Dynamic Programming

1-D DP
2-D DP
string DP
Grid DP
Game strategy

Recursion + Memoization
&
Bottom Up.

1-D DP

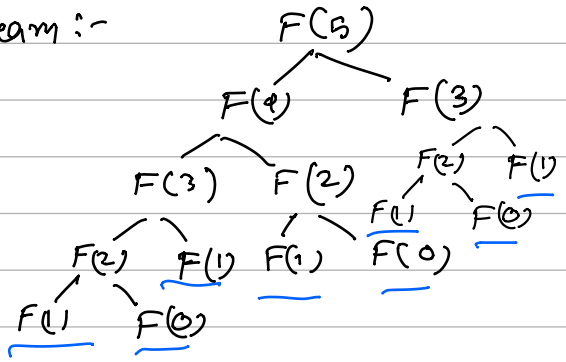
- Fibonacci Number
- Climbing Stairs (N-stairs Problem)
- House Robbers I II (Stickler Thief)
- Maximum Alternative Subsequence Sum

1D DP

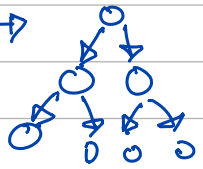
→ Fibonacci Number :- $F(n) = F(n-1) + F(n-2)$

$F(0) = 0 \quad F(1) = 1 \quad F(2) = 1$

① Tree diagram :-

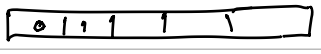


memoization : 1 variable is changing, so 1D DP

→  Vanilla Recursion TC = 2^n

Memoization :- dp : 

```
int F(n){
    if (n<=1) return n.
    if (dp[n] != -1) return dp[n];
    return dp[n] = F(n-1) + F(n-2); }
```

Bottom Up :- dp: 

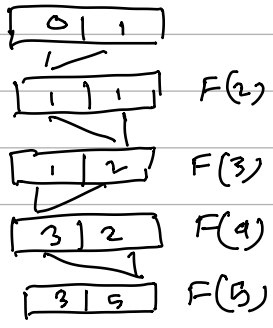
```
for i = 2 to n
    dp[i] = dp[i-1] + dp[i-2] ;
```

→ Time Complexity = O(n) one for loop

Space Complexity = O(n)

Space Optimization :- dp[i] = dp[i-1] + dp[i-2]

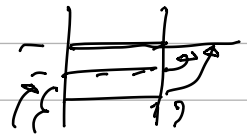
at each step only previous 2 element's value matters. so only 2 variable needed.

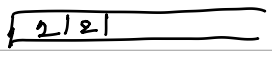


✓ climbing Stairs

n steps stair ; 1 step or 2 step

return how many distinct ways possible

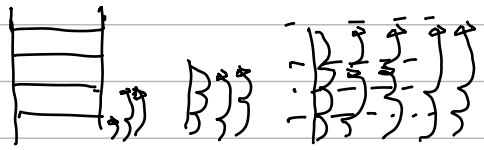




$dp[3] = dp[1] + 2$ ✗

↓

1 step + 2 step.



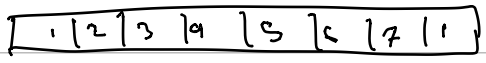
$dp[3] = dp[1] + dp[2]$

↓ then 1 step ↓ then 2 step

n-1 to straight → is a unique way

n-2 to straight → is a unique way.

✓ House Robber :-



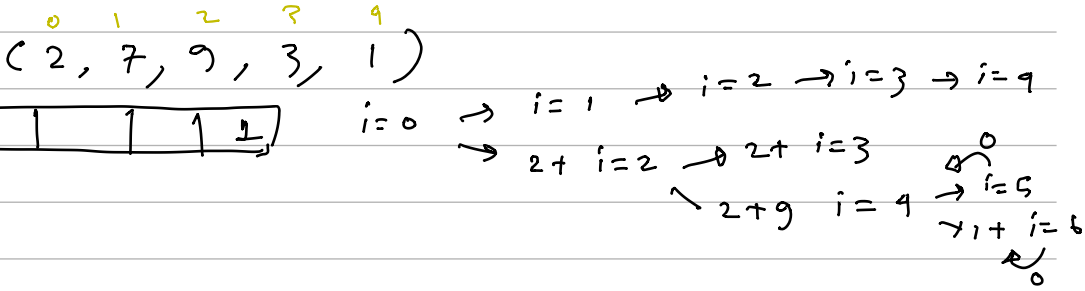
can't robb 2 consecutive house

recursion :- ith house take not take.

