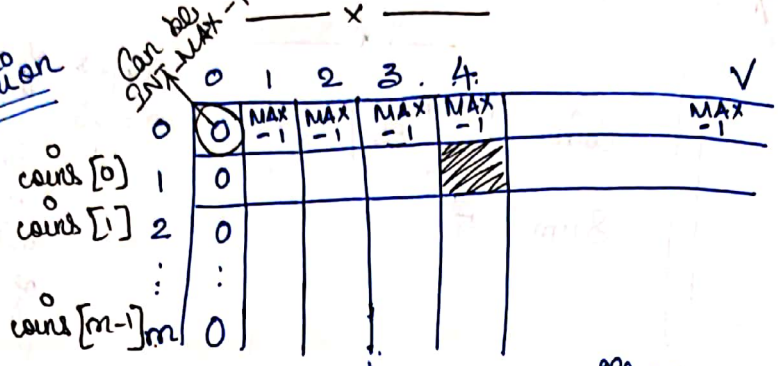
Code

```
long long int count(int* coin, int m, int sum)
long long int dp[m+1][sum+1].
for(int i=0; i<m+1; i++){
  for(int j=0; j<sum+1; j++){
    if(i==0)
      dp[i][j] = 0;
    if(j==0)
      dp[i][j] = 1;
  }}
for(int i=1; i<m+1; i++){
  for(int j=1; j<sum+1; j++){
    if(dp coin[i-1] <= sum)
      dp[i][j] = dp[i-1][j] + dp[i][j-coin[i-1]];
    else
      dp[i][j] = dp[i-1][j].
  }}
return dp[m][n]
}
```

Minimum Number of Coins - Coin Change II

Given a value V and array coins[] of size M, the task is to make change for V ~~coins~~ cents; given you have infinite supply of coins. {coin1, coin2, coin3, ..., coinm}. Find minimum number of coins to make the change. If not possible return -1.

Initialisation



Only question where you initialise (2nd row) [Though redundant]. Not req. / mandatory.

1st row → There are no coins but we need to make a sum. ∴ ∞ number of coins required (INT_MAX - 1) => -1 to avoid overflow

1st column → We have coins however in no way can we make sum = 0; ∴ 0.

*** (Redundant, Not mandatorily Required)

2nd Row → coin → | 3 | 4 | 5 |

For the highlighted cell;
coin available value = 3.
Sum To make = 4.

In ~~Only~~ No way you can make a sum of 4 from 3 value coins. So in that case. INT_MAX - 1.

```
if (j % coins[0] == 0)
    dp[i][j] = j % coins[0].
else
    dp[i][j] = INT_MAX - 1.
```

Code

```
int minCoins(int* coins, int m, int V){
    int dp[m+1][V+1];
    for(int i=0; i<m+1; i++){
        for(int j=0; j<V+1; j++){
            if(i==0)
                dp[i][j] = INT_MAX-1
            if(j==0)
                dp[i][j] = 0
        }
    }
    // 2nd Row (Not Mandatory)
    // for(int j=1; j<V+1; j++){
    //     if(j%coins[0]==0)
    //         dp[i][j] = j%coins[0];
    //     else
    //         dp[i][j] = INT_MAX-1;
    // }

    for(int i=1; i<m+1; i++){
        for(int j=1; j<V+1; j++){
            if(coins[i-1]<=j)
                dp[i][j] = min(dp[i-1][j],
                               1+dp[i][j-coins[i-1]]);
            else
                dp[i][j] = dp[i-1][j];
        }
    }
    int res = dp[m][V];
    if(res >= INT_MAX-1)
        return -1;
    else
        return res;
}
```

