40 DP PROBLEMS INTUITIONS AND PATTERN

STRIVER

for you

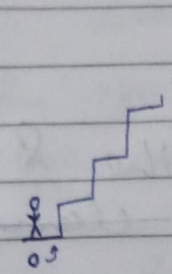
NOTES ->

Dynamic Programming

162

Climbing Stairs :

n stepped ladder to reach the top



x ← curr. posn

destn : n

⇝ (x) ⟶ (x + 1)

or

⇝ (x) ⟶ (x + 2)

} options.

Base case : if

~~x := n-1 [return 1]~~

~~x := n-2 [return 1]~~

~~x := n [return 0]~~

base : if x := n ⟶ return 1.

frog jump :

it is the almost same as Cat question but each stair step has a height gap which has to be added to energy consumption.

⇒ (x) ⟶ (x+1) + abs(x+1 - x) (solve)

(x) ⟶ (x+2) + abs(x+2 - x) (solve)

base if x := n-1 ⟶ return 0;

Good Write

Max Sum of Non-Adjacent elements

See when it comes to non adjacency then the only option we have in the beginning is

either we keep the S^{th} element or not

if you do!	if not
then you dont get the choise of $S+1^{th}$ element thus not taking it thus val $+(S+2)^{th}$	then same gn repeated for $S+1^{th}$

(16-3)

Ninja's Training

DATE: / /
PAGE:

if we start thinking of a recursive soln we can actually think that yes on each day ninja has 3 options.

Run | fight | Practise

Thus obv. we can replicate the past recursion case here doing s+1, but there is an issue: you cant do the same thing @ 2 consecutive days, thus will have to store the last donle activity and send it during recursion

we can have:

```
Solve (s, last, points) {
    for (i=0; i<3; i++) {
        if (i == last) { continue; }
        maxi = max(maxi, point[s] + solve[s+1, i, points]
    }
Base: if s > e ⟶ return 0;
```

we can ofc make a dp for the same of size 3 * n

62 Unique Paths

DATE: / /
PAGE:

On this question the first most approach that comes to mind is use backtracking // or recursion // or maybe dfs.

→ BT: we will kinda do that tho, we aren't supposed to maintain a visited matrix here as we are never gonna go back as for 'right' there is no 'left' & for 'down' there is no 'up'.

→ Recursion: That will be a good way to go as we are asking for number of paths so that would be a sum of paths from (x+1, y) ← Hater (x, y+1) ←

→ DFS: Technically recursion is itself DFS only.

→ Unique Paths 2

just check if grid[x][y] == 1

return 0 as a base case.

Good Write

→ min ... & an answer

Min path Sum in grid

The approaches that comes to mind are

→ recursion / backtracking:

Since we are just going R & D thus we dont need any vis matrix & no BT.

recursion can be done to get the min path sum from x+1, y & y+1, x and then add the grid[x][y]

base case is: if (x == n-1 & y == n-1) then return grid[x-1][y-1]

→ Djakstra

Or you can use the plain old priority queue with dijakstra's algorithm. to do this work.

DS used: ⇒ distance matrix
⇒ priorityqueue

Good Write

→ Min falling Sum :

here we must just fill the dp table for all the boxes. And will be the min of the top row.

that's all.

→ Chocolate pickup

Its Exactly quite an easy question.

you can go from $x,y \rightarrow x+1,y-1$ | $x+1,y$ | $x+1$ only.

Since we have 2 bloom thus that would be x_1, x_2, y_1, y_2 as the variable in every step, but wait $x_1 == x_2$ always as @ every step we are always doing $x+1$.

Thus. there are total 9 cases after that

$x_1, x_2 \rightarrow x_1+1, x_2+1$

y_1 → y_1-1, y_1+1, y_2

$\times$

y_2 → y_2-1, y_2+1, y_2

$\Rightarrow 3 \times 3 = 9$

thus a 3D-DP of (x, y_1, y_2)

Good Write

DP on Subsequences

DATE: __/__/____
PAGE: _____

1. Subset Sum equal to target:

Can easily be done by take / not take approach.

2. Partition Equal SubSet Sum:

we dont have to maintain the sum of both if for the subset, sum is SI then for other it is SO.

either you can go this way or just set the target to sum/2. in the question 1.