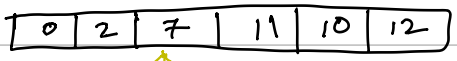
```
f (i, nums)
return max( f (i+1, nums) ,
            nums[i] + f (i+2, nums) )
```

dp[i] = ?

Dry run



Dry run Bottom Up :



max(skip, take)

skip = 2
take = 7+0

skip = 7
take = 11

take = nums[i] + ((i<2) ? 0 : nums[i-2])

↑ This also works

#miscellaneous-STL

can be done in constant space

only previous 2 values are needed.

edge cases ; 2 1 1 2

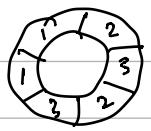
prev prev = 0

prev → stored iteration

temp = max(2, 1+0)

= 0

✓ House Robber 2 :-



[1 2 3 2 3 1] ← these 2 are neighbors

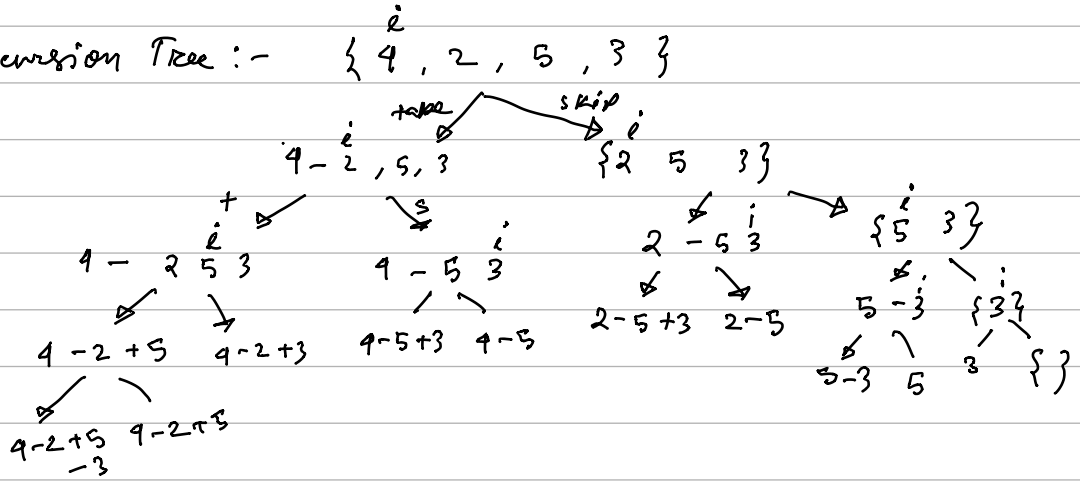
in a circle

HR1 (1 2 3 2 3) (2 3 2 3 1) HR1

max(a, b) = ans of HR2

Alternating Subsequence Sum :-

Recursion Tree :- {4, 2, 5, 3}



when skip sign remains same,
when take sign toggles.

skip = solver(i+1, nums, sign)

take = solver(i+1, nums, sign^2) + (sign ? +n : -n)

Bottom up :-

Array : a_1 a_2 a_3 a_4 a_5 a_6

t[i] has 2 possibility either if is appending to make sequence even or odd

t[i] =

t[i][even] = max(t[i-1][odd] - nums[i], t[i-1][even])

t[i][odd] = max(t[i-1][even] + nums[i], t[i-1][odd])

(0,0)

4	2	5	3
i			
	i		
		i	
			i

evenLen = 0
oddLen = 4

evenLen = 2 (4-2, 0)
oddLen = 4 (0+4, 0)

evenLen = evenPrev or oddPrev - nums[i] = 2
oddLen = oddPrev or evenPrev + nums[i] = 7

evenLen = (2, 7-3) = 2
oddLen = (7, 2+3) = 7

0 0 + ( )
0 0 0 - ( )

* maximum answer till i having the subsequence odd length or even length.

	4	2	5	3	
even	0	(4-2, 0) → 2	(4+5, 2) → 2	(9, 2) → 9	→ these are dp array or
odd	4	(4, 0+2) → 4	(2+5, 4) → 7	(7, 5)	→ just 2 variable.