Longest Common Subsequence

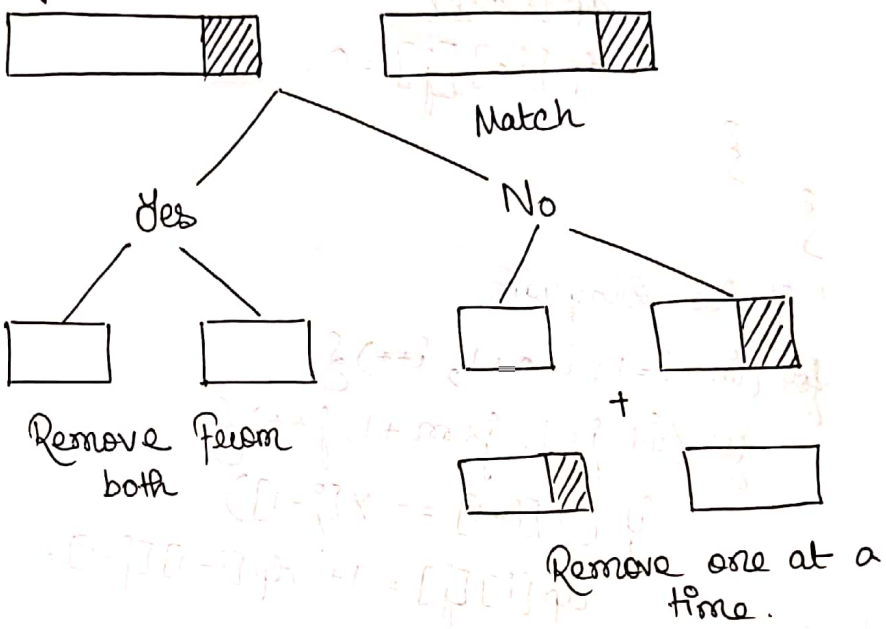
Given 2 sequences, find the length of the longest subsequence present in both of them. A subsequence is in the same relative order. May not be contiguous.

— x —

X: a b c d a f
Y: a c b c f

Output : 4 (length)
abcf (sequence)

Choice Diagram



Recursive

```
int LCS (String X, string Y, int m, int n) {
    if (n == 0 || m == 0)
        return 0;
    // Choice Diagram
    if (X[n-1] == Y[m-1])
        return (1 + LCS(X, Y, m-1, n-1))   // Check from remaining of both strings.
    else if (X[n-1] != Y[m-1])
        return max (LCS(X, Y, m-1, n), LCS(X, Y, m, n-1))
                                            // One at a time.
}
```

m → length Y
n → length X.

Top-Down

x length y length.

```
int LCS(String X, String Y, int n, int m){
    int dp[n+1][m+1]

    //Base
    for(int i=0; i<n+1; i++){
        for(int j=0; j<m+1; j++){
            if (i==0 || j==0)
                dp[i][j] = 0;
        }
    }
    // Choice Diagram
    for (int i=1; i<n+1; i++){
        for(int j=1; j<m+1; j++){
            if (X[i-1] == Y[j-1])
                dp[i][j] = 1+ dp[i-1][j-1];
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[n][m];
}
```

Printing lcs

X: @ c (b) (c) (f)
Y: @ (b) (c) d a (f)

o/p ← "abcf"

// if not equal compare max of above cell and left cell and copy it
// if equal add 1 with the previous diagonal.

algorithm

a) initialise i, j value at the end cell.
b) Compare X[i-1] and Y[j-1].
c) Check if equal
d) Yes; Go to the diagonal before (i--; j--); Push string
e) No;
Compare max b/w left and top cell
and go to it.
i-- / j-- (as req).

Continue steps a-e; until entire strings are traversed.

```
int i = X.length(); int j = Y.length(); // end of dp matrix.
String s = "";
while(i>0 && j>0){
    if(X[i-1] == Y[j-1]){
        //++ s.pushback(X[i-1]);
        i--; j--;
    }
    else{ if(dp[i-1][j] > dp[i][j-1]){
            i--;
        } else
            j--;
    }
} return (Reverse(s));
```

Longest Common Substring

Substrings are in the same relative order and are contiguous (only difference from subsequence)

— x —

a: a b c d e O/P: 2
b: a b f c e.

