DP

↳ Overlapping sub-problems

↳ bigger problem should be dependent on smaller problem

(Ik) DP can't be applied for recursion on the way up type problems.

Fibonacci series

Approaches
* Recursive
* Memoized
* Tabulation

→ Decide storage dimension (varying variables)
→ check if task is precalculated
↳ If not store value after calculation

$$fib(n) = fib(n-1) + fib(n-2)$$

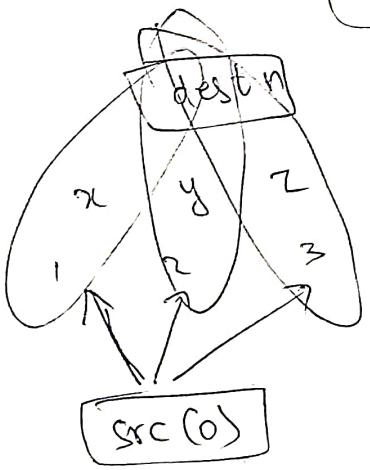
Recursive
TC: Exponential
AuxilarySC: O(1)

Memoized & Tabulation
TC: O(n)
SC: O(n)

Climb Stairs (3 steps)

no. of ways

* Recursive
* Memoized
* Tabulation



src → dest

Base case:
To move from src (0) to dest (0), we have 1 way, i.e., Don't move

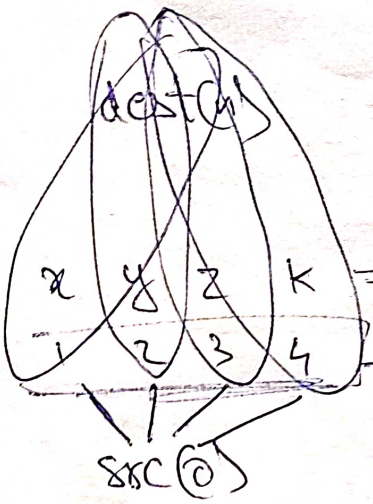
Recursive
TC: exponential
SC: O(1)
↓
Auxiliary

dp[i] ⇒ no. of ways to reach ith stair

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

Climb stairs with variable jumps

* Recursive
* Memoized
* Tabulation



cur src = x+y+z+k ways

→ Max jump from cur src

Base case:

src(0) to dest(0) => 1 way

Recursive

TC : Exponential(n^n)

Aux. SC : O(1)

Memoized & Tabulation

TC : $O(n^2)$

SC : O(n)

dp[i] => total no. of ways to reach destination starting from i^{th} index.

Target Sum subsets

Tabulation | TC $O(n^2)$ | SC $O(n^2)$

$(n+1) * (tar+1)$

M:

dp[i][j] → arr(0 to i-1), j → ?

Hack: For tabulation DP

check no. of variables which are varying in each recursion, from recursive code. those no. of variables may be used for dp storage definition 1D, 2D

Two choices

dp[i][j] → dp[i-1][j] [∵ j → represents tar]

dp[i-1][updated tar]

Space optimization: Use '2' [1-D DP] (To maintain previous & current state)

Recursive TC $O(2^n)$

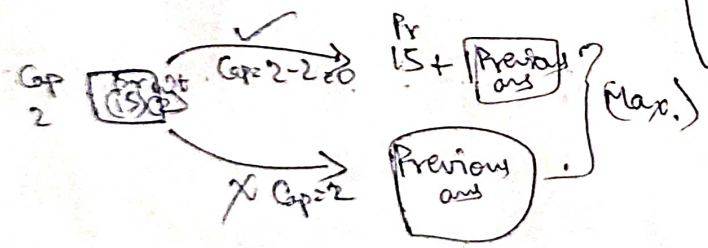
Ideal TC for tabulation method: $O(tar * n)$

