- Graph DFS & BFS -

q: Graph: (start) 1 ⟷ 2 → 3, 1 ⟷ 4, 2 ⟷ 4

1) BFS - used queue

o/p: 1, 2, 4, 3

- queue store neighbour but due to FIFO BFS

```
def bfs(node):
    visit: set()        ← empty set initialized.
    queue = dqueue(node)  # begin with start node
    while queue:        → we do it until que becomes empty
```

node select & visit operation:

```
    process_node: queue.popleft()   → start from 1st node
    if process.node in visited:
        continue.
    visited.add(process.node)   → add in visited
```

only que operation:

```
    for nb in process_node.neighbour:   → start pushing neighbour in que
        if nb in visited:
            continue.
        queue.append(nb)
```

before iteration: queue: [1]

iteration 1: node: 1 queue [ ] visited: [1]

after loop: queue [2, 4]

iteration 2: node: 2 queue [4] visited: [1, 2]

after loop: queue [4, 3]

iteration 3: node: 4 queue [3] visited: [1, 2, 4]

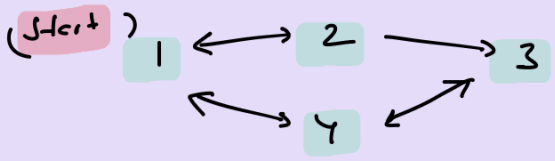
after loop: que [3] → no neighbors to vis'

iteration 4: node: 3 queue [ ] visited: [1, 2, 4, 3]

- loop → no need (no neighbour) → que empty → stop.

dfs - use stack or recursion :

stack for LIFO:




```
def dfs (node):
    visited : set()
    stack : deque ([node])
    while stack :
        node_process : stack.pop()
        if node.process in visited:
            continue
        visited.add(node-process)
        for neib in reversed(node-process.neighbour) :
            if neib in visited:
                Continue
            stack.append(neib)
```

it 0 : stack : [1] visited : ( )

it 1 : node : 1 visited : (1)

loop : stack [4, 2]

it 2 : node : 2 visited : (1, 2)

loop : stack [4, 3]

it 3 : node : 3 visited : (1, 2, 3)

loop : stack [4] → no neighbour

it 4 : node : 4 visited : (1, 2, 3, 4)

loop : → no neighbour.

Soln : (1, 2, 3, 4)

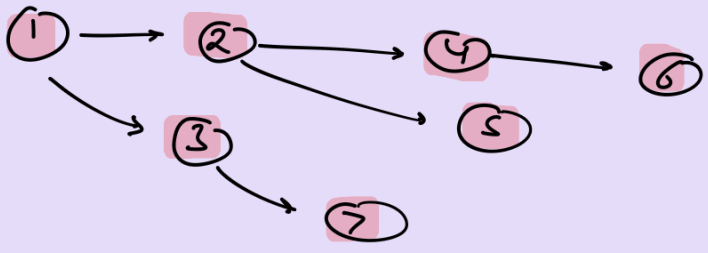
dfs using recursion -

```
def dfs(node, visited = None):
    if not visited:
        visited = set()
    if node in visited:
        return
    visited.add(node)
    for neib in node.neighbours:
        dfs(neib, visited)
```

if visited no defined

if it's last node or visited return

neighbor consider as tree & do dfs.



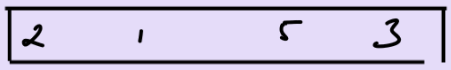
1, 2, 4, 6, 5, 3, 7 : o/p

1) Two Sum → given i/p array & target sum.
find index i, j that sum up to target.

Ex: 2, 1, 5, 3 target = 4 o/p: 1, 3

→ Soln: Use hash map to store num & index.

Hash map	
Val	index
2	0



Check (target - val) in hash map
- if present return current index and hashmap index
- if not then save val & index in map.

2) Best time to buy & sell stock -
given array of element showing price at each day
We can buy / sell once per day.

Ex 7, 1, 5, 3, 6, 4 o/p: 5 → buy: 1 sell: 6

→ Soln = 2 pointer
L: 0, R: 1 → update R: R+1 (need to move everyday forward)
Save max profit: arr[R] - arr[L]
if arr[L] > arr[R] → L: R (search min price)

3) Find Duplicate in array & return T or F
→ Soln: use hashSet → set() & find if element present

4) Product of array element except self.

eg. [1, 2, 3, 4] o/p: [24, 12, 8, 6]

arr: 1, 2, 3, 4

prefix: 1, 2, 6, 24

postfix: 24, 24, 12, 4

at 3: prefix (2) × post (4) : 8

- space complexity is there.

