

40

DP PROBLEMS
INTUITIONS AND
PATTERN

STRIVER

for you

NOTES ->

Dynamic Programming

By Page

(16.2)

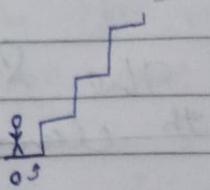
Climbing Stairs :

n stepped ladder to reach the top

\sqrt{n}

$x \leftarrow \text{unpos}^n$

destⁿ : n



$\rightarrow (x) \rightarrow (x+1)$

or

$\rightarrow (x) \rightarrow (x+2)$

} options.

Base case : if

$x = n-1$ [return 1]

$x = n-2$ [return 1]

$x = n$ [return 0]

base : if $x = n \rightarrow$ return 1.

frog jump :

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

$$\Rightarrow (x) \xrightarrow{\text{solve}} (x+1) + \text{abs}(x+1 - x)$$

$$(x) \xrightarrow{\text{solve}} (x+2) + \text{abs}(x+2 - x)$$

base if $x = n-1 \rightarrow$ return 0;

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 i^{th} element
 or ~~but~~ $i+1^{th}$ element
 or not

If you do!

then you don't get
 the choice of
 $i+1^{th}$ element thus
 not taking it

thus $\text{Val} + (\beta+2)^{th}$

if not

then same
 in
 repeated
 $\beta+1^{th}$

$$(x - \beta) + \alpha + (\beta+2) \rightarrow (i) \leftarrow$$

$$(x - \beta) + \alpha + (\beta+2) \rightarrow (i)$$

16.3

#

Ninja's Training

DATE: / /
PAGE:

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

Run | Fight | Practise

Thus obv. we can replicate the part 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 done activity and consider during recursion

we can have:

Solve(s, last, points) {

for (i=0; i<3; i++) {

if (i == last) { continue; }

maxi = max(maxi, points[s] + solve(s+1, i, points))

}

Base: if s > e → return 0;

We can ofc make a dp for the same
of size $3 \times n$.

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

→ BT: we will kinda do that tho, we want 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 $(x+1, y)$ $\xrightarrow{\text{States}}$ $(x, y+1)$

→ DFS: Technically recursion is itself DFS only.

→ Unique Paths 2

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

return 0 as a base case.

#

Min path Sum in grid

DATE: _____
PAGE: _____

The approaches that comes to mind are

→ Recursion / backtracking:

Since we are just going R & D thus we don't 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 $[x][y]$

→ Dijkstra

Or you can use the plain old priority queue with dijkstra's algo with in to do this work.

DS used \Rightarrow distance matrix
 \Rightarrow priority queue

Min falling sum :

DATE: / /
PAGE: / /

here we must first fill the dp table
for all the boxes, ans will be the
min of the top row.

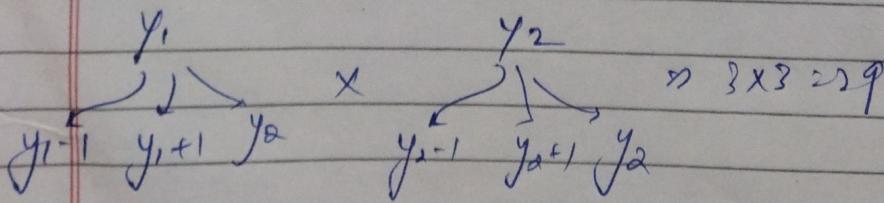
that's all.

Chocolate picking

It's honestly quite an easy question,
you can go from $x, y \rightarrow x+1, y-1 \quad | \quad x+1, y \quad | \quad x_2$
only.

Since we have 2 beans thus the
would be x_1, x_2, y_1, y_2 as the var.
in every step, but wait $x_1 = x_2$ also
as @ every step we are always doing $\underline{x_1}$.
Thus there are total 9 cases after that

$$\underline{x_1, x_2 \rightarrow x_1+1, x_2+1}$$



thus a 3D-DP of (x_1, y_1, x_2)

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 don't have to maintain the sum for both of for the subset sum is $\frac{N}{2}$ then for other its $\frac{N}{2}$.

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. At first glance you may feel like trying to use a new recursion app. for it & it works too for a tonne but gives TLE.

then my app. was:

finding the min abs. sum if arr[8] is either taken or not - taken but we also have to send off a count variable to the same. for more volatile solution?