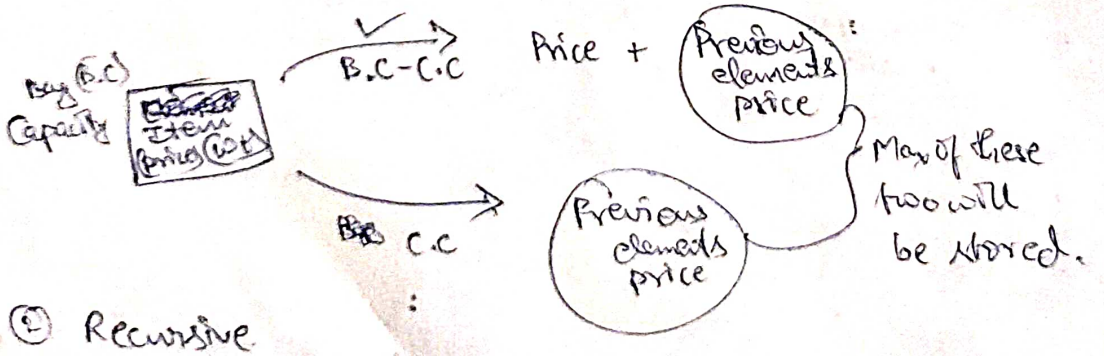
Zero One knapsack:

Note: You can never use memoization or DP on recursion on the way up.

① Tabulation:

dp[i][j] → arr[0 to i-1], cap - j, max profit?





② Recursive

③ Memoized

Coin change combinations

ways: (1) Recursive → Here TC ∞ (amt → ∞ because ∞ coin supply)
(2) Memoization
(3) Tabulation

(1) For recursive approach In this problem, if the array contains element 0, then the amt doesn't decrease and recursion goes on (stack overflow).

Tabulation: 1-D DP (Space Optimized) TC: $O(n^2)$

dp[i] → Total no. of combinations to pay i^{th} amount.

Hint to solve: Put effects of each coin on all amount iteration. Thus combination achieved

Coin change permutations

Tabulation: Same like in above tabulation combinations just ⇒ (1)

Hint to solve: Put all coins effects at one particular amt index one-by-one.

Recursive: put a loop of all coins on the amount possible at each levels.

(1) Refer next page for recursive diagram for combinations and permutations

0	1	2	3
2	3	5	7

Amount = 10

Permutations: to form permutations formula: $nC_r \times r! = nP_r$

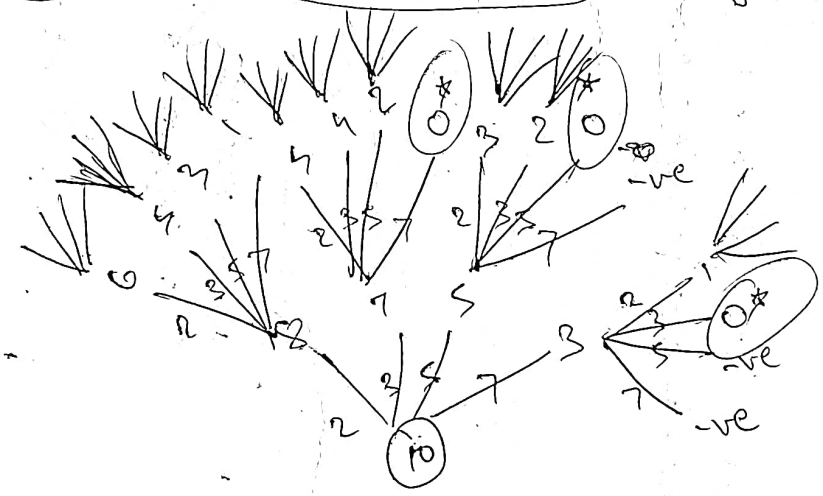
Here $r = 3$

To make combinations

$$\frac{nP_r}{r!}$$

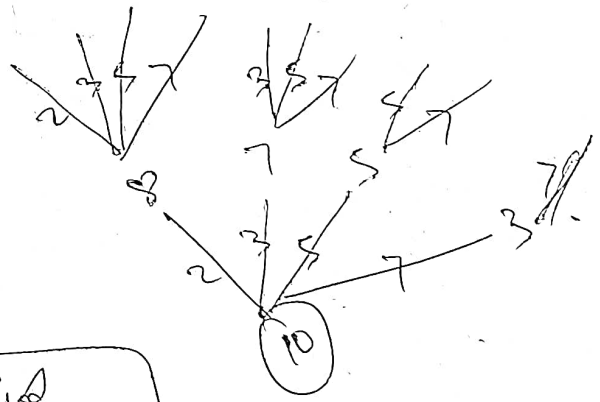
2,2	3,2	5,2	7,2
2,3	3,3	5,3	7,3
2,5	3,5	5,5	7,5
2,7	3,7	5,7	7,7