→ without extra space:

1, 2, 3, 4

① → 1, 1, 2, 6 (1st iteration)

1×24, 1×12, 2×4, 6×1 ← ① (2nd iteration)

24, 12, 8, 6 o/p

5) Max subarray - contiguous subarray largest sum.

eg: [-2, 1, -3, 4, -1, 2, 1, -5, 4]

o/p: 6

→ Soln: Kadane's Algorithm -

i) maintain current sum & max sum starting with 1st ele.

curr_sum : arr[0] max_sum : arr[0]

ii) forward pass & compute current sum

- if current sum becomes -ve → discard everything and begin from ele

- else keep on calculating

curr_sum: max (element → start fresh , element + curr_sum → keep on calculating.)

→ check max_sum & update.

6) Maximum Product subarray - find contiguous with largest product

eg: i/p : [2, 3, -2, 4] : o/p : 2 x 3 = 6

i/p : [-1, 2, 3, 4, -1] : o/p : 24

→ Soln : use Dynamic programming.

i) we will calculate curr_min & curr_max to 1, 1
- update res: max (list)

ii) we calculate num[i] x curr_min & num[i] x curr_max
and update curr_min & curr_max

iii) edge case of 0 → either on left side of 0 we have max or right side so we make curr_min & curr_max 1, 1 again

eg: -1, -2, -3, 0, 3, ←

max : 2
min : -2

max : 6
min : -6

1, 1 . . . . .

7) find min element in rotated sorted list.

eg: arr : [3, 4, 5, 1, 2] → 3 times rotated

soln → binary search.

[3, 4, 5, 1, 2]
↓ ↓ ↓
l mid h

either left side of mid or right side is sorted

ie arr[l] ≤ arr[mid] → left sorted

- in this case shift to right

l = mid+1

else r = mid - 1

i) [3 , 4 , 5 , 1 , 2]

l mid h

{ return when arr[l] ≤ arr[r] }

ii)

l h

(mid)

eg 2 : [4 , 5 , 6 , 7 , 0 , 1 , 2]

mid r

i) l

l mid r

ii)

8) Searching in rotated sorted array -

4 : [4 , 5 , 6 , 7 , 0 , 1 , 2] → T = 1 o/p = 5

→ soln 4 5 6 7 0 1 2

l mid R

→ In above l, mid, R situation one of the sides will be sorted

i) if a[l] ≤ a[mid] → we are in left sorted side.

if T ≤ arr[mid] or T ≥ arr[l]:

r: mid - 1

else : l: mid + 1

if right sorted side.

if a[mid] ≤ T or a[r] ≥ T:

l: mid+1

else : r: mid - 1

→ 2nd soln : find pivot → run 2 binary searches

9) 3 sum problem given an array → find all possible combination. (sum = 0)

eg: [-1, 0, 1, 2, -1, -4] o/p: [-1, 0, 1], [-1, -1, 2]

→ soln: 1) sort the array.

2) use 2 loops -

i) 1st loop is a element

ii) 2nd loop use two sum II pointer soln to get b & c

Time complexity - sort: O(n log n) 2 sum: O(n²)

= O(n²)

[-1, 0, 1, 2, -1, -4]

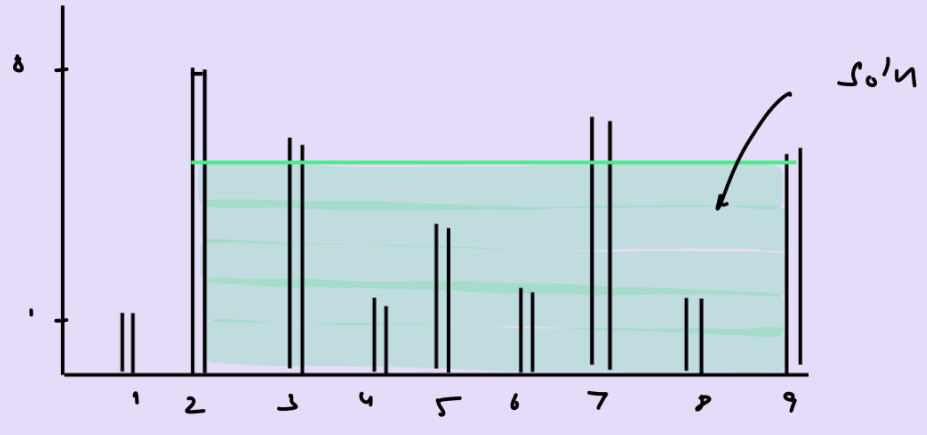
= [-4, -1, -1, 0, 1, 2]

a l r

2 sum 2 problem

10) Container with most water. Given heights.