The following of ideas, another one is to just look if a specific target sum is possible for a specific array or not to go from 0 to return.

Just check if its true then we take the min diff out of all those options.

dp[0][i] → give you the full row of these target values

in one time only. Note it will be

$$0 \left\{ \begin{array}{l} \text{if } \star (\text{sum}) \\ \text{else } \end{array} \right\} \text{time}$$

* It will give false if you inc negative numbers as there can will get ~~the~~ nothing → no which can't be sp. by dp so either we shift them by a number or find another approach.

Lc qn: 139

Word Break

Brute approaches

1. Normally use a Trie & do the deal of breaking by modifying the search function accordingly

Intuition:

Close off by the character we start from S & keep adding the string matches we find if its possible to get answer if we consider that as one word thus do solve (~~str~~)

Ex:
lin
= apple fun apple
i carry]

Note: j, A₀) return char for parapp's.

Flow:

aaaa and word: did:

" " " " " " " "
thus we both send recursion but
Good Write also moving ahead to check for
small bigger matches.

4. Count Subsets with Sum k

DATE: / /
PAGE: / /

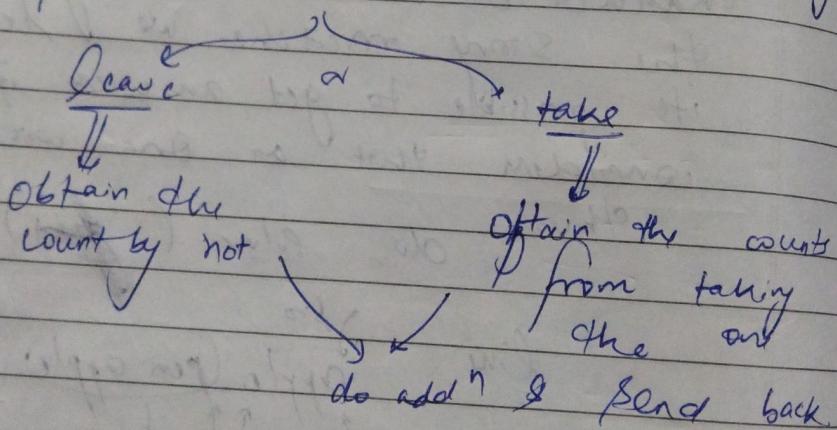
[1, 1, 2, 5]

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

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

(or)

→ Recursion may for a array all have 2 targets options for the i^{th} index



base:
$$\begin{cases} f.(k=0) \{ \text{return } 1; \} \\ f(k > e) \{ \text{return } 0; \} \end{cases}$$

→ Minimum Coins :

It's 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-var)
as all coins are ∞ in terms of

// 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 91

After doing min coins, ifc yr 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 order of numbers like

$$\left[[1, 2, 5] \right] \text{ target} = 5$$

$$\rightarrow \text{choose } 2 \text{ ans target} = 4 \quad (2, 5)$$

$$\begin{matrix} 2 \\ 3 \end{matrix} = \text{Ans: } \boxed{\begin{matrix} 2+2 \\ 1+1+2 \\ 1+1+1+1 \end{matrix}}$$

$$\rightarrow \text{choose } 1 \text{ ans target} = 3$$

$$\begin{matrix} 2 \\ 3 \end{matrix} \left[\begin{matrix} 1+2 \\ \hline \text{f f f} \end{matrix} \right]$$

$$\rightarrow \text{choose } 5 \text{ ans target} = 0 \quad \text{repeat}$$

$$\begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

→ try to visualize this i.e.

we already do consider the
 picking of 1 & 2 when we picked 1

~~Given~~ ~~Write~~ we just selected for index=1
 then we must start from 1

and not restart at 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 as all min coins 2
that we are picking but picking but
the option of having the same item
doesn't diminish 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 previous
& 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 $\text{text1}[i]$ & $\text{text2}[j]$ then

if $\text{text1}[i] == \text{text2}[j]$

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

Not

\rightarrow i^{th} element ($i, j+1$)
may be part
of another LCS

\rightarrow j^{th} can be ($j, i+1$)

none:

($i+1, j+1$)

we choose
& \max it.

ii) print the string :

other than dp of type int

make of type string \rightarrow rest is almost
same & understandable too.

Longest Common Substring

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

The reason we were using the strategy of
storing state (i_1, j_1) for LCS is the
fact as once a char is matched for
max LCS we will never take it into final
consideration always, not here.