→ this give combinations



Combinations nC_r

[; this is derived from permutation ;]



Another method (combinations)

Count Binary strings:

Tabulation :-

1	2	3	4

Ending at 0 [0];

Ending at 1 [1];

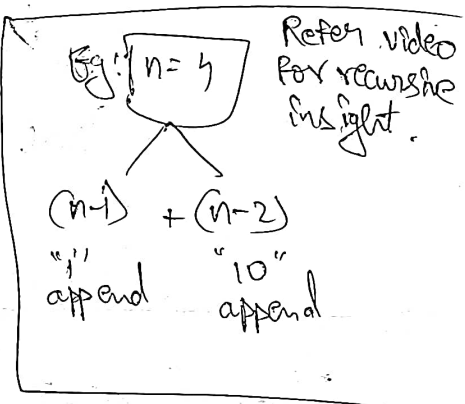
2	3	5	8	13

TC : O(n) SC : O(n)

This thing forms a fibonacci pattern. [:. this prblm. is a part of modified fibonacci.]

Recursive: You can use the fibonacci formula itself since after all it forms a fibonacci series. So use this formula

$$f(n) = f(n-1) + f(n-2)$$



Refer video for recursive insight.

Arrange Buildings

Same approach & ideology like count binary strings.

Tabulation: Return total ways of one side multiplied (total * total)

TC : O(n) & SC : O(n)

Count Encodings

ways

1. Recursive
2. Memorized (Use HashMap, put a string in map with corresponding any.)

   Better memoized : Use index in recursion: Array based with index
3. Tabulation

dp[i] = Total no. of encoding till ith index.

Some where you have to check with this formula dp[i] = dp[i-1] + dp[i-2]

Floor Tiling:

(2 * 1)

I can place tile in horizontal direction or vertical direction

Vertical → (n-1)
Horizontal → (n-2)

f(n-1) f(n-2)
'v' 'h'

$$f(n) = f(n-1) + f(n-2)$$

$$dp[i] = dp[i-1] + dp[i-2]$$

Floor Tiling

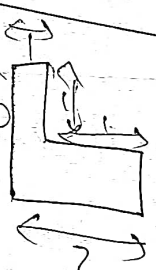
(n * m)

Same approach $f(n) = f(n-1) + f(n-m)$

$$dp[i] = dp[i-1] + dp[i-m]$$

Floor Tiling-2: Domino

Tromino



Here, we need to take care of both fully filled tiles and partially filled tiles.

Fully filled

$$f(n) = f(n-1) + f(n-2) + p(n-1)$$

* Add 'v' in (n-1)
* Add 'h' in (n-2)
* Add a tromino in partial filled of p(n-1)

Partially filled

~~$p(n) = p(n-1) + (2 * f(n-2))$~~

$$p(n) = p(n-1) + 2f(n-2)$$

* Add a tromino in fully filled of f(n-2)
* Add a tromino in partial filled of p(n-1).

$f(n) = f(n-1) + f(n-2) + p(n-1)$ — (1) $\Rightarrow p(n) = p(n-1) + 2f(n-2)$ — (2)

$\Rightarrow p(n-1) = f(n) - f(n-1) - f(n-2)$ — (3)

Subs. (3) in (2),

$$p(n) = f(n) - f(n-1) - f(n-2) + 2f(n-2)$$

Again $n \to n-1$,

$$p(n-1) = f(n-1) - f(n-2) - f(n-3) + 2f(n-3)$$

$$\Rightarrow p(n-1) = f(n-1) - f(n-2) + f(n-3) \text{ — (4)}$$

Subs (4) in (1),

$$f(n) = f(n-1) + f(n-2) + f(n-1) - f(n-2) + f(n-3)$$

$$\boxed{f(n) = 2f(n-1) + f(n-3)} \text{ — (5)}$$

$$\boxed{dp[i] = 2\,dp[i-1] + dp[i-3]}$$

Modular arithmetic used in this prblm,

$$\boxed{(a+b)\%m = (a\%m + b\%m)\%m}$$

$10^9+7 \Rightarrow$ Prime no.