eg:



[1, 8, 6, 2, 5, 4, 8, 3, 7]

→ soln: i) 2 pointer method at start & end.

ii) store max area

iii) if change pointer whichever is smaller.

if left small → l += 1 else r -= 1

11) # of 1 bits.

ip: 0000 ... 1011 o/p: 3

→ soln: 1) check last bit is 1 & right shift

n & 1 == 1 if last bit is 1

n = n >> 1 right shift

2) n & (n-1)

(i) 1001 (n)
1000 (n-1)
1000

(ii) 1000 (n)
0111 (n-1)
0000
done

12) Counting bits - given a value 'n' return array of length n+1 i.e 0 to n with each value having count of 1's in binary representation.

ip: n = 2 → 0 = 0, 1 = 1, 2 = 10

[0 1 1] ← soln.

→ soln: Solve with dp.

0	0000	0
1	0001	1 = 1 + dp[n-0]
2	0010	1 = 1 + dp[n-2]
3	0011	2 = 1 + dp[n-2]
4	0100	1 = 1 + dp[n-4]
5	0101	2 = 1 + dp[n-4]
6	0110	2 = 1 + dp[n-4]
7	0111	3 = 1 + dp[n-4]
8	1000	1 = 1 + dp[n-8]

all are 2^n ← multiple of 2

whenever encounter powers of 2 just update call it 'offset'

Complexity: O(n)

13) Missing element - given array containing n distinct numbers in range [0, n] find only number missing in nums.

→ soln: sum (all numbers of n) − sum(array) : missing no.

— Time complexity: O(n) space: O(1)

Calculate sum in 1 go.

iterate over 0 to n →

sum: sum + (i − arr[i]) if i < len(arr)

14) Reverse bits → given a 32 bit no. reverse the bits to obtain new no.

eg: n: 4 0...0100

o/p: 0010....0

→ soln → i) To obtain a bit value we do and (&)

eg: 0010 & 0001 : 0

0010 & 0010 : 1

left shift by 1

Note: Don't left shift the 1 instead right shift the number and (&) with 1 to get last value.

ii) To put a bit we or (|) it with 0 bit

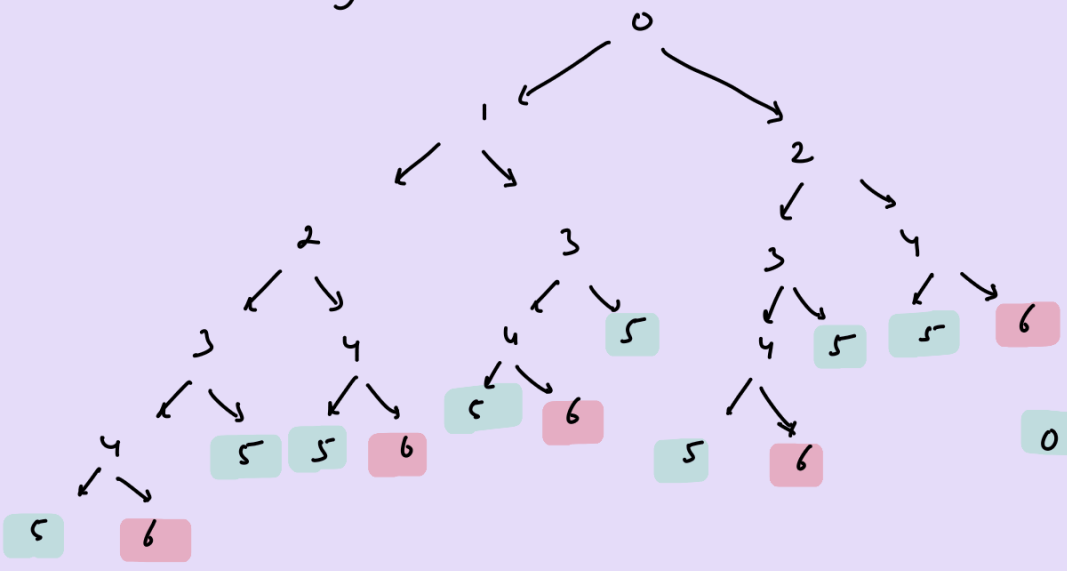
eg: rb: 0000 | 1 = 0010

1 right shifted by 1

ii) Climbing stairs - no of ways to climb n stairs and each time we take either 1 or 2 steps.