3. Partition Set into 2 Subsets with min absolute sum

This is an interesting one. @ first glance you may feel like trying to use a new recursion app. for it & it works too for a tonne but gives TLE.

there my app. was:

finding the min abs. sum if arr[s] is either taken or not-taken but we also have to send off a count var. to the same. { more volatile solution ?}

Good Write

the taking of ideas, another one is to just find if a specific target sum is possible for a specific array or not range is from 0 to netSum.

& just check if it's true then we take the min diff out of all those options

→ dp[0][i] ← gives you the full row of these req values

in n^2 time only. Not it will be

$O\{n^2 + (Sum)\}$ time

* If will ofc fail if you inc negative numbers as there we will get ~~target~~ netSum → -ve which can't be req by DP so either we shift them by a number or find another approach.

DP : converts wing stairs, hence.

DATE: __/__/__
PAGE:

4. LC Qn : 139

Word Break

Brute approaches

1. Normally use a trie & do the deal of searching by modifying the search function accordingly (done @ gfg by that)

2. String matching & then recursion for the rest of the part

Intuition :
we start from s & keep adding characters to our string. Once the string matches we send if its possible to get answer if we consider that as one word
thus
do solve ( )

line
apple pen apple
i j
solve(j, A)
now check for penapple.

Flaw: aaaaaaa   word : dict:
"aaaa" "aaa"

thus we both send recursion but also moving ahead to check for even bigger matches.

4. Count Subsets with Sum K

[1, 1, 4, 5]

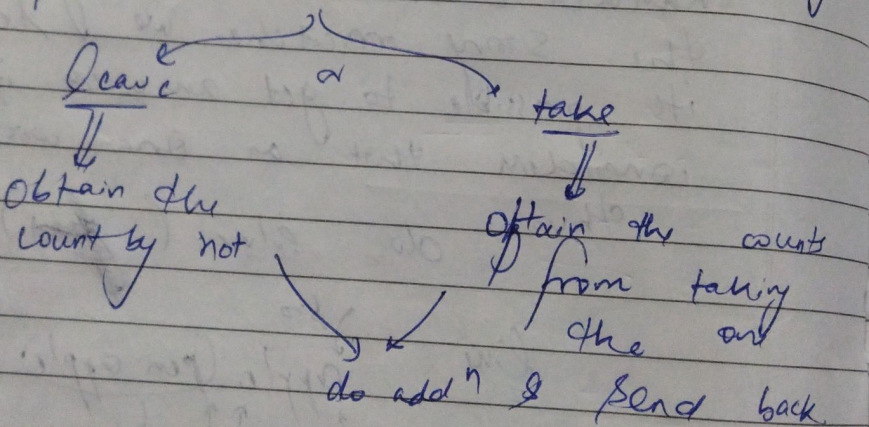
in this question we have to find which all subsets get sum as target value.

→ Thus all we gotta do is just set a condition @ (k == 0) & do cnt++.

(or)

→ recursion way:

for a array we have 2 options for the i^{th} index



base:

```
if (k==0){return 1;}
if (i>e){return 0;}
```

Good Write

→ Minimum Coins :

DATE : __/__/____
PAGE : _____

Its a question which doesn't works on pick / not pick as there will be no change in the options even if you pick something.

Rather here we have choices to take on what to choose first among all the elements in the array.

we choose any 1 ⟶ solve(amt - val)

& all coins are ∞ in number.

```
// base
if (amt == 0) { return 0; }

for (i = 0; i < n; i++) {
    mini = min(mini, solve(amt - val[i]))
}

also check if [amt - val >= 0] ✓
```

→ Minimum Coins ||

After doing min coins, d/c ur brain tells you to just add all the cases where you did amount - coins[i] and shit done!

Its kinda right but has a flaw, when you do this thing you are gonna take into consideration even the orders of numbers like

[1, 2, 5] target = 5

choose 1 now target = 4 (2, 5)

(3) = and: 2+2, 1+1+2, 1+1+1+1

choose 2 now target = 3

(2) [1+2, 1+1+1] repeated

choose 5 now target = 0

(1) [ ]

→ try to trivialise this i.e. we already do consider the picking of 2 when we picked 1 if we just selected for index : then warrant start from 2

Gone Write

and not restart as that would just repeat cases.

Code:

```
int Solve(int amt, vector<int> coins, int i){
    if(amt == 0){ return 1; }
    int ans = 0
    for(int i=0; i<coins.size(); i++){
        if(amt - coins[i] >= 0){
            ans += Solve(amt, coins, i);}
                    coins[i]
    }
    return ans;
}
```

→ Unbounded KP

This is the same qn as min Coins 2 that we are picking / not picking but the option of having the same item doesn't diminishes even. So.

