"Those who can not remember Past, are condemned to repeat it !"

Dynamic programming

—By Steiver.

Dynamic Programming Playlist | Interview Questions | Recursion | Tabulation | Striver | C++ | Java | DSA | Placements

take U forward

The playlist aims to teach you Dynamic Programming in depth. The focus of the playlist is to cov...**MORE**



PLAY


SHUFFLE

57 videos

Description

The playlist aims to teach you Dynamic Programming in depth. The focus of the playlist is to cover all the concepts, and then follow it up with a lot of problems so that the concepts go into your head and stay there.

The focus is on logic, so no matter in which language you code, you can easily convert it into code, as we will be writing the pseudocode while teaching.

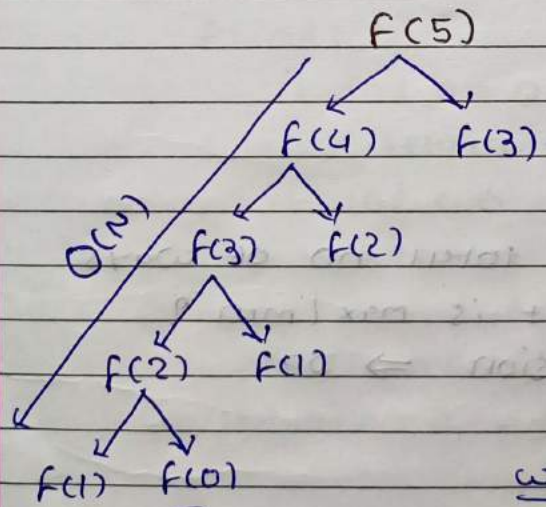
You can also find notes in the description of all the videos so that you can easily revise.

1) Tabulation - Bottom UP

2) memoization - Top down.

Overlapping subproblem :- It is part of our dp problem and need to comput again and again. like for fibonaci(5) we will need fibonaci(2) and then also for fbonacci(6) we will be needing fibonaci(2), so we will compute once and will store it. that is called memoization.

(1) Fibonaci Number :-



with recursion :-
If we use memoization then we will not recalculate some already calculated values so,

TC: O(N)

SC: O(N) + O(N)
Rec. stack
Array of dp.

Iterative method:-

then SC will be O(N) only:
cause there will not be any recursion stack.

this is promethod

```
prev2 = 0
prev1 = 1
for i=2 to i=n
    curr = prev1 + prev2
    prev2 = prev
    prev1 = curr
print (prev1)
```

And SC = O(1).

(*) Short tricks:-

when, count total no of ways
so what is max/min ?
will use Recursion ⇒ DP

How ?

→ Try to represent the problem in terms of index
→ Do all possible stuff on index according to Problem.
→ count all way = sum of all stuff
find min/max = min/max of all stuff

*) Max sum of non-adjacent elements.

Input : {5, 5, 10, 100, 10, 5}
Output : 110

How ?
we are not allowed to choose two adjacent elements
so to get max, we will select {5, 100, 5}
index 2 4 6

Approach 1 :-

try out all possible sub-sequence.

```
f(ind)
    if ind == 0
        return a[ind]
    if ind < 0
        return 0

    pick = a[ind] + f(ind-2)
    not_pick = 0 + f(ind-1)

    return (max(pick, not_pick)
```

} means we haven't pick ind 1 so pick at index 0

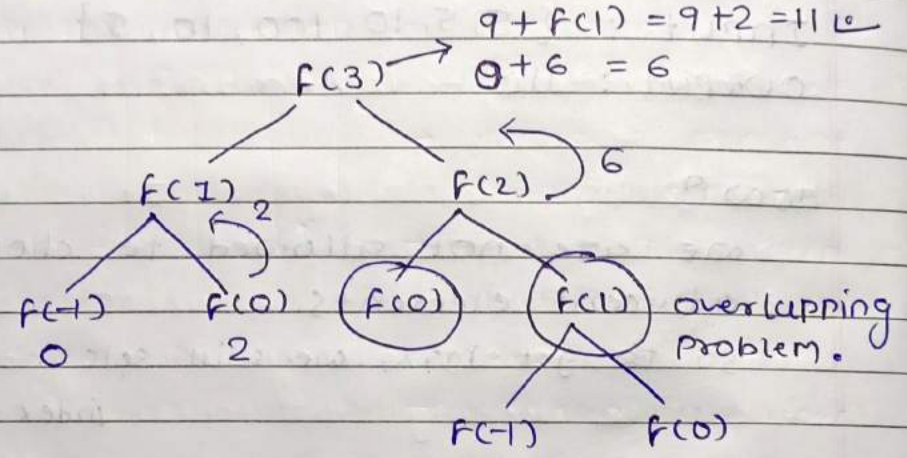
} can't pick adj so ind-2
can pick adj so ind-1

} returning max of it.

→ f(i) says that max sum from index 0 to i if you do not pick or any adjacent

let's take example :-

a = {2, 1, 4, 9}



for f(2)

pick = 4 + f(0) = 4 + 2 = 6 } max → 6

not-pick = 0 + f(1) = 2

so Answer is 11

at f(3) we have two option,

if we pick at index 3

then 9 + f(1) = 9 + 2 = 11

else we do not

then 0 + f(2) = 0 + 6 = 6

and 11 is max here.

so TC :- O(n)

SC :- O(n) + O(n)

It's taking O(n) space for recursive stack let's do tabulation.

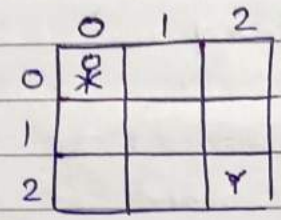
```
solve(a)
    n = a.size
    dp[n]
    dp[0] = a[0]

    for i = 1 to i = n-1
        take = a[i]
        if i > 1
            take = take + dp[i-2]
        not_take = dp[i-1]

        dp[i] = max(take, not_take)

    return dp[n-1]
```

Optimized the best Approach

Here we can see that we just need the last two state (dp[i-1], dp[i-2]) we can store this in 2 variables and can ommit the O(n) space.

*) DP on Grids :-

Total unique path :-



→ You are present standing at (0,0)

→ Find unique path that leads you to (2,2)

→ You can only move to downward or rightward.

How?

1. Represent index in terms of row no and col. no ⇒ (i,j)
2. Explore all path
3. Do action (sum, min, max etc) as per question.

let's solve question now :

f(i,j) ⇒ from cell (i,j) how many path are there possible to reach (0,0)

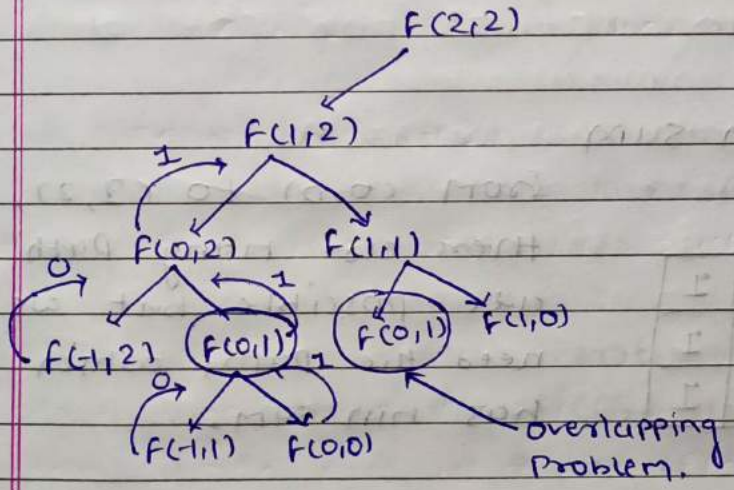
So let's code

```
f(i,j)
    if(i==0, and j==0)
        return 1
    if (i<0 or j<0)
        return 0

    up = f(i-1, j)
    left = f(i, j-1)

    return up + left
```

TC = $O(2^{m \times n})$

SC = O(max path length)

To improve TC we need to find that if there is overlapping problem or not ?



So we have state i and j which will be useful for applying DP.