We can find a substring but it may
may not be the best so How do you
get this done?

e.g. $\alpha = a \underline{b c j k l p}$

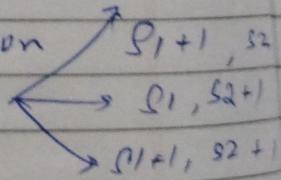
$\beta = a \underline{c j k p}$

here here are 2 components:

We first try to find the largest subst.
(common) from α & β . & store its
cnt

like here it will be 1 ("a")

then we have to send for recursion



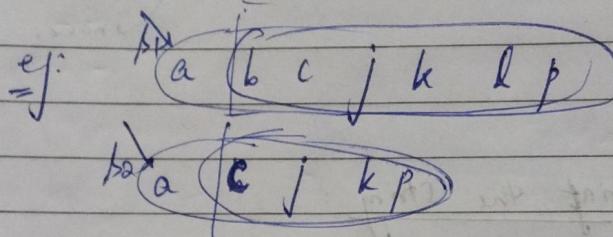
As now we should have both S_1 & S_2 in
the substring (also. containing cnt)
thus we find max of all & these 3
& sent max back.

Longest Common Substring

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

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

We can find a substring but it may
may not be the best so how do you
get this done?

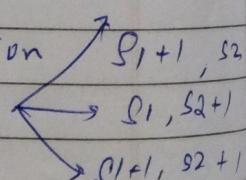


here here are components:

We first try to find the largest subst.
(common) from S_1 & S_2 & store its
cnt

like here it will be 1 ("a")

then we have to send for recursion



As now we should have both S_1 & S_2 in
the substring (or. cons in cnt)

thus we find max of all & these 3
are sent max back.

Longest Palindromic Subsequence

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

Lemma:

longest palindromic subseq
is same as

les (s, reverse of s).

M. Insertion Steps to make a string palindrome

So when we start with this question our first 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 $S[i] \rightarrow S[e-i]$

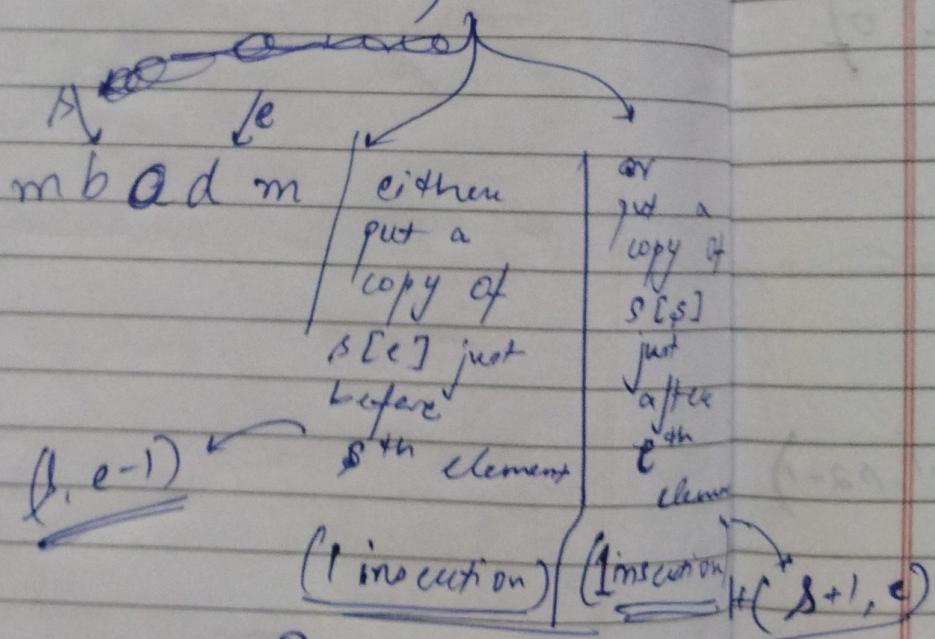
call recursion

(makes sense?)

$O+(S+1, e-1)$

they don't match

here is the weird part
we know some
that we gotta
do something about
this for sure.