Thus we choose every option for once & return the max poss. outcome.

→ Rod Cutting problem:

again we will consider all option one by one just like prev qn & find the max, by just changing the n value.

16.5 DP on Strings

DATE: __/__/____
PAGE: ____

⟹ LCS

i) count/size of longest:

when you are comparing any characters say text1[i] & text2[j] then

if text1[i] == text2[j]

ans += (1 + solve(i+1, j+1))

Not

→ i^{th} element (i, j+1) may be a part of another LCS

→ j^{th} can be (j, i+1)

→ none: (i+1, j+1)

we choose max & return.

ii) print the string:

Rather than dp of type int make of type String ⟹ rest is almost same & understandable too.

Good Write

Longest Common Substring

See this is not the same as LCS, you have to think accordingly.

The reason we were using the strategy of dp[i+1][j+1] for LCS is due to fact as once a char is matched for max LCS we will obv. take it into final consideration always. not here.

we can find a substring but it may/ maynt be the best so how do you get this done?

eg: s1 = a b c j k l p

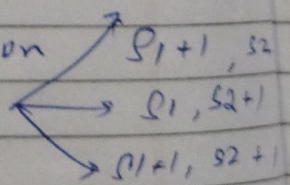
s2 = a c j k p

here here we 2 components:

we first try to find the largest substr, (common) from s1 & s2. & store its cnt

like here it will be 1 ("a")

then we have to send for recursion



as now we should have both s1 & s2 in the substring (obv. cone in cnt)

thus we find max of cnt & these & sent max back.

Good Write

⇒ Longest Common Substring

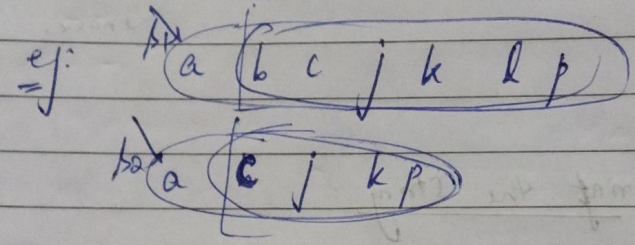
DATE: __/__/____
PAGE: _____

See this is not the same as LCS, you have to think accordingly.

The reason we were using the strategy of do: if solve (i+1, j+1) for LCS is the fact as once a char is matched for max LCS we will obv. take it into final consideration always. not here.

we can find a substring but it may/ maynt be the best so how do you get this done?

eg:

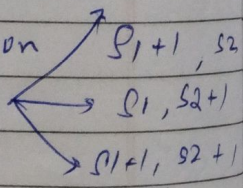


here here are 2 components:

we first try to find the largest substring (common) from S1 & S2. & store its cnt

like here it will be 1 ("a")

then we have to send for recursion



as now we should have both S1 & S2 in the substring (obv. consin cnt)

thus we find max of cnt & these 3 & sent max back.

Good Write

Longest Palindromic Subsequence

DATE: __/__/____
PAGE:

It is quite a good question if handled individually

We wanna find the longest palindromic subsequence.

First thoughts are to try to check from front & back both but its kinda bad & time taking

Remember:

longest palindrom subseq is same as

lcs (s, reverse of s).

Min Insertion Steps to make a String palindrome

So when we start with this question our first obv thoughts are the fact that we must try comparing the ends & keep moving.