So we will just add dp[i][j] to reccurion

Memoization to tabulation :

steps:-

1. Declare base case
2. Express all states in loop
3. copy recurrence relation

so

```
for i=0 to i=m-1
    for j=0 to j=n-1
        if i==0 and j==0
            dp[0][0] = 1
        else if i>0 and j>0
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
```

*) Min path sum

1	3	1
1	5	1
4	2	1

from (0,0) to (2,2), there are many path are possible but we need the path, which has min sum.

obviously it's Grid DP as we can see

Note:- Here we are not counting the path instead of we are finding min/max value, so on out of bound index case then will be different logic.

```
solve (i, j, dp, grid)
    if i == 0 and j == 0
        return grid[0][0]
    if i < 0 or j < 0
        return int_max
    if dp[i][j] != -1
        return dp[i][j]

    int left = grid[i][j] + solve(i, j-1, dp, grid)
    int up = grid[i][j] + solve(i-1, j, dp, grid)

    return min(left, up)
```

⇒ tabulation is easy just run two for loops

⇒ To optimize space, we just need to use two array of cols size prev and curr.

*) Triangle Path Problem.

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

given a triangle find min path from top to bottom at ith index, there will be i+1 elements.

you can move down to next row or right diagonally to next row.

that is,
can move to index i or i+1
into next row.

⇒ let's start.

Here grid greedy solution fails because value distribution is not uniform (sorted).

So we will try out all path and find the minimum.

→ so we can start from (0,0)
go to (0+1,0) and (0+1, 0+1)
took minimum
base case will be i = n

```
solve (i, j, tri, dp)
    if (i == n-1) return tri[i][j]

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

    down = tri[i][j] + solve (i+1, j, tri, dp)
    diag = tri[i][j] + solve (i+1, j+1, tri, dp)

    return dp[i][j] = min (down, diag)
```

tabulation.

last row is our base case so will start from that as dp[n-1] = tri[n-1]

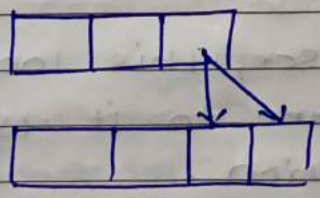
then

```
for i = n-2 to i = 0
    j = i to j = 0
        down = tri[i][j]
               + dp[i+1][j]
        diag = tri[i][j]
               + dp[i+1][j+1]
        dp[i][j] = min(down, diag)
```

and our answer will be at dp[0][0].

Space optimization

If we notice we just need curr row that is for present process.



and pre. row to look of so we can do with just one array and one temp array.

* Maximum path sum :-

```
10  2  3          10 → 7 → 8
 3  7  2    ⇒       = 25 is max
 8  1  5
```

you are at first row, can start from any column.
from a cell you can move
- down
- down left diagonal
- down right diagonal

find out the max path.

⇒ as usual greedy will not work because values are not sorted.

so we need to try out all path and find the max one.

if we found sub problem overlapping then we need dp.

⇒ As we can see, this looks like path sum, we can do it with recursion and memoization.

So we will ~~start~~ with
f(i, j) ~~min~~ max path from (i,j) cell to last row

So base case : i == last row
out of bound case : j < 0 or j >= n.

So,

```
f(i, j, n, grid, dp)
    if j >= n or j < 0 return -10^9
    if i == n return 0

    if dp[i][j] != -1 return dp[i][j]

    down = grid[i][j] + f(i+1, j, n, grid, dp)
    d-left = grid[i][j] + f(i+1, j-1, n, grid, dp)
    d-right = grid[i][j] + f(i+1, j+1, n, grid, dp)

    return dp[i][j] = ~~min~~ max([down, d_left, d_right])
```

Note: Here starting point is not fixed so,

```
for i = 0 to i = cols
    f(0, i, n, grid, dp)
```

*) chocolate pickup (3D DP)

= problem:-

you have R x C grid, having chocolates at each cell, two friend start picking chocolate from (0,0) and (0,C) cell. you want to pick max chocolate, find the max no of chocolates they can pick by reaching to last row.

= conditions:-

they can move to
- just down
- left down
- right down

If both are in same cell then only one will pick the cocolate.

=

Intution:-

As we can see it's path problem grid DP will be solution for this.

we will proceed both friend simutaneously because we need to check that if both are picking from same cell then they are only one allouded to tylee chocolate.

Here for simaltaneously movement

when A friend is moving down, B can move to down, left-down or right down

this is how for 1 movement of A, B can make 3 movement.

because we are not sure that both are making same movement.

but we are sure that both will here same row at a time.

So let's start :-

f(i, j_A, j_B) :- max chocolate can obtain when friend A is in (i, j_A) cell and friend B is in (i, j_B) cell.

base case :- i == n - 1

if j_A == j_B

return grid[i][j_A]

else

return grid[i][j_A] + grid[i][j_B]

So, it will go like this:

```
f(i, jA, jB, r, c, grid, dP)
    if jA < 0 or jB < 0 or
       jA >= c or jB >= c
          return -10^8

    if i == r-1
        if jA == jB
            return grid[i][jA]
        else
            return grid[i][jA]
                   + grid[i][jB]
    if dP[i][jA][jB] != -1 return dP[i][jA][jB]
    int maxi = -10^8

    for j1 = -1 to j1 = 1
        for j2 = -1 to j2 = 1
            if jA == jB
                value = grid[i][jA]
            else
                value = grid[i][jA]
                        + grid[i][jB]

            value += f(i+1, jA+j1, jB+j2, r, c,
                       grid, dP)

            maxi = max(maxi, value)

    return maxi dP[i][jA][jB] = maxi
```

Note:-

Here we can go to,

down : (i,j) ⇒ (i+1, j+0)

left : (i,j) ⇒ (i+1, j-1)

right : (i,j) ⇒ (i+1, j+1)

so we need for loop of -1 to 1.

*) DP on subset / subsequences :-

*) subset sum equal to k :-

you are given an array and integer k, you need to check that is there exists a subset of array, with sum equal to k.

Eg :- {1,2,3,4} k = 4

yes :- {1,3} and {4}

output : true.

let's start :-

First approach come in mind is generate all the subsequences.

we can do using power set or recursion

Here powerset generates all, but we need only one subset so we will follow the recursion.

So we can start like this.

(1) f(ind, target)
(2) Explore all possibility.
    (i) ind is part of subseq.
    (ii) ind is not part of subseq.
(3) return T|F.

Here f(n-1, target) says that in the entire array till n-1 from 0, subset sum target exists or not.

Base case :-

target == 0 return true;
if ind == 0
    return target == arr[0]

we are decreasing target
so
f(0, target) says that till 0 is target sum exist or not.

Exploring possibility:-

- take = false
  if target <= arr[ind]
  take = f(ind-1, target - arr[ind]

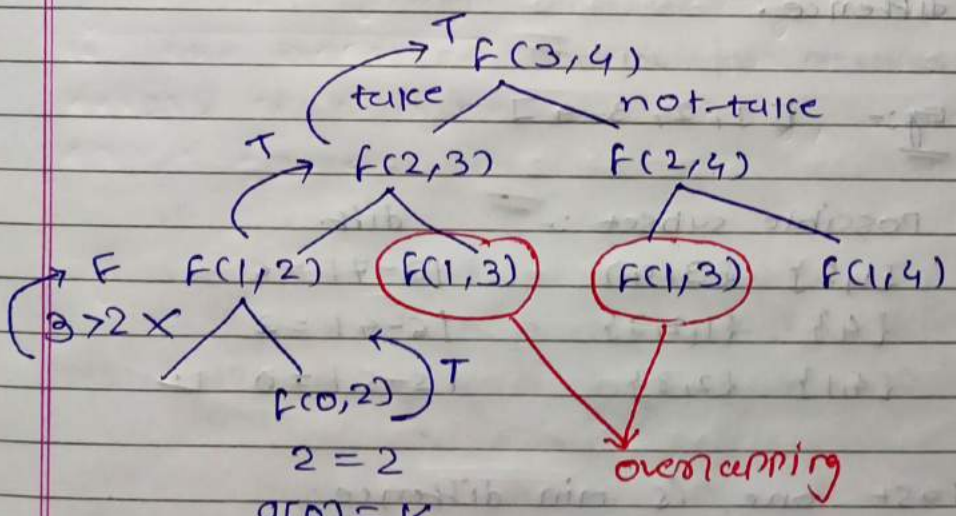
- not_take = f(ind-1, target)

returning:

- return take or not_take

⇒ Now let's draw Rec-tree:-

2,3,1,1
[~~1~~ 2, ~~3~~ 4] K = 4



For Recursion:-

TC : $O(2^n)$ Recursion
SC : O(n) :- stack space

But we can apply memoization
states : ind, target.

so TC :- O(N x target)
SC :- O(N) + O(N x target)

dp[N][target]
=.