Longest Increasing Subsequence :-

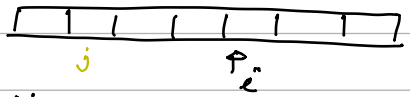
strictly increasing :- {0 1 0 3 2 3}

recursion tree :- 0 1 i 0 3 2 3

take ↙ ↘ skip also note previous element (if possible)

* for memoization dp[i][previous] ; TC = $O(n^2)$

bottom up :-



what's the longest sub seq till i + 1 if nums[i] > nums[j]

another way : Patience Sorting ($O(n \log n)$)

Similar Q :- Maximum length of pair chain :-

{a, b}, {c, d} .... longest chain of pairs such that b < c

(xi < yi given) in any order, so sort the arr.

1 2, 1 3, 1 9, 3 4, 9 5, 6 8.

0 0 1

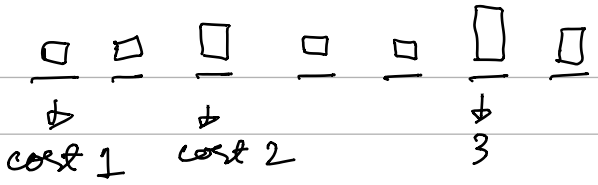
Find the maximum with K comparison :-

#LeetCode/Hard

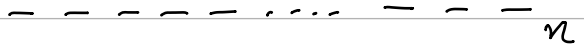
if (arr[i] > mx val): update & cost++;

return number of ways to make the arr for given K, m

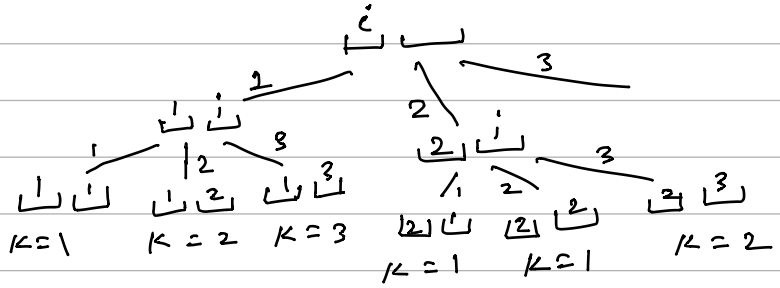
e.g. n = 2 (len of array), m = 3 (available nums 1, 2, 3), K = 1 (cost)



longest increasing (strictly) subsequence



recursion Tree for $n = 2$, $m = 3$, $k = 1$



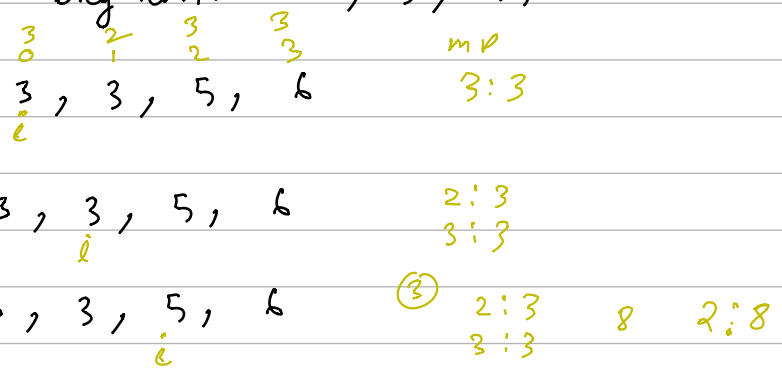
Rec + memo sol^n :-

```
solver(i, cost, maxsofar)
{
    res = 0
    for(j = 1 to m)
        if(j > maxsofar)
        res += solver(j+1, cost+1, maxso)
        else res = solver(j+1, cost, maxS)
    return res, }
```

Maximum Balanced Subsequence Sum :-

#LeetCode Hard

Ory num: 3, 3, 5, 6



LIS in $n \log(n)$ / Patience Sorting;

1	7	8	4	5	6	-1	9
{1}	{1,7}	{1,7,8}	{1,7,8}	{1,7,8}	{1,7,8}	{1,7,8}	{1,7,8,9}
			{1,4}	{1,4,5}	{1,4,5,6}	{1,4,5,6}	{1,4,5,6,9}
						{-1}	{-1,9}

without making new arrays.

1 → [1]
7 → [1 7]
8 → [1 7 8] * Remove the element just bigger than target element.
4 → [1 (4) 8]
5 → [1 4 (5)] * Instead of 1 7 8 make 1 4 5; same stuff but with more opportunity to increase.
6 → [1 4 5 6]
-1 → [-1 4 5 6]
9 → [-1 4 5 6 9]

DP on Strings

- Longest Common Subsequence (LCS)
- print LCS
- Edit Distance
- Shortest Common Supersequence (SCSS)
- Print SCSS
- Longest Palindromic Subsequence.