now when you compare the ends there can be 2 options

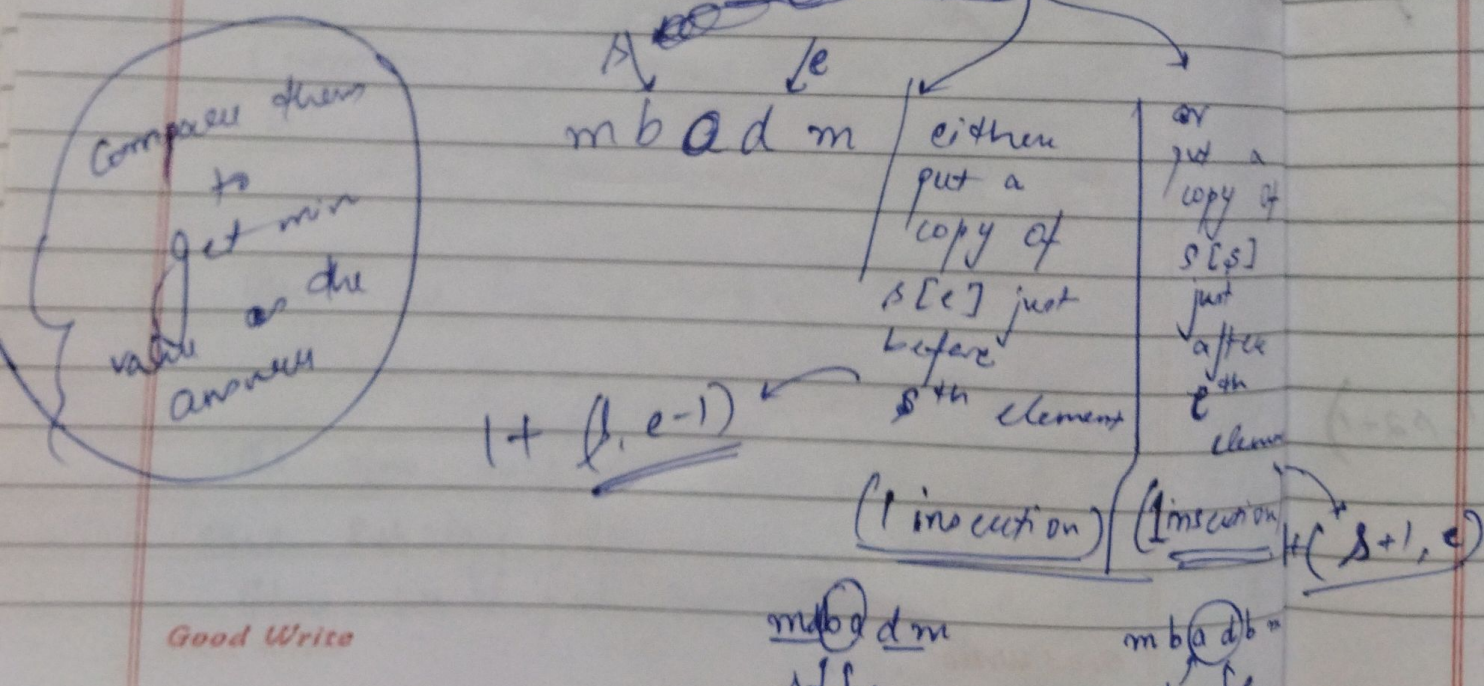
they match

if they do just do $s++$, $e--$ → call recursion (makes sense?)

$0 + (s+1, e-1)$

they don't match

here is the weird part we know somehow that we gotta do smth about this for sure



Good Write

⇒ Delete Operation for two strings

easy | easy → find the lcs of both ⇒ that is the actual end game we wanna reach

↳ ans:

⇐ size1 + size2 - 2 * lcs.size()

* Similar qn on Min ASCII del sum for 2 strings

⇒ Shortest Common Super Sequence

We have to make our sequence such that both strings are the subsequences of this.

See idea is simple to find the ans of this we just have to first find the lcs string *now*

for the subsequence thing to work we have to make sure every character which comes before the common characters stays on left of it

eg m a d p a

t a s k p l e → lcs: ap

H(s+1, e)

m t a d s k p a l e

Good Write

Distinct Subsequences

This is a v. interesting qn

① Tc: $O(n^2 * m)$: Not the best but intuitive

if we have

→ (b a b g b a g) s

→ (b a g) t

we will compare t[0] with all chars of s.

& will do

ans += solve(s1+1, s2+1)

if s[s1] == t[s2]

i.e. every case where the starting will match 'b' here

we will add the number of times "ag" occurs in the continued array

i.e. for s1 = 0 → runs from 0 to n-1

s2 = 0

→ char c = t[s2]

if s[k] == c { ans += solve(s1+1, s2+1)

Good Write

DATE: __/__/____
PAGE: _____

(2) The problem was in the iteration we were doing in that $s2$ to n-1 of 's'.

So we start our $s1$ from 1
$s2$ from 1

* if [s(s1-1) == t(s2-1)] then

ans = solve(s1+1, s2+1) + solve(s1+1, s2)

'ag' standing ahead

covers all the cases of "bag" which comes later directly

else

* ans = solve (s1+1, s2)

ans not start here, move ahead.