(*) Partition a set into two subsets such that difference of subset is minimum.

You just need to find the minimum difference.

Eg :- [1, 2, 3, 4]

Possible subset :- diff.

{1,2}	{3,4}	|3-7| = 4
{4}	{1,2,3}	|6-4| = 2
{4,1}	{2,3}	|5-5| = 0

last one is min difference

so output :- 0.

For this we will have S_1 and S_2 and we want to make $|S_1 - S_2|$ as minimum as possible.

So, we will find all possible S_1 and $S_2 = total - S_1$ and find the min difference.

So we need to have dp[n][total]

here total = $a_1 + a_2 + \dots + a_{n-1}$.

So,

for i = 0 to i = totalSum/2

if dp[n-1][i] == true

ans = min(ans, abs(S_1 - (total - S_1))

Here we take totalSum/2 because for {1,2,3,4}

total sum = 10

So

S_1	0	1	2	3	4	5	6	7	8	9	10
S_2	10	9	8	7	6	5	4	3	2	1	0
$\|S_1 - S_2\|$	10	8	6	4	2	0	2	4	6	8	10

same. So,

Here dp[n-1][i] is that from 0 to n-1 is subset sum = i exist or not ?

Note :-

while writing reccyrance if we told to find count of something then,

i) return 1 for right case
   return 0 for fail case
ii) sum of all possible operation

like,

```
if ind == 0
    if sum == a[0]
        return 1
    return 0

if sum == 0
    return 1

pick = solve(ind-1, sum - a[ind])
notPick = solve(ind-1, sum)

return pick + notPick
```

*) 0/1 knapsack

you are given weight and value of items and knapsack capasity, find the max value can get.

Eg:- wt :- 3 4 5 capacity = 8
value :- 30 50 60

item selected :- 3 5 = 8 < 9
30 60 = 90.

we can try out all possible ways and select the max.

so f(ind, w) :- till index ind what the max value you will get with weight w (remaining)

→ and we will proceed with take and not take,
→ while taking the item, we need to check either item weight is less or equal to bag capacity.
→ we will start from f(n-1, w) so base case:

```
if ind == 0
    if weight[0] <= w
        return value[0]
    return 0;
```

} take if capasity is ok, ∈

Here we will do recursion that will take $O(2^n)$ time

Now, we can optimize it using memoization

our changing state is
- ind
- weight

Recursive DP code is easy
for that:
Time complexity :- $O(n \times w)$
Space complexity :- $O(n \times w) + \underbrace{O(n)}_{\text{stack}}$

So let's do tabulation:-

1. we will have $dp[n][w] = \{0\}$
2. Fill value for base case.
   base case. { for every $w <= capacity$ and $w \geq wt[0]$
   $dp[0][i] = val[0]$
3. do nested loop for states.
4. copy or write recurrence.

So let's code it:-

```
knapsack(n, maxw, weight[], value[])
    dp[n][maxw+1] = {0}

    for i = weight[0] to i = maxw.
        dp[0][i] = value[0]

    for i = 1 to i = n-1
        for w = 0 to w = maxw
            nottake = dp[i-1][w]
            take = -10^8
            if w >= weight[i]
                take = value[i] + dp[i-1]
                                  [w - weight[i]]
            dp[i][w] = max(nottake, take)

    return dp[n-1][maxw]
```

So If we talk about

time complexity :. O(n x maxw)

space complexity :. O(n x maxw)

we can ommit the space complexity to O(maxw) by using 1D DP, 2 array, prev and curr.

*) coin change:-

We are given array of some distinct coins, every coin has infinite supply and given target, we need to use minimum no of coin as possible and print that coin number count.

Eg:- {1, 2, 3} target = {7}

Ans:- {3, 3, 1} ⇒ 3

Here greedy works for many of case but fails sometime.

Eg: {9, 5, 6, 1} target = 11

greedy = {9, 1, 1} ⇒ 3
but right is 2 ⇒ {5, 6}.
=.

So we will try out all possible path and choose the minimum one.

So, f(ind, target) :- How many coins required to form 'target' using '0 to ind' coins.

So possible cases:-

take = Int_max
if coins[ind] <= target
    take = 1 + f(ind, target - coins[ind])

not take = f(ind-1, T)

returning required value:-

    return min(take, nottake)

base case:-
    we will start with f(n-1, target)
    so
    if ind == 0
        if (target % coins[ind] == 0)
            return target / coins[ind]
        else
            return Int_max

    It is like we have coin[0] = 4
    and target = 12
        then we need 12/4 = 3 coin
        this possible because we can make
        12 using 4' coin.

    but if target = 11
        then we can not so return Int_Max

let's solve it:-

```
solve ( coins, x )
    n = coins.size()
    dp[n][x+1] = {0}

    for t = 0 to t = x
        if t % coins[0] == 0
            dp[0][t] = t / coins[0]
        else
            dp[0][t] = 10^9

    for ind = 1 to ind = n-1
        for target = 0 to target = x
            take = dp[ind-1][target]
            nottake = 10^9
            if target >= coins[ind]
                nottake = 1 + dp[ind]
                                [target - coins[ind]
            dp[i][target] = min(take,
                                nottake)

    return dp[n-1][x]
```

So time complexity = O(x) + O(n x x)
space complexity = O(n x x

*) unbounded knapsack:-

given profit and weight arrays, capacity you need to get max profit using this capacity but here you have unlimited source of item, that is you can pick any item, any time you want.

same as previous knapsack problem but here we have unlimited resource of item so there will be slight change in base case and recursion.

base case:-

```
if (ind == 0)
    return (w / weight[0]) * profit[0]
```

Recursion

take = profit[ind] + f(ind, w - weight[ind])

this is same as coin change as we had unlimited coins.

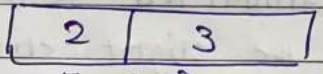
so when we have unlimited supply, then while taking don't change index.

(*) Rod cutting

You are given an rod length and price for rod length piece. You need to get max possible price.

Eg:- N = 5 , {2, 5, 7, 8, 10}

Ans = {2, 3} ⇒ 5+7 = 12 is max.



5 + 7 = 12

It looks somewhat same as knapsack, where we have weights of 1 to N and want to make W = N, using this weight which gives max value.

also we have infinite supply. (of rod lengths)

```
Solve(ind, N)
  if ind == 0
     return N * Price[0]

  notTake = solve(ind-1, N)
  Take = -10^9
  if ind+1 ≤ N
     take = Prince[ind] + solve(ind, N-ind-1)

  return max(take, notTake)
```

Note:- Here we have/can take rod of same length for more than one time.

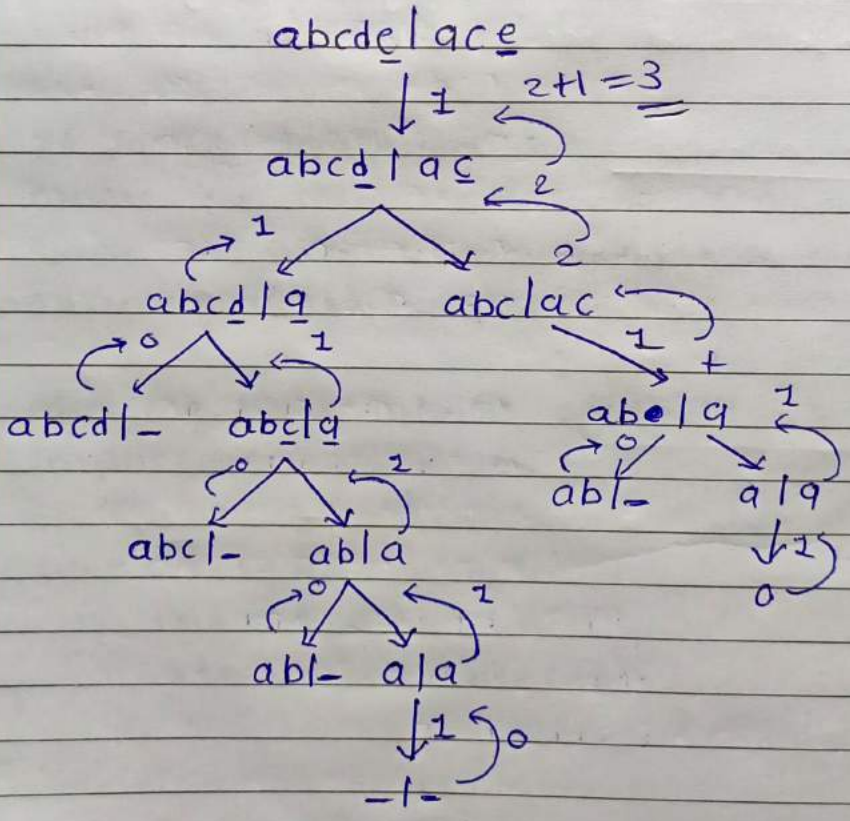
So while taking we are not doing ind-1.