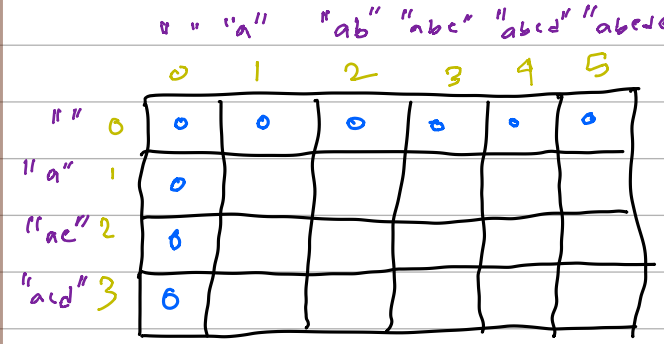
LCS :- recursion : solver (int i, int j, string s_1, s_2)

2 variable changing

bottom up :- s1 = "acd"

s2 = "abcde"

dp (m+1) x (n+1)



"" & "" have no common subsq. hence dp[0][0] = 0

t[2][3] = 1 + t[1][2]

"ac" "abc"

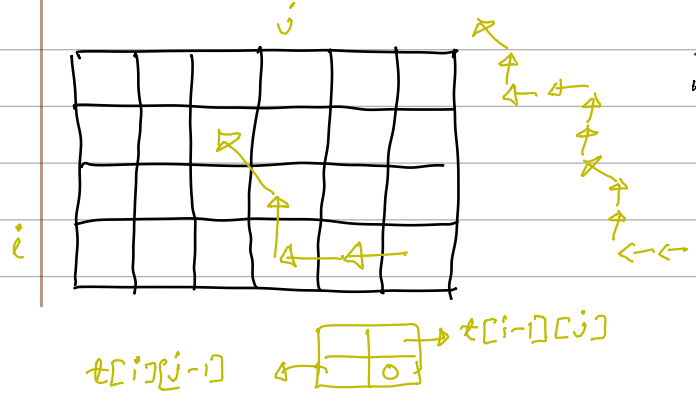
s1[2] == s2[3] so answer to the subproblem is 1 + subP("a", "ab")

t[2][4] = max (t[1][4] , t[2][3])

"ac" "abcd"

s1[2] ≠ s2[4] hence subproblem is

max { sub("ac", "abc") ✓ Both will
sub("a", "abcd") ✓ be computed.



Print :

```
while( i>0 && j>0)
   if(s1[i-1] = s2[j-1])
      { lcs.pushback( s1[i-1]);
        i--
        j-- }
else { if( dp[i-1][j] > dp[i][j-1]) i--;
       else
                                   j--;
```

Shortest Common Supersequence :-

↳ simply $m+n-LCS(S1,S2)$

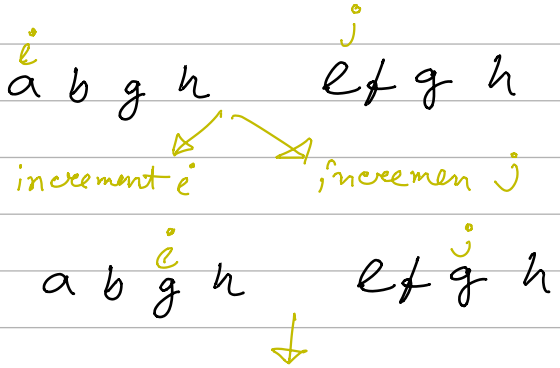
S1 : a b g h

S2 : e f g h

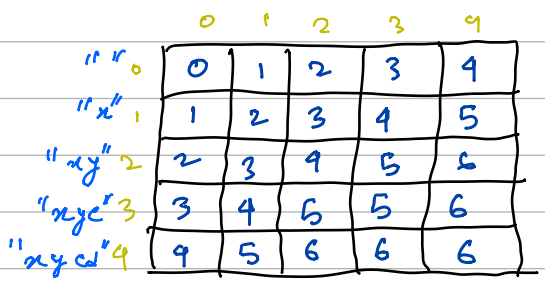
ans : "a b e f g h"

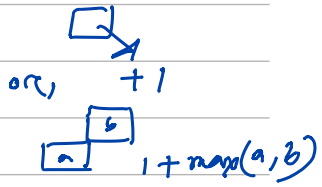
but let's try to solve as a stand alone problem.