Good Write

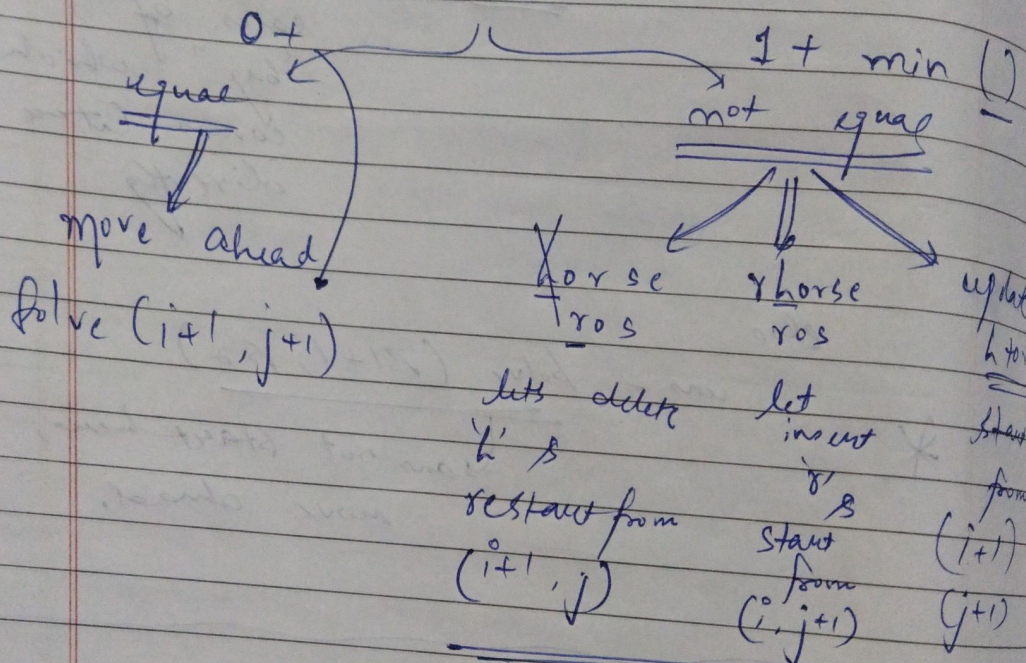
DATE: / /
PAGE:

⇒ Edit distance :

So we have to make our word1 == word2.

⟶ One rule I can observe in String qus & checking equalities is we always move forward like check the $s_1[i]$ & $s_2[j]$

we have 2 case



DP on Stocks

DATE: __/__/____
PAGE:

1. Best time to buy & sell stock (I)

→ Here we are given an arr. of the prices of stock of size n representing n days.

→ In the first form we are supposed to make only 1 transaction and maximise our profit.

→ We can only first buy a stock on i^{th} day & then sell it on $(i+1)^{th}$ to n^{th} day to maximise our profit:

prices[j] - prices[i]

→ if we try to do this just using linear searching then in that case:

we can move in a line a keep mainting the min number encured yet & @ every point we try to maximise the profit.

[7 1 5 3 6 4]

```
for (i=0 ; i<n; i++){
    mini = min(mini, arr[i]);
    profit = max(profit, arr[i]-mini);
}
```

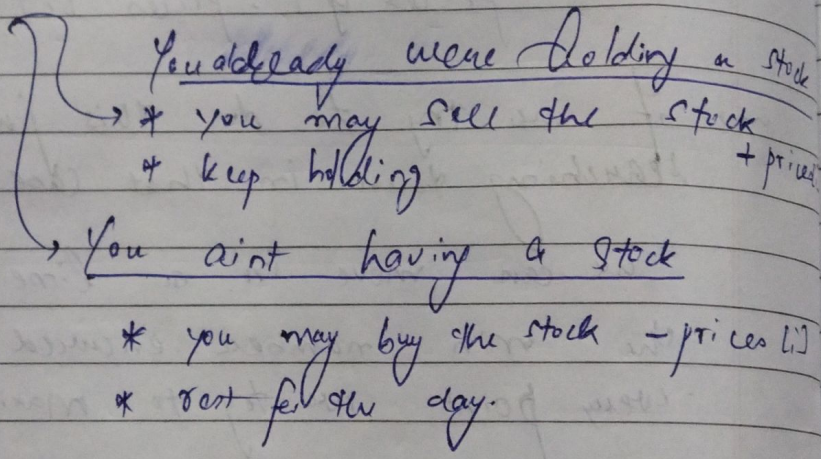
in this question we didn't really needed anything like DP as we can just start the mini yet & we have goods to go.