DP on string:-

longest common sub-sequence.

Given two string you need to print the string that is longest and sub-sequence in both.

Eg:- abcde ⟹ ace.
ace



this is how we will do.

start computing from last
if char is same at that index
go to next/prev index for both
and add 1
else
first go for one string 1 prev.
then go for another string 1 prev
return max of them.

so we will have two index:
ind1 } this is state for
ind2 } our dp.

so
f(ind1, ind2)
if ind1 < 0 or ind2 < 0
return 0

if s1[ind1] == s2[ind2]
return 1 + f (ind1-1, ind2-1)

else
op1 = f (ind1-1, ind2)
op2 = f(ind1, ind2-1)
return max (op1, op2)

Now we can add dp for our states
ind1, ind2.

For tabulation we will need to do index shifting.

because in recursion base case is for ind < 0, so -1, but table can not have minus index

so,

base case will be ind == 0

so,

```
solve ( s1, s2)
    n = s1.length()
    m = s2.length()
    dp[n+1][m+1] = {0}

    for i = 0 to i = n
        dp[i][0] = 0

    for i = 0 to i = m
        dp[0][j] = 0

    for i = 1 to i = n
        for j = 1 to j = m
            if s1[i] = s2[j]
                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]
```

base case

Note:- Here we have asked about subsequence, but if we want to do same for substring then second case will be unusable.

So,

```
for i=1 to i=n
    for j=1 to j=m
        if s1[i] == s2[j]
            dp[i][j] = 1+dp[i-1][j-1]
        else
            dp[i][j] = 0
        ans = max(ans, dp[i][j])
```

Now to print LCS, we need dp array then,

```
i=n, & j=m, ans = ""
while(i>0 and j>0)
    if s1[i-1] == s2[j-1]
        ans = ans + s1[i-1]
    else if dp[i-1][j] > dp[i][j-1]
        i--
    else
        j--

reverse(ans)
print ans
```

* Longest palindromic subsequence:

Given a steing, find out the longest palindromic subsequence.

Eg:- bbbab ⇒ 4 (bbbb)

Eg:- cbbd ⇒ 2 (bb)

If we observe ans is LCS of s and reverse of s

that is,

LCS of { bbbab ⇒ 4 (bbbb)
babbb ← reverse of s.

so,

solve (s)
    steing s_1 = s
    reverse(s.begin(), s.end())
    steing s_2 = s

    return LCS(s_1, s_2);

*) Minimum Insertion required to make string palindrome :-

Given a string s, you need to make it palindrom with minimum insertion of alphabets.

Eg:- abcaa ⟹ 2 (aabcbaa)

So we will make Longest palindrom constant and make some insertion for remaining.

So for abcaa Longest palindrom is aca

So,

a b c a a → we need pair for 'b' and 'a'

a b c baa → inserted 'b'

aab c b a a → inserted 'a'

Now,

aabcbaa ⟹ palindrom.

this is how we need to add

length(s) − LCS(s, reverse(s))

characters.

(*) Shortest common supersequence:-

You are given two string S_1 and S_2. You need to construct the sortest string that contains S_1 and S_2 as sub-sequence.

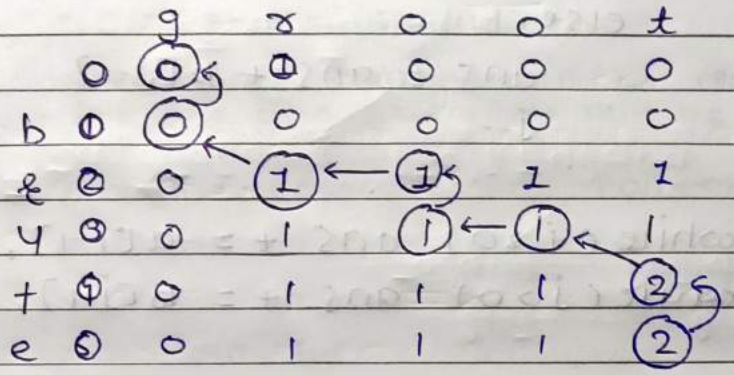
Eg:- S_1 = bute
S_2 = groot

output:- b g r o o t e

it contains both as sub-sequence

So what we will do is
i) find out LCS
ii) write down S_1 by substracting LCS from it
iii) Do same for S_2
iv) now write LCS.

let's make LCS dp table



so, e t o y o b g.

→ if cell has same value write once, else write both.
→ reverse it
⇒ gboyote ✓

so,
solve (a, b)
LCS (a, b)
dp[n+1][m+1]

i = n, j = m
ans = " "

while i > 0 and j > 0
if a[i-1] == b[j-1]
ans = ans + a[i-1]

else if dp[i-1][j] > dp[i][j-1]

```
        ans = ans + a[i-1]
        i--
    else
        ans = ans + b[j-1]
        j--

while(i>0) ans += a[i-1], i--
while(j>0) ans += b[j-1], j--

reverse(ans)

return ans.
```

(*) Distinct subsequences.

given two string s1 and s2, want to know how many distinct subsequence of s2 are present in s1.

Eg:. s1 = babgbag
s2 = bag

output :- 5

~~babgbag~~
~~babgbag~~
babgbag
babgbag
babgbag

so

we will start from tail

if s1[ind] == s2[ind]

then both pointer will be decreased but one time s2 pointer will be same.

else

one time decrease one pointer of s1.

~~one time decrease other pointer.~~

base case will be

if s1 is exhausted return 0

if s2 is exhausted,

means we have found s2 in s1

so return 1.

states:- both pointer

so,

solve(i, j, s1, s2, dp)

if i < 0 return 0

if j < 0 return 1

if dp[i][j] ≠ -1

return dp[i][j]

if s1[i] == s2[j]

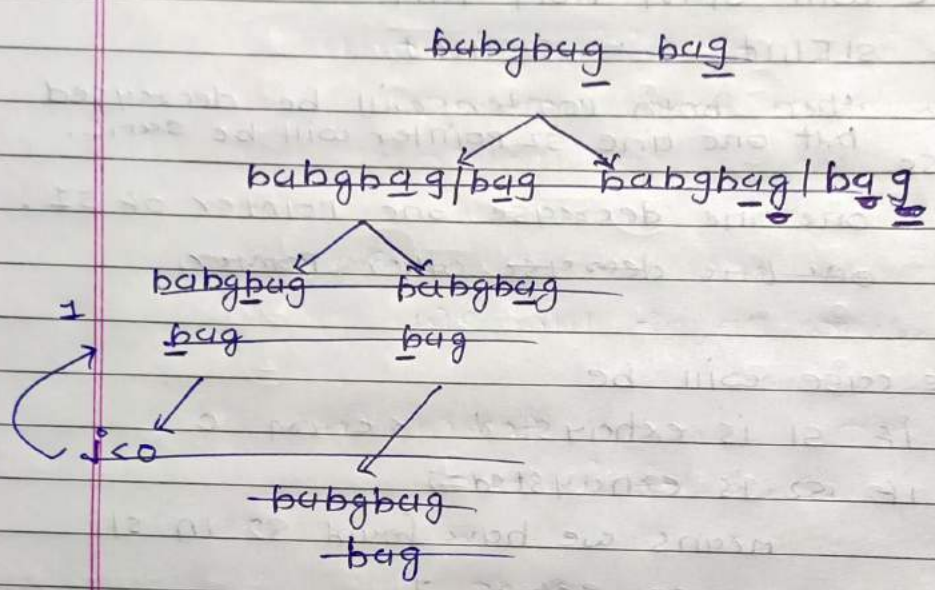
return solve(i-1, j-1, s1, s2, dp)