Good Write

$mbad$
 sle

$mbad$
 sle

→ Delete Op's for two strings

easy peasy → find the lcs of both ⇒ that is the actual end game we wanna reach
↳ ans:

$$\leftarrow \text{size}_1 + \text{size}_2 - 2 * \text{lcs}(\text{size}_1)$$

* Similar qn on Min ASCII del sum for 2 strig

→ Shortest Common SuperSequence

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

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

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

e.g. m a d p a

t a s k p l e → LCS: ap

Good Write $\binom{m}{x} \otimes \binom{d}{sk} \otimes \binom{a}{l.e.} \rightarrow \underline{m t a d s k p a l e}$

Distinct Subsequences

This is a v. interesting qn

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

if we have

$$\rightarrow (ba^b g \ b a g) \quad b \\ \rightarrow (bag) \quad t$$

we will compare $t[0]$ with all char
of s .
& will do

$$\text{ans} += \text{solve}(s_1+1, s_2+1) \\ \text{if } \underline{s[s_1]} = \underline{t[t_2]}$$

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

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

i.e. for $s_1 = 0 \rightarrow$ run from 0 to $m-1$
 $s_2 = 0$
 \searrow
 $\text{char}_c = t[t_2]$

$$\text{if } \underline{s[s_k]} = c \quad \left\{ \begin{array}{l} \text{ans} += \text{solve}(s_1+1, s_2+1) \\ \dots \end{array} \right.$$

The problem was in the iteration we were doing in that s_1 to s_{n-1} of 's'.

So we start our s_1 from 1
 s_2 from 1

* $\text{if } [s(s_{1-1}) = s + (s_{0-1})] \text{ then}$

ans = solve(s_1+1, s_2+1) + solve(
 s_1+1, s_2)

'as' standing
ahead covers all the cases of 'bag' which comes later directly

else

ans = solve(s_1+1, s_2)

ans not start here,
move ahead.

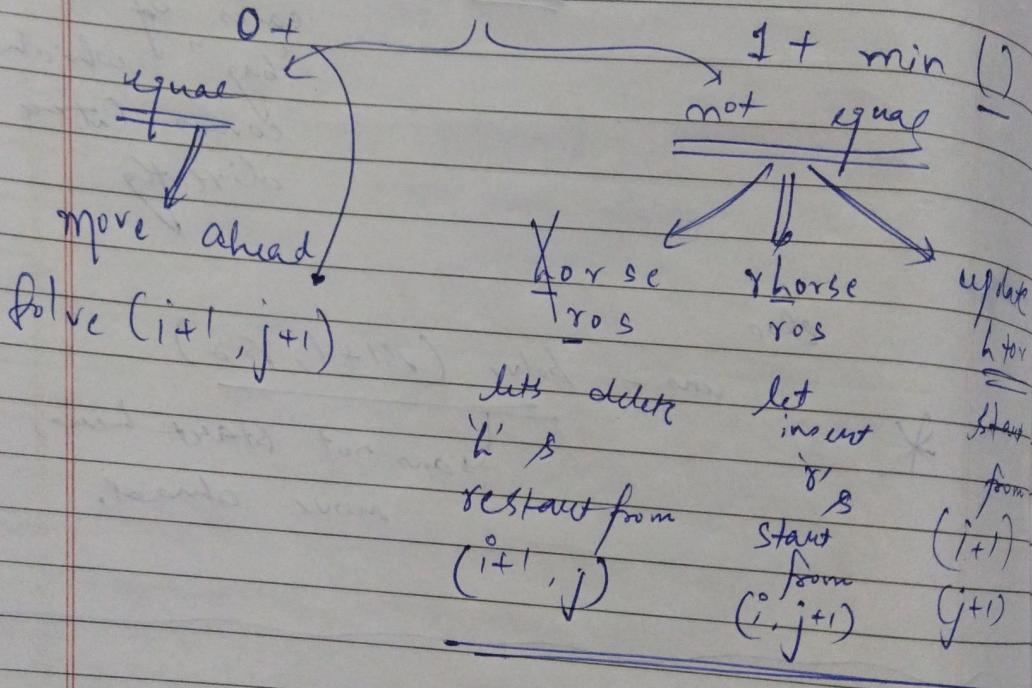
Edit distance :

So we have to make our words.

One rule I can observe in string ops & checking equalities is we always move forward like

check the $s_1[i]$ & $s_2[j]$

we have 2 cases



DP on Stocks

DATE: _____
PAGE: _____

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

- Here we are given an array 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 next day to maximise our profit.
 $\text{prices}[j] - \text{prices}[i]$

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

we can move in a line & keep maintaining the min number encouned 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 need anything like DP as we can just start the mini yet & we are going to go.