eg. n: 3 o/p: 3 (1+1+1) (1+2) (2+1)

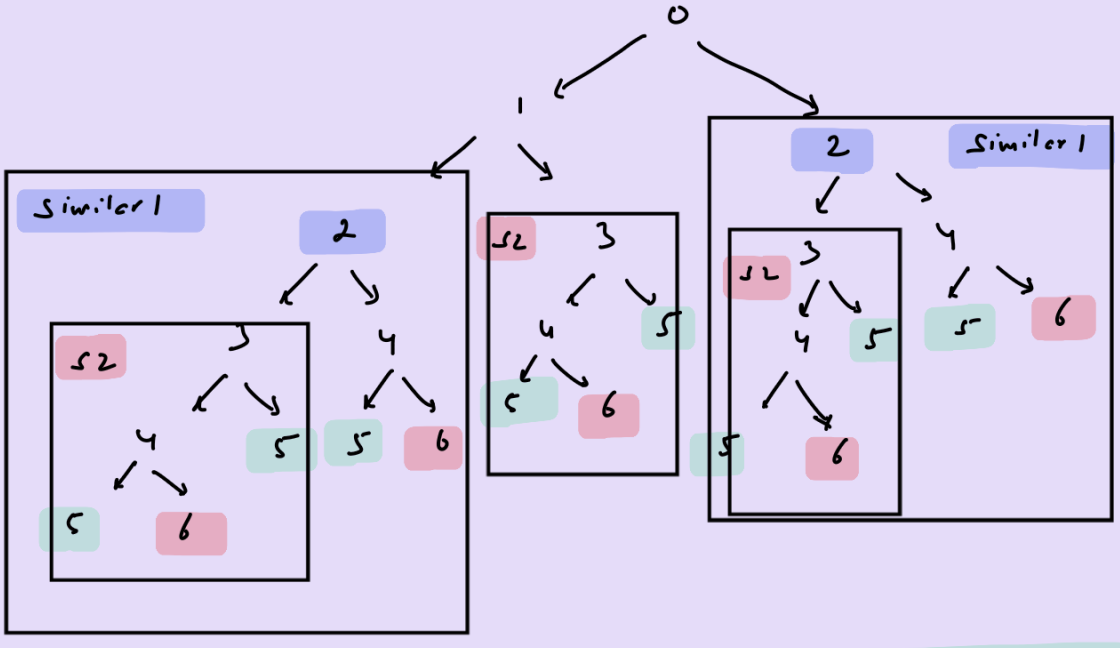
→ Soln: looking at the tree to find for n: 5



=> 8 ways.

O(2^n): O(2^5) Complexity

→ Redundancy -

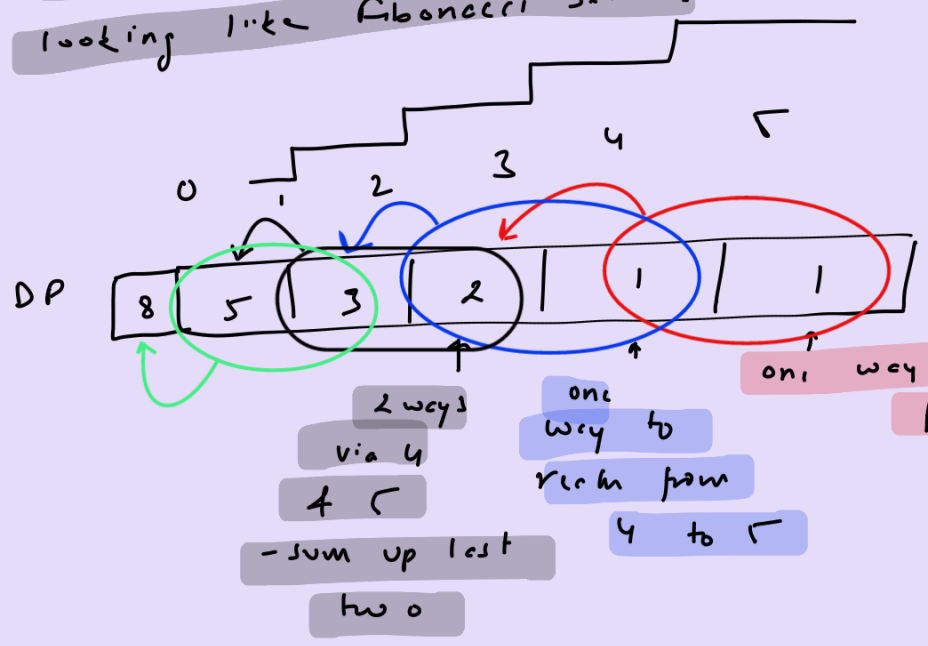


→ if we breakdown there is one go to find out 4 better to start from 5 to 0 to get ans.