---

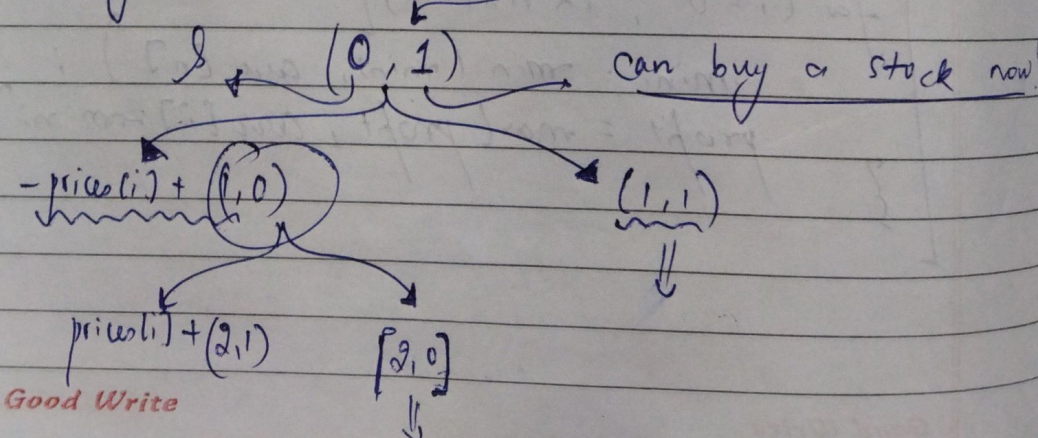
2. Best time to buy & sell Stock II

Here is the point where we can generalize the stock problems.

See if you thing about it @ every point we have like some 4 viable cases available



In the beginning when we start of we are holding no stock thus



Good Write

base case will be

```
if (s == n) { return 0; }
if (allow){
    return max(solve(!allow, s+1) + (-prices[i]),
               solve(allow, s+1));
}
return max(solve(!allow, s+1) + prices[i],
           solve(allow, s+1));
```

O(c) here since for a point we just require 2 stored variables i.e.

* solve(!allow, s+1), &
* solve(allow, s+1)

3. There is a max on the amount of transactions you can do

Best time to buy & sell stock III

Here what the problem is that there is only a limited number of transactions you can make

both buying & selling

Thus we can just keep track of the number of transactions left to us. or no. of trans" available.

Base

transaction_count == 0 → return 0;

ind >= n → return 0.

→ @ time of (not holding a stock)

(1)

buying

price[i] + (ind+1, 0, trans) (ind+1, 1, trans)

→ holding stock

(0)

price[i] + (ind+1, 1, trans-1) (ind+1, 0, trans)

---

4. for K transactions possible

just change the count to k.

rest all is same.

5. Buy & Sell Stocks with Count down period

for a cool down period after selling we just have to do

solve (s+k, !allow) + prices[s]

rather than s+1. thats all.

6. Transaction fee

Its applied only once for a completed transaction thus just @ the time of selling

* solve (s, !allow) + prices[s] - fee.

---

* Once generalized its all - mighty now.

Good Write

DP on "LIS"

∴ Leetcode Soln.

→ longest Inc. Subsequence
- order matters
- subsequence[i] > subsequence[i-1]

→ Brute force:
Lets try to make every possible subsequence and return the longest of those.

TC: 2^n SC: O(1)

→ Recursive app.:
First hint is the word subsequence → take / not take

So now I have to visualise what happens when we take & not take.

Not take: return 0 + solve(s+1, cap-index)

take: return 1 + solve(s+1, s)
- option available if & only if nums[s] > cap

Our app is to set a cap index or prev. index in the subsequence as we can only take an element if it is following the inc. order of things.

```
if (s >= n) return 0;
if (cap_index == -1 || nums[s] > nums[cap_index]) {
    maxi = max(maxi, 1 + solve(s+1, s));
}
maxi = max(maxi, solve(s+1, cap_index));
return maxi;
```

Good Write

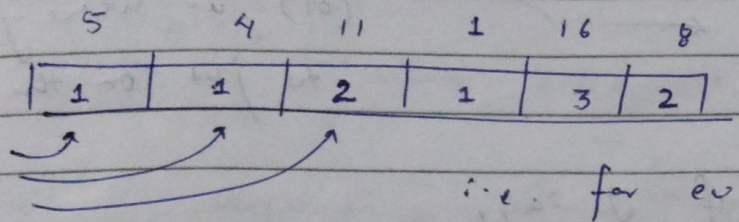
→ we can obv. memoize it.

by using a 2D array of size $(n) \times (n+1)$

↓ rep^n of -1.