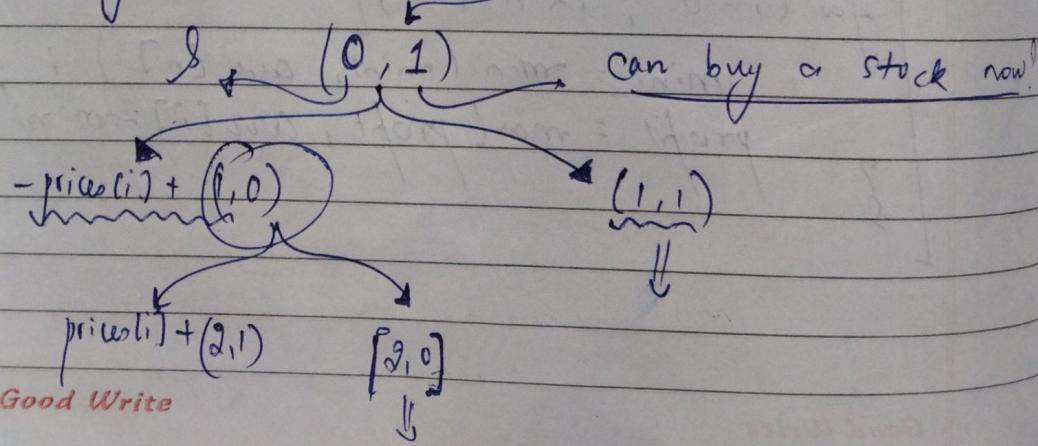
a. Best time to buy & sell Stock II

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

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

- { You already were holding a stock
 - * you may sell the stock + price
 - * keep holding
- , You aint having a stock
 - * you may buy the stock - prices[i]
 - * rest for the day.

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



base case will be

if ($s \geq n$) { 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, $s+1$));

Or here since for a point we just require 2 stored variables i.e.

* solve(allow, $s+1$), s

* 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 trans actions you can make

both buying & selling

thus we can just keep track of the number of Good Write Available trans left to us or no. of trans available

base

$\text{transaction_count} == 0 \rightarrow \text{return } 0;$
 $S >= n \rightarrow \text{return } 0.$

ans. @ time of (not holding a stock)

①
 buying —————> - $\text{price}[i] + (S+1, 0, \text{trans})$ $(S+1, 1, \text{trans})$

ans holding a stock
 ②
 $\text{price}[i] + (S+1, 1, \text{trans})$ $(S+1, 0, \text{trans})$

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

$$\text{Solve } (\underline{s+k}, \text{allow}) + \text{prices}[s]$$

rather than $s+1$. That's all.

6. Transaction fee

It's applied only once for a completed transaction thus just @ the time of selling

$$* \text{Solve } (s, \text{allow}) + \text{prices}[s] - \text{fee}.$$

* Once generalized it's all - mig lithy now.

(minimum costs \leq costs of buying)

$\Rightarrow (2, 12) \text{ costs } + 2, \text{ costs } \geq \text{ costs}$

(minimum costs \leq costs of buying)

DP on "LIS"

SP DATE: / /
PAGE: / /

→ Let's do Soln

→ longest Inc. Subsequence

order matters

Subsequence [i] > Subsequence [i+1]

Brute force:

Let's 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] > nums[cap-index]

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;

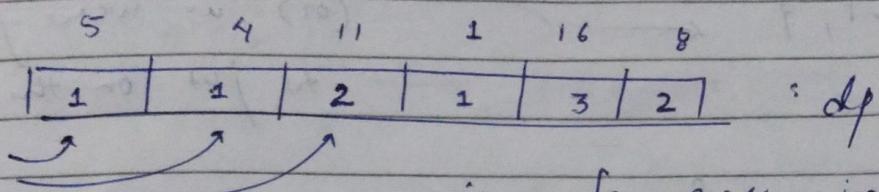
→ we can also memorize it.