After modular arithmetic formula will be,

$$\boxed{dp[i] = \left((2 * dp[i-1])\%m + (dp[i-3])\%m\right)\%m}$$

Modular arithmetic

$m = 10^9 + 7$

range $\Rightarrow$ $0 \longrightarrow 10^9 + 7 - 1$ (Integer)

∴ 1 byte = 8 bits

1 Integer $\rightarrow$ 4 bytes $\Rightarrow$ 32 bits

↓

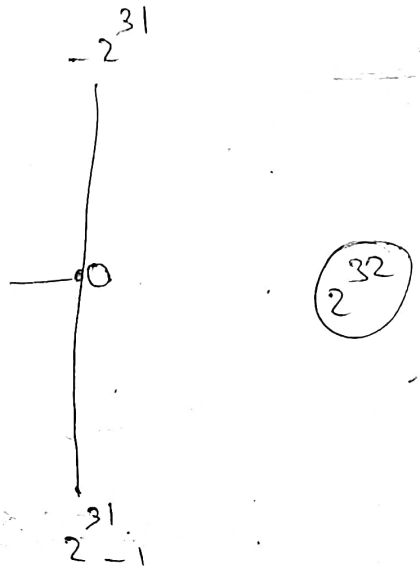
8 bits

$(2^{10}) \rightarrow 1024 \simeq 10^3$

Cube both,

$(2)^{10\times 3} \simeq (10)^{3\times 3}$

$2^{30} \simeq 10^9$



(LIS) Length of Longest Increasing Subsequence

1. Recursive
2. Memoized
3. Tabulation

Substring/subarray $\Rightarrow \frac{n(n+1)}{2}$

Subsequence/subset $\Rightarrow 2^n$

dp[i][j] $\Rightarrow$ arr(0 — i), previndex $\rightarrow$ j

length of LIS $\rightarrow$?

Recursive 2 approaches

Better approach: Normal — TC $O(2^n)$ ↓ (With 2D DP) Memoized

with looping — TC $O(n * n^n)$ Exponential ↓ Memoized (with 1D DP)

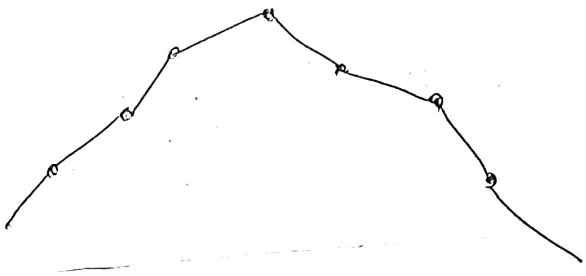
Tabulation:

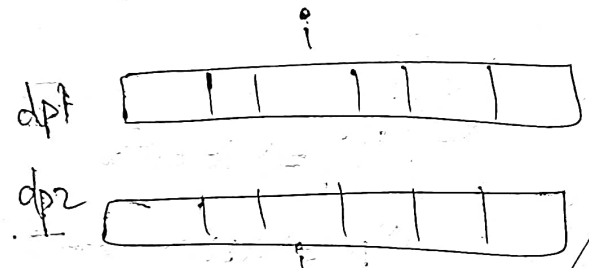
dp[i] => length of longest increasing subsequence ending at i^{th} index.

TC $O(n^2)$

SC $O(n)$

Longest Bitonic subsequence (Variation of LIS)