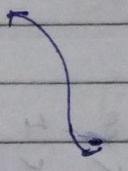
→ Tabulation : (To print the LIS)

The thought process is to make a 1D array to store the length of the LIS formed till the i^{th} index eg:



i.e. for every index i we check its all prev elements if they are smaller than it then we add its LIS count.

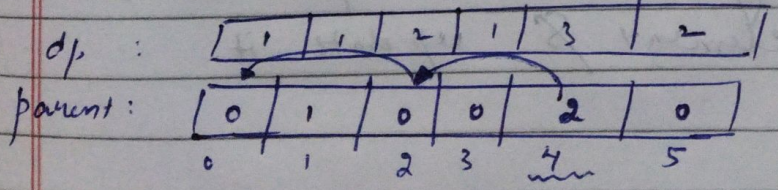
eg: for 16 ⟹ LIS(16) = max(LIS(11)+1, LIS(5)+1, LIS(4)+1, LIS(1)+1)



tho it still take $O(n^2)$ time & $O(n)$ space

↳ both achievable by space opt^n too.

we can here keep track of prev by having another vector



[1 D-DP use krna]

Binary Search approach

DATE: __/__/____
PAGE: _____

[1 7 8 4 5 6 -1 9]

normally when doing this & making LIS what we can do is.

[1, 7, 8, 9 ←
[1, 4, 5, 6, 9 ←
[-1, 9 ←

we just made diff strings to fit (or) we were just allow to put on the top.

but what if I say, since I am only considered with the length. I can merge them to get my answer in less space.

how?

1 7 8 4 5 6 -1 9
✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

[✗ ✗ 8 6 9
-1 4 5

length = 5

we basically traverse & find the lowerbound for each element & update it.

Good Write

→ order is irr element

Longest Divisible Subset

DATE: __/__/____
PAGE: _____

∴ Leet code Sol^n

One thing I will say is:
if the question is asking for a subset and not a subsequence, then it is good hint to try sorting the array to convert this problem to a subsequence type of problem where we can go by pick / no pick

→ first will get one divisibility rule:

if a > b > c then

a % b == 0
&
b % c == 0 ⟶ a % c == 0

thus if the array is sorted already.
then in that case we just have to check by dividing with the top element if its divisible or not

thus now this is literally a LIS question variant as
Longest Div. Subsstrig

we make a 1d dp [ ]

and dp[i] store the longest div subset till the ith index.

(& you maintain parent)

Good Write

Longest String Chain

If you look @ it, you can say clearly that this is a question of LIS on strings but in subset fashion.

thus first will sort it based on length.

then for next we do the exact same steps but just the condition for checking on index j will be

→ isPre(words[i], words[j])

we have to check if (s_1, s_2) string are such that we add 1 char to $s_1 \longrightarrow s_2$

we can just use 2 pointers & keep moving forward.

Good Write

Longest Bitonic Subsequence

Well from the experience of LIS we can see one thing that first we can see a LIS and then see LDS → decreasing.

Thus we just have to find LIS & LDS @ every i value,

LIS → 0 to ith

LDS → from the ith to n-1th

ans =

$$\left(\begin{matrix}\text{Size} \\ \text{of} \\ \text{LIS}\end{matrix} + \begin{matrix}\text{Size} \\ \text{of} \\ \text{LDS}\end{matrix} - 1\right)$$

Good Write

Number of LIS

This question reminds me of that graphs question where you had to count the number of paths somehow i.e. we used to maintain a vector of number of times the node is reached using min distance.

if a new min dist. came → Set to cnt[new]

else add the cnt[new]

Similarly here

we again use the 1D-DP approach but also maintain a cnt variable array,

So now when

```
if(arr[j] < arr[i]){
    if(dp[i] == dp[j]+1){
        cnt[i] += cnt[j];}
    dp[i] < dp[j]+1{
        cnt[i] = cnt[j]
        dp[i] = dp[j]+1.
```

to return we add all the nodes where dp[i] == maxi

↳ as ans.

Good Write