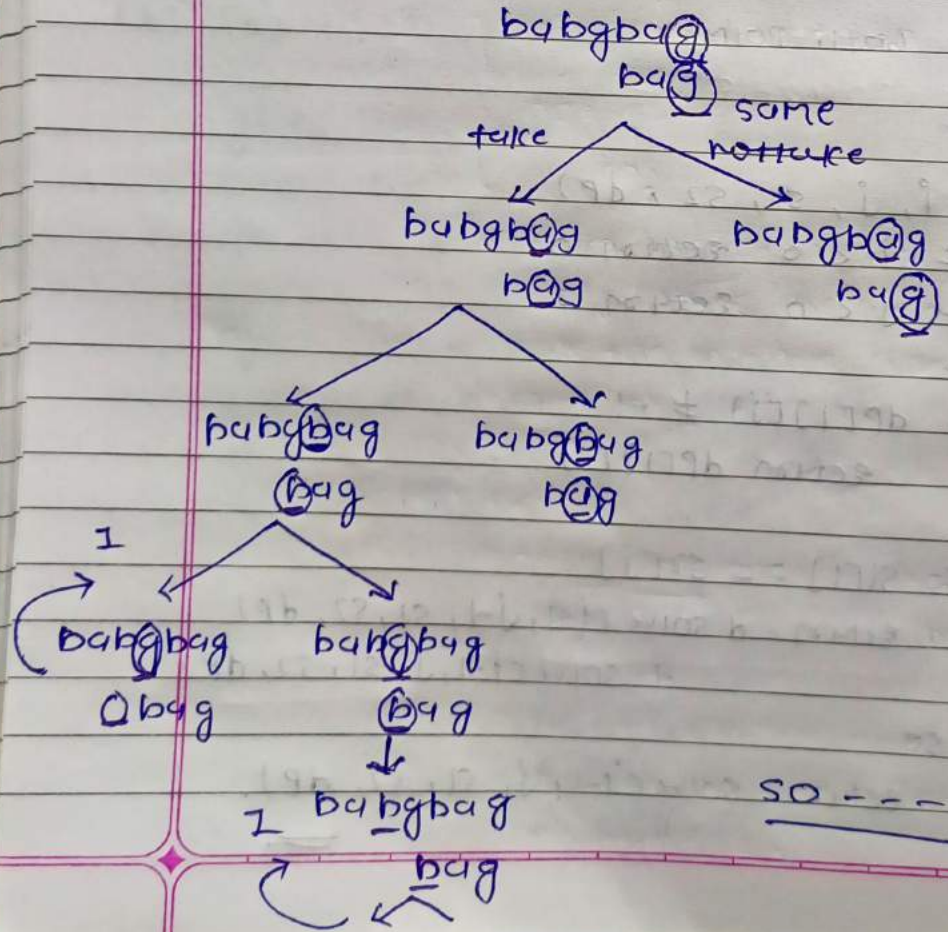
+ solve(i-1, j, s1, s2, dp)

else

return solve(i-1, j, s1, s2 dp).

Eg:- babgbag bag





(*) Edit distance:-

we are given steing s1 and s2. we need to convert s1 into s2. following three operation is allowed

i) Deletion of character
ii) Replacement of character with other char
iii) Insertion of character.

we have to return minimum number of o/p.

⇒ so we have 3 operation, we will try all combination and then will find min.

so it is about recursion.

Eg:- s_1 = horse op:- 3
s_2 = ros

horse → rorse → rose → ros

so we will follow three step:-

(i) Express everything in terms of Index.

so,

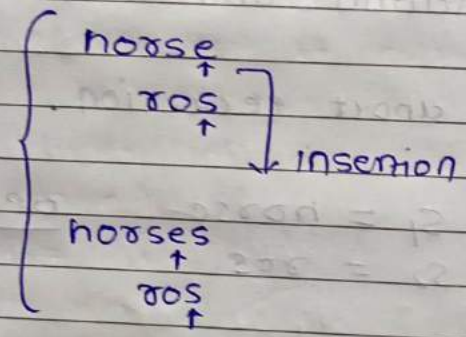
f(i,j) ⇒ min no of operation required to convert s1[0..i] to s2[0..j]

(ii) Tryout all possible ways :-

(i) when s1[i] == s2[j]
we don't need to perform any operation.

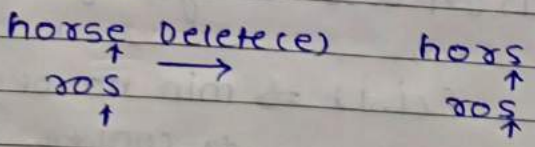
else,
we will try all ways,

(i) insert
solve(i, j-1, s1, s2)



→ After insertion second pointer only moves, because hypothetically we are inserting after the ith pointer.

(ii) Delete



→ here ith pointer moves only because we have no match for s1 we are still finding s.

(iii) Replace

horse → Replace → horss

ros → ros

we will replace with desired char and now both pointer

so,

solve (i-1, j-1, s1, s2)

(iii) return min of all.

Now base case:-

(i) when $i < 0$

so s1 has got exhausted.

horos we have as s1

s2 = horoso

we will need to insert need character, as our source has got empty now.

so return $\underline{i+1}$ (oth index so +1)

(ii)

when $j < 0$

we got our s2 in s1, but still having some extra char in s1, we need to delete them

so return $\underline{i+1}$

(*) DP on Stock 1:

(i) You are given prices, you can perform one transaction only, find-out the max profit you can make

Eg:- [7,1,5,3,6,4] O/P:- 5 (1-6)

We can max the profit by bying at min price and sell it after max price so,

```
solve (price)
    minP = price[0]
    profit = 0

    for i = 1 to i < price.size()
        cost = price[i] - minP
        profit = max(profit, cost)
        minP = min(minP, price[i])

    return profit.
```

(ii) on each day you can buy or sell, but can not sell before buy or can not buy after buy immediatly. you have to sell before buy.

Here we have many try available and we have one buy-sell constraint.

so we will try all possible combination of buy and sell then find out max.

See while buying our profit will be

$\Rightarrow -price[i]$

Steps:-

f(ind, buy) $\Rightarrow$ max profit we can make at till index ind and can buy or not on curr day.

if buy = 1, we have to buy or not buy, but can not sell for sure

when buy = 0, we have to sell, we can not buy. we can also hold though.

(ii) All possibility :-

if(buy == 1)
    buy = solve(i+1, 0) - price[i]
    notbuy = solve(i+1, 1)

else

    sell = solve(i+1, 1) + price[i]
    notsell = solve(i+1, 0)

(iii) find out max profit

    profit = max(buy, notbuy)
    profit = max(sell, notsell).

Base case :-

when i == n
    return 0;

now we do not have any day left so, 0.

(iii) Atmost two transaction :-

add one more state to the previous question that is cap, = 2.

while selling decrease it by 1, because we have completed one transaction

and add one more base case:-

```
if cap == 0 return 0;
```

that's it.

(*) DP on longest Increasing subsequence

(i) longest increasing subsequence.

Given an array, you need to return max length of increasing subsequence.

Eg:- [10, 9, 2, 5, 3, 7, 101, 18]

Ans:- 4 [2, 3, 7, 101]
or
[2, 3, 7, 18]

So, subsequence
↓
try all Path
↓
Recursion
↓
memoization

Here while constructing LIS, we look to previous guy, that if previous guy is smaller than current then add curr to our answer.

So urs states are:-
current index
prev index.

So f(ind, prev-ind)
↳ length of LIS starting from ind, where the last index that is part of LIS is prev-ind.

Now we will explore all possibility:-

that is take, not take

(i) If we does not take curr index as our subsequence part, then our previous index will not change.

So, f(ind+1, prevind)

(ii) when we can take curr index?
if a[curr index] > a[prev index]
or
our starting prev index will be -1 wen.

so, if prev-ind == -1 or
a[ind] > a[prev-ind]
take = 1 + f(ind+1, ind)

Here our length will increase by one, and our previous index will be update so.

Now return max of these.

Now, base case will be when ind == a.size

so if ind == a.size()
return 0;

Now for memoization we need to take

dp[n][n+1]
n = a.size()

Here n+1, because -1 can't be index so we will add +1 to it always.