Recursion Tree :

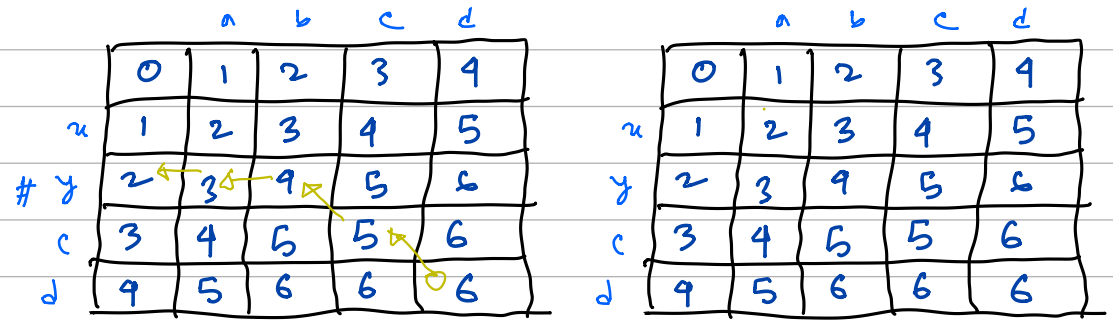


Bottom up :-





print :-



S2 finished so one by one insert S1.

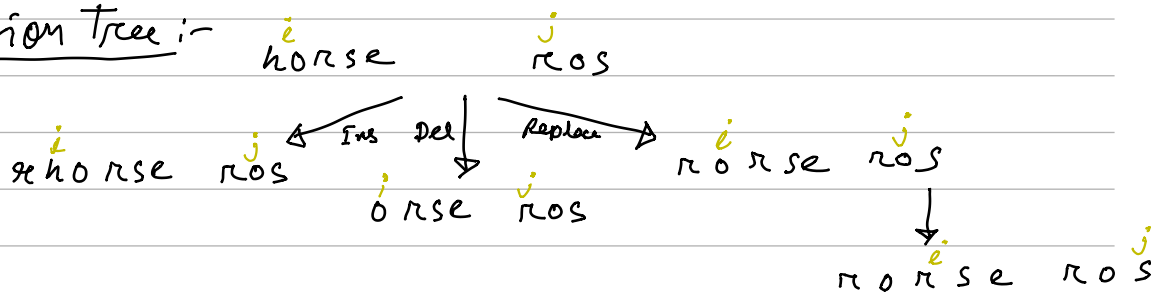
Edit Distance :-

word 1 : "horse"
word 2 : "ros"

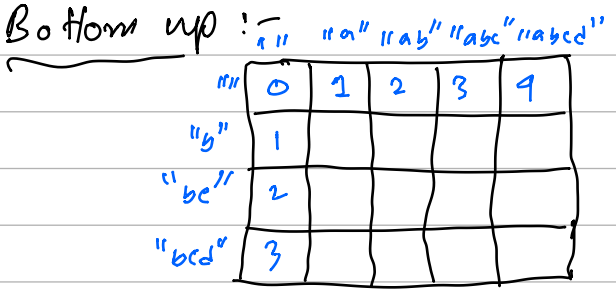
return min moves.
to change word 1 to "ros".

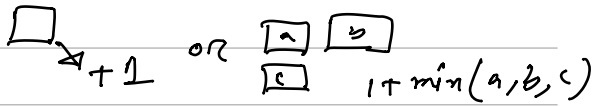
moves;
- ✓ insert a character
- ✓ Delet a character
- ✓ Replace a character.

Recursion Tree :-



Bottom up :-





→ palindromic Substring :-

s : "aaa" ans = 6 [a, a, a, aa, aa, aaa]

brute force : a b a b a d b a

i ⟷ j

check for each (i, j)

optimize : store dp[i][j] true / false to eleminate extra work.

Purely Bottom up :-

s :- "ababab"

① dp[n][n] = T

② if (j - i = 1) dp[i][j] = s[i] == s[j]

③ For filling generic dp[i][j] = dp[i+1][j-1] & (s[i] == s[j])

	a	b	a	b	a	b
a	T					
b		T				
a			T			
b				T		
a					T	
b						T

	a 0	b 1	a 2	b 3	a 4	b 5
a 0	T	F	T	F		
b 1	F	T				
a 2	T		T			
b 3				T		
a 4					T	
b 5						T

0 = dp[0][2] = (a == b) & dp[1][1]