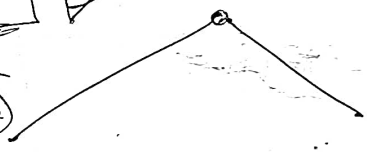


Longest Increasing subsq. from Left to Right

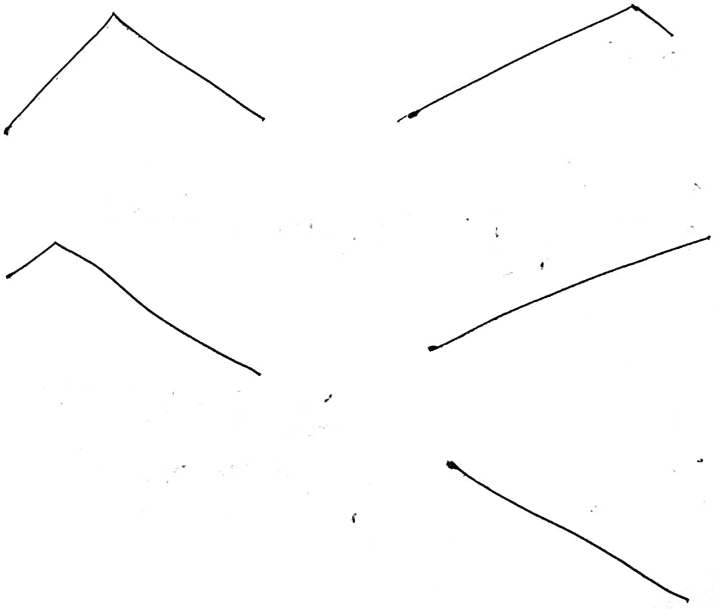
LIS from right to left

$dp1[i] + dp2[i] - 1$

(Because there can be only one peak)

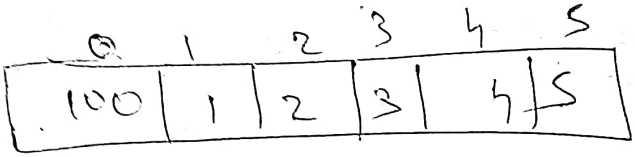


Types in Bitonic sequence pattern:



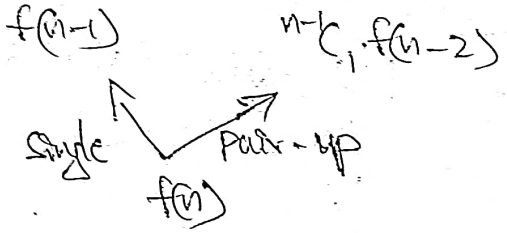
Max Sum Increasing Subsequence (Variation of LIS)

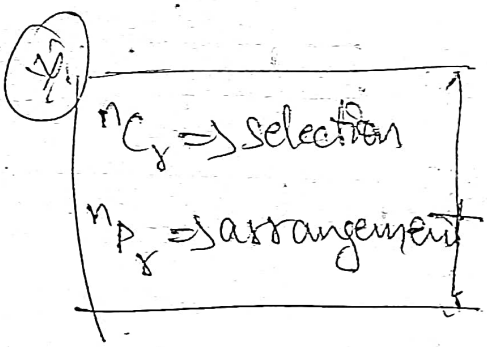
Wrong case Initial thought: Thinking that longest increasing subsq. will have max sum. To break this though refer example, below



dp[i] → Max. sum increasing subseq. ending at i^{th} index

Friends pairing

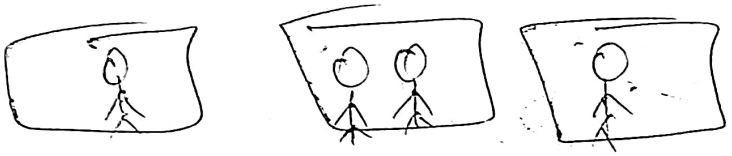




$f(n) = f(n-1) + (n-1) * f(n-2)$

$dp[i] = dp[i-1] + (i-1)\,dp[i-2]$

Partition into subsets



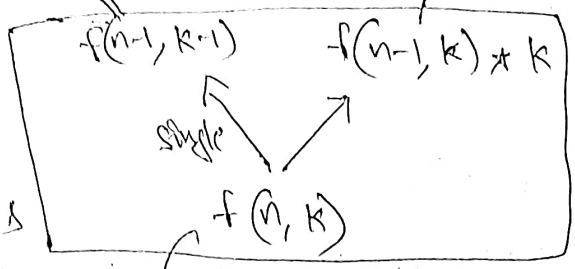
⇐ K = subsets

n = 4 => 1, 2, 3, 4

Example: settle 4 guys in 3 rooms

No. of ways to arrange 'n-1' elements in 'k-1' subsets such that no subset is empty

No. of ways to arrange 'n-1' elements in 'k' subsets such that no subset is empty.



n -> no. of elements

k -> no. of subsets

[No. of ways to arrange 'n' elements into 'k' subsets such that no subset is empty].