Solve(ind, prev_ind, a, dp)
if ind == a.size
return 0

if dp[ind][prev_ind+1] != -1
return dp[ind][prev_ind+1]

nottake = solve(ind+1, prev_ind, a, dp)
take = 0
if (prev_ind == -1 or
a[ind] > a[prev_ind])
take = 1 + solve(ind+1, ind, a, dp)

return dp[ind][prev_ind+1] = max(take, nottake)

Here we have done $-1 \to 0$, $0 \to 1$, $1 \to 2$, ---
$n-1 \to n$ increment change for prev-ind that is called co-ordinate change.

TC :- $O(n \times n)$
SC :- $O(n \times (n+1)) + O(n)$

we can do tabulation by,

```
dp[n+1][n+1]

for i = n-1 ; to i = 0
    for prev = i-1 to i = -1
        {copy recurence.}

return dp[0][0]
```

After this space optimization will be there
that is,

using two array of

curr(n+1)
prev(n+1)

But we can do it in 1D DP also
let's see it.

*) largest Divisive subset

Given a set of distinct positive integers, return the largest subset such that for every pair (i,j)

ans[i] % ans[j] == 0

or

ans[j] % ans[i] == 0

Eg:- {1, 16, 7, 8, 4}

Ans = {1, 16, 8, 4}

so we will have to sort it

as 1 4 7 8 16

so, if we pair {1, 4 --}

and looking for {8}

then,

if 8 is divisible by last (4)

it is divisible by all others too

so we will loop through array,

actually this is same as LIS with some changes we need to use tabulation method of LIS to solve this.

```
Solve ( arr, n )
    dp[n] = {1}, hash[n] = {0}
    sort (arr)

    for i = 0 to i = n , hash[i] = i
        for j = 0 to j < i
            if arr[i] % arr[j] == 0
                    and
                dp[i] < dp[j] + 1
                    dp[i] = dp[j] + 1
                    hash[i] = j

    int ans = -1
    int lastIndex = -1

    for i = 0 to i < n
        if dp[i] > ans
            ans = dp[i]
            last index = i

    ans = { }
    ans.push_back ( arr[last Index] )

    while hash[lastIndex] != lastIndex
        lastIndex = hash[lastIndex]
        ans. pushback( arr[lastIndex] )

    return ans.
```

*) Longest steing chain

you are given array of words return the length of longest chain.

so if there is w_1, w_2 two word then if we can insert any one char in w_1 and can make w_2 then it is chain.

Eg: [a, b, ba, bca, bda, bdca]

output:- 4

a, ba, bca, bdca.

Here that is not necessary to have subsequence so will sort it based on length

then just adding one more condition to last problem and removing mod condition we are good to go

Like,
we will take
'a', now if diff between (a, b) = 1 then take it else not

to check if the different is one or not

```
bca     bdcq   ⟹ same so both pointer ++
↑       ↑
bca     bdcq   ⟹ not same then increment the largest pointer only,
 ↑       ↑
bca     bdcq
 ↑       ↑
bca     bdca
 ↑       ↑
```

If they reach end at the same time then possible else not

big small

so check (s1, s2)

```
n = s1.size()
m = s2.size()
if n != m+1 return false

first = 0, second = 0
while (first < n)
    if s1[first] == s2[seco.
        first ++
        second++
    else
        first ++

if first == n and second == m
    return true
return false.
```

*) longest bitonic subsequence:-

Given an array of integer, you need to print the longest length of bitonic subsequence.

Here bitonic means increasing and then decreasing.

Eg:- {1, 11, 2, 10, 4, 5, 2, 1}

ANS. ⇒ 6 {1, 2, 10, 4, 2, 1}

Here, It can be decreasing only or increasing only.

so we will find,

for index i

I-dp[i] :- how many number are in 0 --- i-1 which can be take for LIS

D-dp[i] : how many number are in i+1 --- n which can be taken for LIS.

now find max for ∀ i

max(D-dp[i] + I-dp[i] + 1)

=.

for I-dP[] we know the process

so for D-dP[]

$$= \begin{cases} \text{for } i = n-1 \text{ till } i = 0 \\ \quad \text{for } j = i+1 \text{ to } n \\ \quad\quad D\text{-}dP[i] = \max(D\text{-}dP[i], \\ \quad\quad\quad 1 + D\text{-}dP[j]) \end{cases}$$

(*) No of longest Increasing subsequences

Given an array, return the number of LIS in that array.

Eg:- [1, 3, 5, 4, 7]

Answer :- 2

How ?

[1, 3, 4, 7] ↵

[1, 3, 5, 7] ↵

let's try for this below example using previous LIS problem knowledge.

	1	5	4	3	2	8	2	6	7	10	8	9
dP[]	1	2	2	2	2			2	3			
count[]	1	1	1	1	1			1	4			
								2	9			
								3				
								4				
								5				

so on---

there is 5 LIS that ends with 6

Here dp[i] :- longest LIS length
count[i] :- no of LIS of dp[i] length.

```
Solve ( arr, n)
    dp[n] ={1}
    count[n] = {1}

    maxi = 1
    for i = 0 to n-1
     for j = 0 to i-1
        if arr[j] < arr[i]
            and dp[i] < 1+dp[j]
            dp[i] = 1 + dp[j]
            count[i] = count[j]
        else if arr[j] < arr[i]
            and dp[i] == 1 + dp[j]
            count[i] = count[i] + count[j]
     maxi = max (maxi, dp[i])

    int ans = 0
    for i = 0 to n-1
        if dp[i] = maxi
            ans = ans + count[i]

    return ans.
```

(*) Partition DP

whenever we need to find the answer to a large problem such that the problem can be broken into subproblems and the final answer varies due to the order in which subproblem are solved, we can think in terms of partition DP.

1. Matrix chain multiplication :-

Problem :-

Given n matrix dimentions, return the min cost to multiply them to a single matrix.

Eg :- A = 10 × 30
B = 30 × 5
C = 5 × 60

So (AB)C ⇒ 10 × 30 × 5 + 10 × 5 × 60
= 4500

A(BC) ⇒ = 10 × 30 × 60 + 30 × 5 × 60
= 27,000

So, ANS = 4500

ABCD,

- (AB)(CD) possible
- (A(BC))D possible
- A(B(CD)) possible

Given [10, 20, 30, 40, 50]

so,

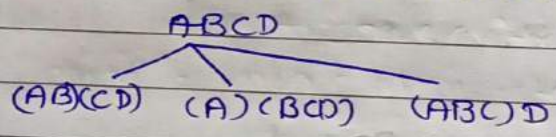
1'st matrix : 10 × 20

2nd matrix : 20 × 30

i^{th} matrix : A[i-1] × A[i]

Rules :-

1. start with entire block/array.



i :- start point.

j :- end point.

2. Try all partitions.

A B C D → (AB)(CD)

A|B C D → (A)(BCD)

A B|C D → (AB)(CD)

ABC|D → (ABC)(D).

3. Return the best possible two partition.

f(1,4) :- Return the min multiplication required to multiply matrix 1 to matrix 4 chain.

i, j will shrink.

so base case:

if i == j return 0

→ when i will be equal to j, there will be only one matrix, so we required 0 operations only.

Now to try all partition,

for k = i to k = j-1 // Here j-1 because then will be only one matrix at last.

f(i,k), f(k+1,j)

so,

steps = A[i-1] * A[k] * A[j]

+

f(i,k) + f(k+1, j)

mini = min(mini, steps)

like, 10 20 30 40 50

i,k j

so (10,20 × 20,30) × (30,40 × 40,50)

↓1 ↓

= 10 × 30 30 × 50

↓

10 × 30 × 50

So,

```
solve(i, j, arr, dp[][])
    if i == j
        return 0

    if dp[i][j] != -1
        return dp[i][j]

    int mini = 10^9

    for k = i to k = j-1
        steps = arr[i-1] x Arr[k] x arr[j]
                + solve(i, k, arr, dp)
                + solve(i+1, j, arr, dp)
        mini = min(mini, steps)

    return dp[i][j] = mini
```

Time complexity : $O(N^3)$.

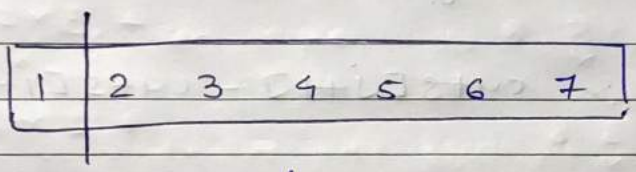
2. Minimum cost to cut the streak (stick)

We are given a stick of length N and a CUTS array. The stick has markings as shown and the CUTS array depicts the marking at which the stick need to

be cut.