This is same as dp[2][0]

but still for(i=0 to m) for(j=0 to n) Doesn't work

#⭐, #Tricky business

→ need to fill the square like this

* Think in terms of subsequence length

```
for (L = 1 ; L <= n ; L++)
    for( i = 0 ; i+L-1 < n ; i++)
        j = i+L-1 ;
```

	a 0	b 1	a 2	b 3	a 4	b 5
a 0	T	2	3	4	5	6
b 1	2	T	2	3	4	5
a 2	3	2	T	2	3	4
b 3	4	3	2	T	2	3
a 4	5	4	3	2	T	2
b 5	6	5	4	3	2	T

a b a b L = 1

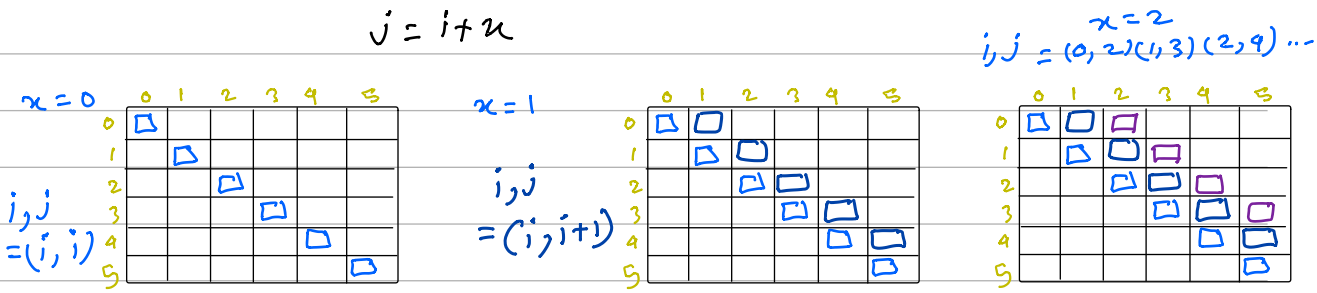
L = 2

L = 3

* need to check when L = 1 True

when L = 2 if s[i] = s[j]

or, for(x=0; x<n; x++)
for(i=0; i+x<n; i++)
j = i+x



✓ another sol^n :-

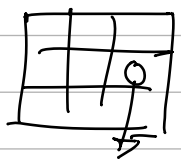
2 types of pallindrome odd len & even len
check all possible centers & spread out.

✓ Longest Pallindromic Substring :-

"a b a b d"
i j

↳ memoize for all (i, j) pair if that is pallindromic or not.

Bottom up :-

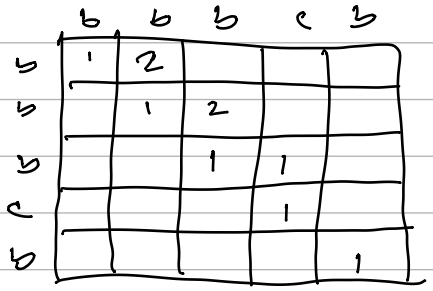


Same as Pallindromic Substring
→ return S.substr(ii, jj)
max (j-i+1) at ii, jj

✓ longest Pallindromic Subsequence :-

" b b b c b"
i j

dp[i][j] longest in between (i,j)
= max((i+1, j), (i, j-1))



Difference b/w LCS :
[i][j-1] ← checking occurs in this fashion → [i+1][j]

1+ dp[i+1][j-1]
or
max((i+1, j), (i, j-1))

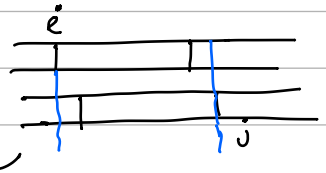
Minimum Insertion to make String Pallindrome:

I/p s: "m b d a b m" ans: 1
m b d a d b m" → Pal.

Soln ⇒ 1: length − longestPalindrome
⇒ n − LIS(S, rev(S))

Approach 2:

m b d a b m
i j



min insertion to make i to j
pallindrome = 1 + min (min insertion to make i to j-1 pallindrome,
" " " " i+1 to j pallindrom)

if s[i] == s[j] then min no of palindrome req = dp[i+1][j-1]

dp[i][j] min char needed to make s[i:j] a pallin

	m_0	b_1	d_2	a_3	b_4	m_5
m_0	0	1	2	3 (mbda)	2 (mbdab)	1 (mbdabm)
b_1		0	1	2	1 (bdab)	2 (bdabm)
d_2			0	1	2	3 (dabm)
a_3				0	1	2
b_4					0	1
m_5						0