~~f(n,k) = f(n-1)~~

$$f(n, k) = f(n-1, k-1) + f(n-1, k) * k$$

$$dp[i][j] = dp[i-1][j-1] + dp[i-1][j] * j$$

Permutations and combinations Gyan : Refer YT (Don't memorise)

$${}^{n}C_{r} = \frac{n!}{r!\,(n-r)!}$$

Combinations (select)

$${}^{n}P_{r} = \frac{n!}{(n-r)!}$$

Permutations (Arrange)

'OR' → Addition (+)

'AND' → Multiplication (*)

Maximum Sum Non-adjacent sub elements

(Must handle negative values also)

Eg:

-3	2	5	-8	-7	3	11
0	1	2	3	4	5	6

1) Recursive 1
2) Memoized 1 (2D DP)
3) Recursive 2
4) Memoized 2 (1D DP)
5) Tabulation 1 (1D DP) dp[i] => Maximum sum with non-adjacent elements till ith index.
   ↳ TC O(n)

~~dp[i] = max(dp[i-1], (dp[i-2] + arr[i]))~~

dp[i] = max(dp[i-1], arr[i-1] + dp[i-2])

6) Tabulation 2 (Here we take 2 DPs)

	0	1	2	3	4
arr	-5	3	2	-4	6

Maxsum Considering elements till ith including current element (ith) → incl(dp)

Maxsum Considering elements till ith excluding the current element (ith) → excl(dp)

incl[i] → max sum of non-adjacent element till ith index & including arr[i].

excl[i] → max sum of non-adjacent element till ith index & excluding arr[i].

incl[i] = arr[i] + excl[i-1]

excl[i] = max(incl[i-1], excl[i-1])

You can also use just variables (previous state), and optimize space to O(1)

HW problem

Max sum of non-adjacent elements a subsequence

Max sum of non-adjacent elements

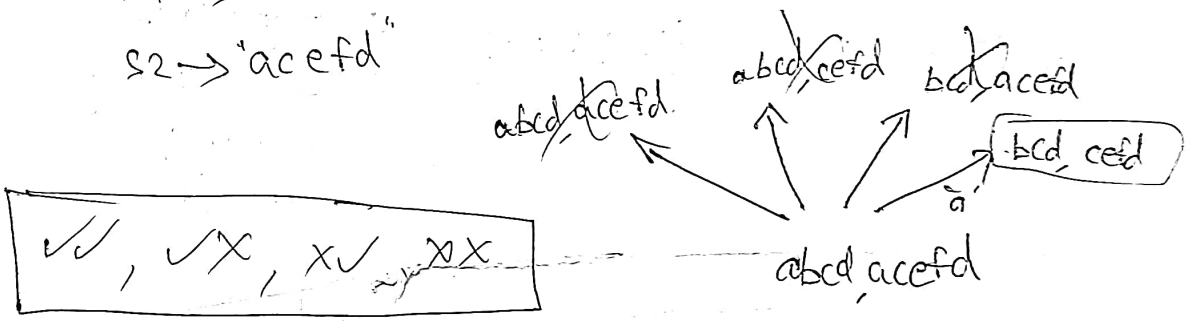
H.W

Max sum of a subsequence having no-adjacent elements & atmax * elements $(K <= n)$

Longest Common Subsequence

s1 → "abcd"