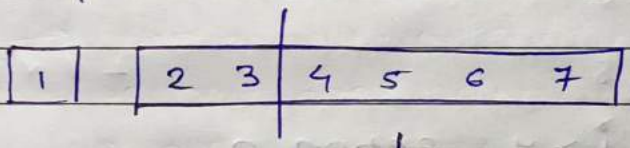
Eg:- CUTS : [1, 3, 4, 5]

n : 7

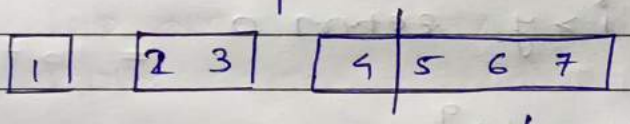


· CUTS[0]

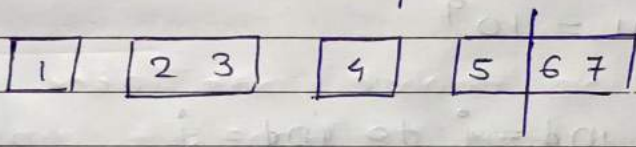
cost = 7



cost += 6



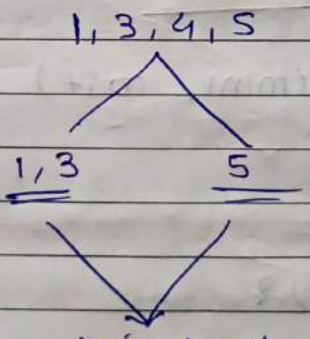
cost += 4



cost += 3.

= 20.

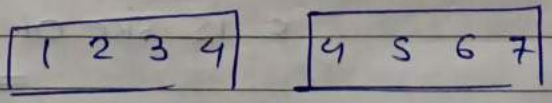
this is brute force approach of cutting we need to minimise it.



suppose we cut at 4

this both stuff is independent from each other.

because



X

0 1 3 4 5 7
i j

cost :- length of stick to be cut

cost :- cuts[j+1] - cuts[i-1]

so,

f(i, j)

if i > y return 0

mini = 10^9

for ind = i to ind = j

cost = cuts[j+1] - cuts[i-1]
+ f(i, ind-1)
+ f(ind+1, j)
} Remaining path.

mini = min(mini, cost)

return mini;

so DP:- states ??

i, j

dp[c+1][c+1]

c is size of CUTS;

Time complexity:-

$O(c^3)$ { two changing parameter and one for loop }.

Space complexity:-

$O((c+1)^2)$.

Now for tabulation, we need to think bottom up;

So

```
for i = c till i >= 1
    for j = 1 to j <= c
        if i > j continue
        // copy recurrence

return dp[1][c].
```

3. Burst Balloons:-

You are given n balloons, painted with some number on it. You asked to burst all them If you burst ith balloon then you will get num[i-1] x num[i] x num[i+1] cost. You are asked to maximise this cost as possible.

Eg:-

3 1 5 8 cost = 3×1×5

3 5 8 + 3×5×8

3 8 + 1×3×8

8 + 1×8×1

= 167.

Here if we choose one index and then there will be two sub problem but they are not independent

like

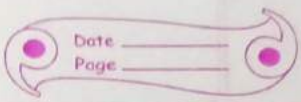
b_1 b_2 b_3 b_4 b_5

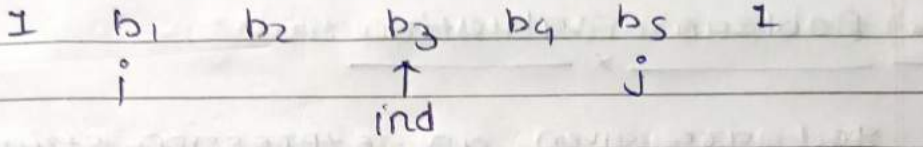
b_1 b_2 b_4 b_5

$= b_2 \times b_4 \times 5$

we need this from other part.

so we will start from down side:-





cost = nums[ind] × nums[i-1]
× nums[j+1]

for all ind = i to ind = j

and

+ f(i, ind-1)
+ f(ind+1, j)

and we will maximise it.

so

```
f(i, j)
    if i > j
        return 0
    for ind = i to ind = j
        cost = nums[i-1] x nums[ind]
                        x nums[j+1]
                + f(i, ind-1)
                + f(ind+1, j)
        maxi = max(maxi, cost)
    return maxi
```

to memoize we will have i and j as states.

4. Boolean Evaluation :-

You are given an expression string, where operands will be T and F, operator will be AND, OR, XOR.

You need to find the number of ways we can parenthesize the expression such that it evaluate to True.

Eg:- F | T ^ F

(F|T)^F ⇒ T ^ F ⇒ T
F|(T^F) ⇒ F | T ⇒ T

So output :- 2

So

T ^ F | T & F ^ T

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

or I can,

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

or I can

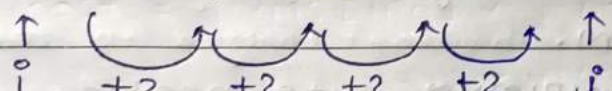
(T^F|T&F)^(T)

Many partition possible.

so pattern is breaking down at operand.

so,

T ^ F | T & F ^ T | F



partition can be at

let's say we have & has at middle index

left & right

x ways for true

y ways for true

so = x x y only and only both are true.

for OR

left	right
T = x_1	T = x_2
F = x_3	F = x_4

so total = $x_1 \times x_2$ (both true)
+ $x_3 \times x_2$ (one is true)
+ $x_4 \times x_4$ (one is true)

for XOR (gives T when both are not same)

left ^	right
T = x_1	T = x_2
F = x_3	F = x_4

total = $x_4 \times x_4$ (both are not same)
+ $x_3 \times x_2$ (both are not same

5 *) palindrome partitioning:-

Given steing s, partition s such that every substeing of the partition is a palindrom.
return minimum cuts needed.

Eg:- bab|abcba|d|c|ede

⇒ 4.

Front partitioning:-
→ start from front
→ check if I can do partion or not, based on question condition

→ ~~b|ababcbq~~
~~yes (b is palindrom)~~
~~⇒ 1 + (ababcba)~~
→ ~~ba|babcba~~
~~NO (ba is not palindrom)~~
→ ~~bab|abcba~~
~~yes (bab is palindrom~~
~~⇒ 1 + (ababcba)~~
→ ~~so on..~~
⇒ a|babcba
yes
→ ~~ab~~|abcba
~~NO~~ yes.

so on..

bababcbadcede

↳ do :- b | (ababcbadcede)

don't :- ba | ---
not possible as "ba" is not palindrom

↳ Go to next

do babl ---
yes we can.

---

this is bont partition

so Now we can implement this :-

F(i) :- min partion required to cut string s[i..n].

so,

```
f(i, s)
    if i == s.size()
        return 0
    mini = 10^9
    for ind = i to ind < s.size
        if (isPalindrom (i, ind, s)
            cuts = 1 + f(ind+1, s)
            mini = min(mini, cuts)

    return mini
```

6 *) Partition Array for max sum

Given array, partition the array into subarrays of max k length. after partitioning each subarray changed to the max of that partion.

return the max sum after partition.

Eg:- [1, 15, 7, 9, 2, 5, 10] , k=3

[1, 15, 7 | 9, | 2, 5, 10]

⇓

[15, 15, 15, 9, 10, 10, 10]

gives 84

Rules or steps for partition recursion :-

1. express every thing in terms of index
2. Try out all possible partition from that index.
3. choose the best from them

F(0) :- max sum if we have array from [0...n]

Here we can use concept of front partitioning.

Here we are allowed the max size of partion is k, that is our condition

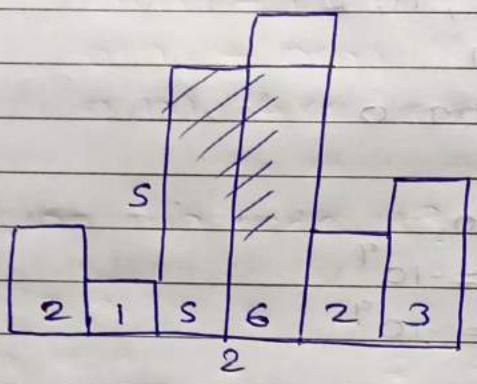
```
solve(i, arr, n, k)
    if i == n
        return 0

    int len = 0
    int maxi = -10^9
    int ans = -10^9

    for j=i to j = min(i+k, n)
        len = len+1
        maxi = max(maxi, arr[j]
        ans = maxi * len + solve(j+1, arr, n, k)

    return ans
```

Here i is state for our dp solution, so we can memoize it using dp array of n size.