// only change: if not equal set to 0.
i.e, when continuity is broken set to 0.

```
int LCSubstring (String X, String Y){
    int n = X.length();
    int m = Y.length();
    int dp[n+1][m+1];
    for(int i=0; i<n+1; i++){
        for(int j=0; j<m+1; j++){
            if(i==0 || j==0)
                dp[i][j]=0
        }
    }
    for(int i=1; i<n+1; i++){
        for(int j=1; j<m+1; j++){
            if(X[i-1]==Y[j-1])
                dp[i][j] = 1+dp[i-1][j-1];
            else
                dp[i][j] = 0. // only change.
        }
    }
    return max_from_dp_matrix
}
```

Shortest Common Supersequence (How to print after 5 problems)

Given 2 substrings, a and b the task is to find the length of the shortest string that has both a and b as subseq.

— x —

a = "geek"
b = "eke"

O/P ← 5
geeke

* Order same as given in strings
* Need not be continuous

} Super Sequence.

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

Common → write once. A G G T A B G X T A Y B

Worst Case O/P ← a + b (length of str1 (m), + length of str2 (n))

But in this example we find "GTAB" appears twice in the same relative order.

So, we remove "GTAB" once to get the superseq.

"GTAB" is the LCS b/w the 2 strings.

LCS b/w 2 given strings. So, we remove the length of the LCS one time.

m + n − LCS ← length of short super sequence.

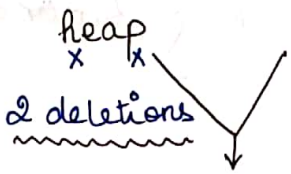
Finally Return

return (m + n − LCS(a, b, m, n))

Minimum number of insertions and deletions to convert a string X to string Y.

X: heap.
Y: pea.

O/P ← Insertions = 1.
Deletions = 2



"ea" (This is the LCS of X & Y).

Number of deletions = X.length() - LCS(X,Y)
Number of insertions = Y.length - LCS(X,Y)

Required many times.

```
// Reversing a string
string reverseString(string x){
    reverse(x.begin(), x.end())
    return x;
}
```

Longest Palindromic Subsequence

Given a str x; find the length of the longest palindromic subsequence present in it.

—x—

x: a g b c b a.

o/p ← 5.

X = a g b c b a

X.reverse = a b c b g a

⑤ "abcba"

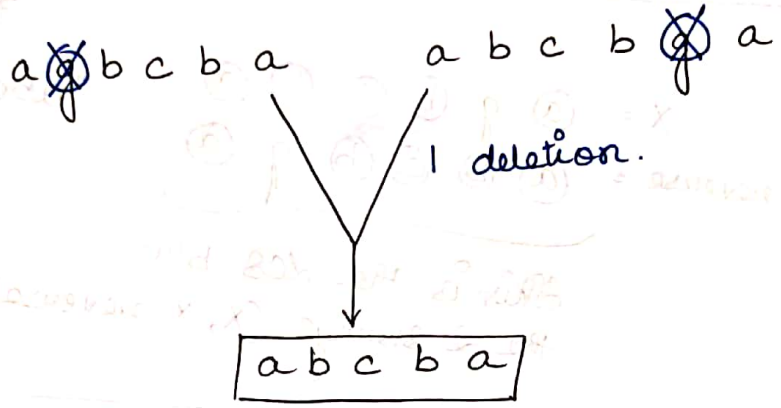
This is the LCS b/w the 2 strings (X, X.reverse())

Longest Palindromic Subsequence (X) = LCS (X, X.reverse())

// Printing the LCS and LPS - Longest Palindromic Subsequence are exactly the same.

Minimum of Deletions in a string / Insertions in a string to make it a palindrome (seperately)

I/P ⇒ X = "a g b c b a" O/P ← 1.



X: a g b c b a

X.reverse: a b c b g a

⟶ LCS = a b c b a.

(b/w X and X.reverse)

ie, LPS

↑↑ Length of LPS ∝ $\frac{1}{\text{No. of Delet}^{n}. \downarrow\downarrow}$ **

Hence we need to find the Longest Palindromic Subsequence.

LPS = LCS (X, X.reverse())

∴ Min. number of Deletions = Min. number of Insertions = X.length − LPS

return (X.length − lengthLPS)