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

\downarrow
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 e.g:



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

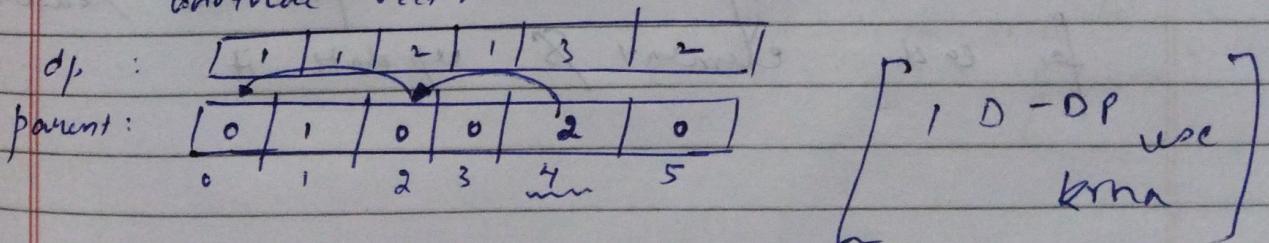
$$\text{e.g. for } 16 \Rightarrow \text{LIS}(16) = \max(\text{LIS}(11)+1, \text{LIS}(5)+1, \text{LIS}(4)+1, \text{LIS}(1)+1)$$

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

↳ both achievable by

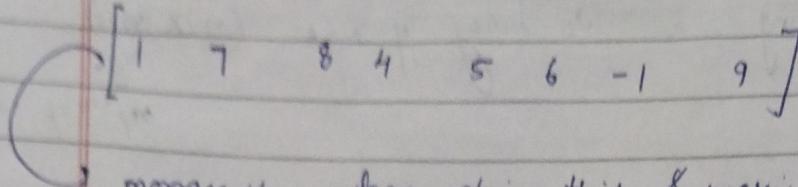
space opt^n too.

we can hence keep track of prev by having another vector

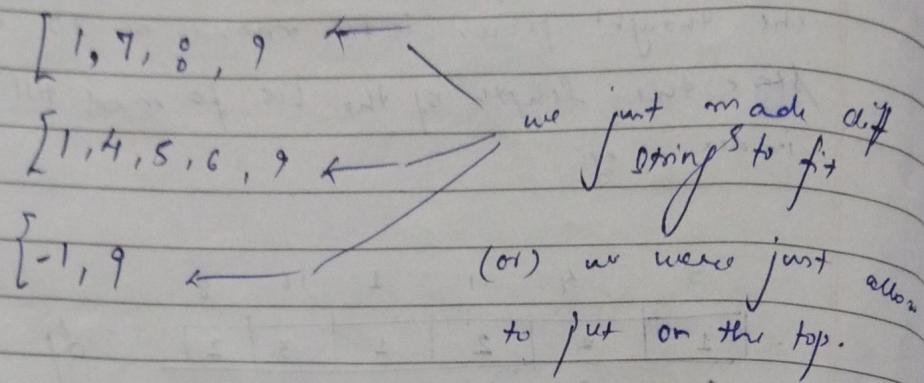


Binary Search approach

DATE: _____
PAGE: _____



normally when doing this & making what we can do it.



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

how?

✓ 1 7 8 4 5 6 -1 9
✓ ✓ ✓ ✓ ✓ ✓ ✓

[X 1 8 6 9
-1 4 5] length = 5

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

Longest Divisible Subset

LeetCode Sol

One thing I will say is:

If the question is asking for a subset and met a subsequence, then it is good 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

$$a \cdot b = 0$$

$$\xrightarrow{b} a \cdot c = 0$$

$$b \cdot c = 0$$

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

Longest Dv. Subseq.

we make a 1d dp

and [dp[i]] store the longest div subset till the i^{th} index.

(if you maintain parent)

Longest String chain

DATE:

PAGE:

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

thus first will sort it based on lengths.
then for next we do the exact same
steps but just the condition for
checking on index j will be

is pre(words[i], words[j])

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

we can just use 2 pointers & keep moving
forward.

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 \downarrow decreasing.

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

\uparrow from the
 0 to ith \uparrow to ith to m-1th

ans =

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

(LIS + LDS)

((LIS) + (LDS))

LIS + LDS = LIS + LIS

44396 > 43496

LIS = LIS

44396 = 43496

so LIS = 44396

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 \rightarrow Set to $cnt[new]$

else add the $cnt[new]$

similarly here.

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

so now when

$\{ arr[j] < arr[i] \}$

$\left(\begin{array}{l} f(dp[i] := dp[j+1]) \\ cnt[i] += cnt[j]; \end{array} \right)$

$dp[i] < dp[j+1]$

$cnt[i] = cnt[j]$

$dp[i] = dp[j] + 1$.

to return we add all the nodes when

$dp[i] := maxi$

\downarrow , as ans.