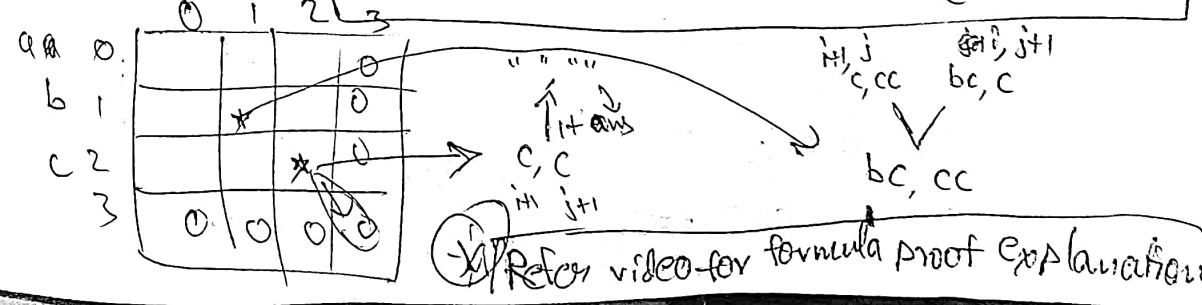
s2 → "acefd"



1. Recursive
2. Memoized
3. Tabulation 1 → Direction to solve (top to down)
4. Tabulation 2 → Direction to solve (Bottom to up)

Refer video to visualize

dp[i][j] ⇒ s1 (i to last), s2 (j to last) ⇒ LCS



Refer video for formula proof explanation

Longest Palindromic Subsequence:

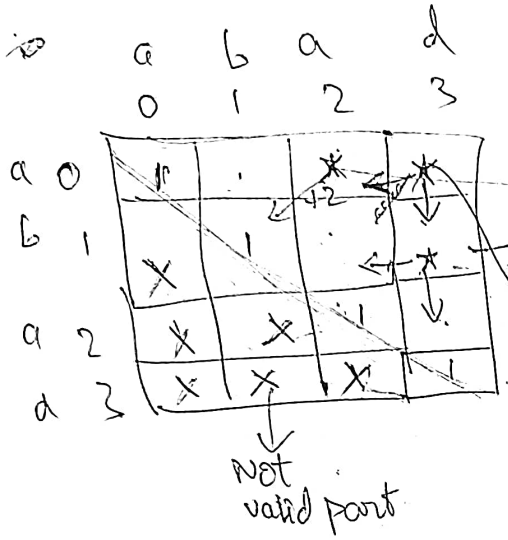
1. Recursive
2. Memoized
3. Tabulation

Another approach

Given str, S1 = ⬭

Reverse S1, S2 = ⬭

Now calculate longest Common subsequence



Refer video for clarity & direction to solve

[You have to travel diagonally top to bottom moving towards top right corner]

dp[i][j] ⇒ str(i, j) LPS of

Count Palindromic Substring | ~~Longest~~ Longest Palindromic Substring

Both of these problems uses the same solution of longest palindromic subsequence ("with some check modification conditions").

[Can also be done recursively by calling @ 3 different calls: check video]

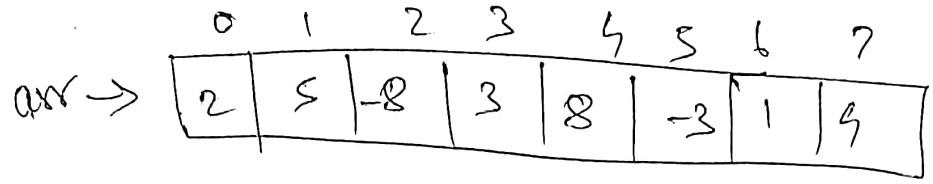
ch1 ≠ ch2 ⇒ false

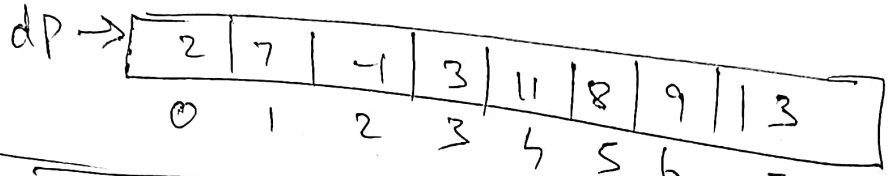
ch1 == ch2 → m == true ⇒ true

ch1 == ch2 → m == false ⇒ false

Max sum SubArray (Kadane's Algorithm)

Tabulation





dp[i] → max(arr[i], dp[i-1] + arr[i])

Ans → Finally max of all items in dp

You can also use just variables, because we depend only on previous element.

Space Optimization