*) prerequisite for next que

7 *) largest Rectangle in Histogram.
Given an array of integers heights represent height of histogram's bar width is 1 for every bar, return the largest rectangle in the histogram.

Eg:- [2, 1, 5, 6, 2, 3]



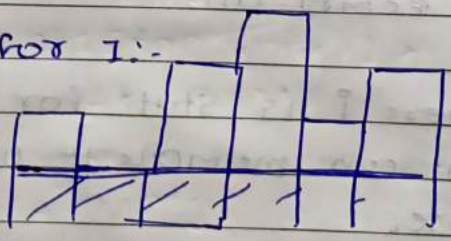
max = 5×2
= 10

So Here starting thought solution can be take the curr bar as consideration and look at left and right of it to make area (Rectangle)

So for 2:-

2×2
=2

for 1:-

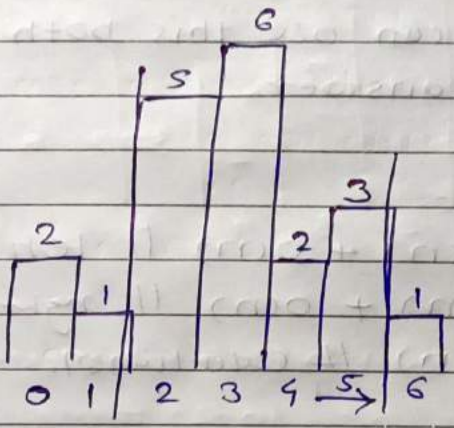


1×6

so on...

Here what we are doing is finding first smaller element than curr on both of the side. that will give us width and curr is our height for sure.

Eg:-



for curr = 2 (height)

$= (5-2+1) \times 2$

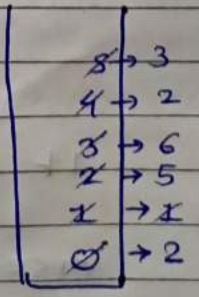
$= 8$

It will give us $O(n^2)$ TC

So for optimisation we should use stack for next smaller elements.

left small.

0	0	2	3	2	5	0
0	1	2	3	4	5	6



as, same as previous,
for right smaller we will start from last.
just we need to minus 1, while hitting smaller on the stack instead of adding 1.

Now we can use this both array to compute answer.

Here TC:-

O(n) + O(n) // left smaller
O(n) + O(n) // right smaller
O(n) // calculation.

How we can code it?

so,

for left:-

```
stack<int> st
left[n]

for i=0 to i<n
    while stack is not empty
                and
          heights[top(stack)] >= heights[i]
                st.pop()

    if not stack.is empty
          left[i] = st.top() + 1
    else
          left[i] = 0

    st.push(i)
```

For right:-

```
stack<int> st
right[n]

for i = n-1 to i = 0
    while stack is not empty
                and
          heights[st.top] >= heights[i]
                st.pop()

    if st is empty
        right[i] = n-1
    else
        right[i] = st.top() - 1
    st.push(i)
```

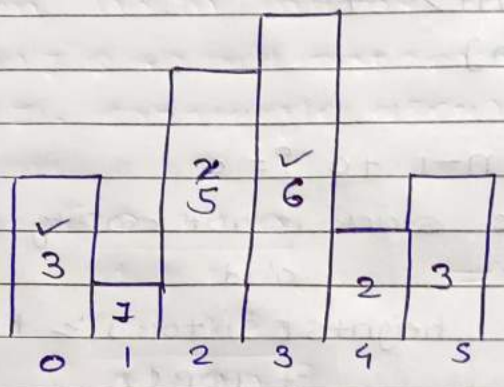
calculate ans:-

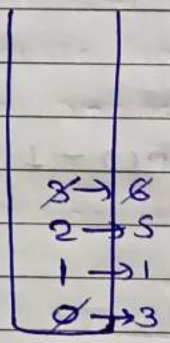
```
ans = 0

for i = 0 to n
    ans = max(ans, heights[i] x (right[i] - left[i] + 1))
```

Formula was:-

(right - left + 1) x curr

can we do in one pass? yes...





→ Now when our pointer came to 1 there is '3' on the stack that is greater than '1', so for '3' ~~left~~ smaller = right is '1', pop stack

→ Now for ~~left~~ smaller we have to load into stack but it is empty so '0' is left smaller

→ so for 3, max area is $3 \times 1 = 3$

→ now push 1

→ now go ahead that is greater than top so push it.

→ Here we are trying to make increasing order into the stack

Now pointer came to i = 4 that is 2 and less than top.

so for top left smaller is at 2 (from stack)
right smaller is at 4 (curr pointer)

→ pop it.

Now on top we have 5 that is greater than curr pointer 4 that is 2 value

1 ⟵ 5 ⟶ 4

= 5 × (4-1-1) = 10

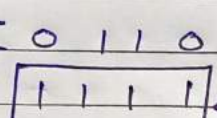
so on... .

```
stack<int> st
ans = 0

for i = 0 to i = n
    while stack is not empty
                and
        i == n or h[st.top()] >= h[i]
            height = h[st.top()]
            st.pop
            if stack is empty w = i
            else w = i - st.top() - 1
            ans = max(ans, height x w)
return ans
```

Max Rectangle area

Given n x m matrix formed of 0's and 1's. find out max area formed by 1. rectangle area)

Eg:- [0 1 1 0

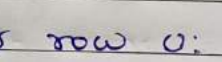


⇒ 8

1 1 0 0]

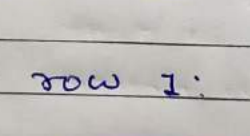
Here we will try to form histogram possible for every row and find the max rectangle area.

like for row 0:



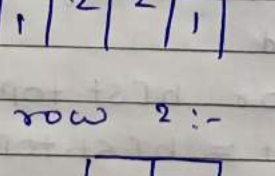
Here max area = 2x1 = 2

for row 1:



Here = 2x2 = 4

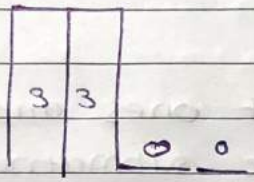
for row 2:-



Here 2x4 = 8

so on

for row 4:-



here $3 \times 2 = 6$

out of all max is 8.

let's code it up

```
solve ( matrix, n, m )
    heights[m] = {0}
    ans = 0

    for i = 0 to n
        for j = 0 to m
            if matrix[i][j] == 1
                heights[i]++
            else
                heights[j] = 0
        area = getMaxArea ( heights, m)
        ans = max (ans, area)

    return ans;
```

(*) count square submatrices with all ones.

given m x n matrix of ones and zeros, return howmuny square submatrices have all ones.

Eg:- 0 1 1 1 / 1 1 1 1 / 0 1 1 1 ⟹ 10 + 4 + 1 = 15

10 ↑ size 1×1, 4 ↑ size 2×2, 1 ↑ size 3×3.

Brute force can be stand at a index and try to expand it's size

but for dp solution, we will make dp matrix of same size m x n

dp[i][j] = how many rectangle ends at (i,j) that is how many square's right bottom is (i,j)

matrix

1	1	1
1	1	1
1	1	1

dp

1	1	1
1	2	2
1	2	3

Here what we are doing is finding min possible in previous neighbour and add one to it.

Matrix

1	1	1	1
1	1	1	1
1	1	1	1

dp

1	1	1	1
1	2	2	2
1	2	3	3

1+1 2+1

like this.

```
Solve(matrix, m, n)
    ~~For i=0 to m~~
    // Here m is row count
          n is col count //
    dp[m][n] = {0}

    For i=0 to m
        dp[i][0] = matrix[i][0]
    For i=0 to n
        dp[0][i] = matrix[0][i]

    For i=1 to m
        for j = 1 to n
            if matrix[i][j] == 1
                dp[i][j] = 1 + max( dp[i-1][j],
                                    dp[i-1][j-1]
                                    dp[i][j-1]
```

```
for i = 0 to m
    for j = 0 to n
        ans = max(ans, dp[i][j])

return ans.
```