4 depends on 5 → 3 depends on 4 & 5

→ 2 depends on 3 & 4 . . . O(n)

→ problem will start looking like fibonacci series.



→ instead of array we take 2 variables as 'one' & 'two'

16) Min Coin Change Problem: Given coins find min combination to get amount

Ex- coins = [1, 2, 5] amount = 11 → o/p = 3 (5+5+1)

Soln: Dynamic Programming

amount = 7 → DP of size 7 coin = [1, 3, 4, 5]

DP[0] = 0 (base case)

DP[1] = 1 = 1 + DP[0] (1 value coin there ∴ no need of other coin)

DP[2] = 1 + DP[1] = 2 (1 coin of value 1 → value of (2-1))

DP[3] = 1 (3 already in coin)

DP[4] = 1 (4 already in coin)

DP[5] = 1 (5 already in coin)

DP[6]:
- 1 ≤ coin + DP[5] : 2
- 3 ≤ coin + DP[3] : 2
- 4 ≤ coin + DP[2] : 1+2 = 3
- 5 ≤ coin + DP[1] : 1+1 = 2

} min 2 coins to get 6

DP[7]:
- 1 ≤ coin + DP[6] : 1+2 = 3
- 3 ≤ coin + DP[4] : 1+1 = 2
- 4 ≤ coin + DP[3] : 1+1 = 2
- 5 ≤ coin + DP[2] : 1+2 = 3

} min = 2

DP[7] is Ans.

17) Longest increasing subsequence -

Given integer array return length longest increasing subsequence

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

o/p: 4 → [2, 5, 7, 101] can also be

[2, 3, 7, 101]

→ Soln: using Dynamic Programming.

eg: [1, 2, 4, 3] ← elements

0 1 2 3 ← index

start from end

LIS[3] = 1 at position 3 max 1 subsequence [3]

LIS[2]: max(1, 1 + LIS[3]) = 1

↓ if can't go forward

↓ if arr[2] < arr[3] then only possible

LIS[1]: max(1, 1 + LIS[2], 1 + LIS[3])

: max(1, 2, 2) : 2

LIS[0]: max(1, 1 + LIS[2], 1 + LIS[3], 1 + LIS[1])

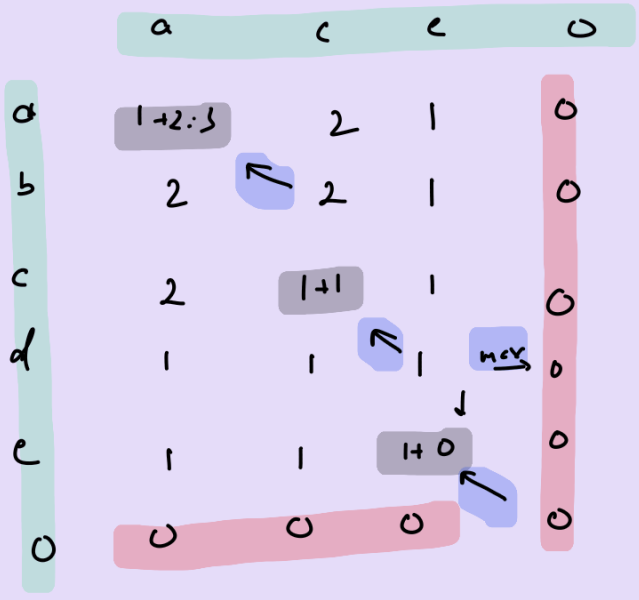
= max(1, 2, 2, 3) : 3

- it is possible that LIS[0] = 1 due to some being very high value so the answer is max(LIS[0], LIS[1], LIS[2], LIS[3])

18) Longest common subsequence: Given text 1 & text 2 find length of longest subsequence -

eg. text1: 'abcde' text 2: 'ace' - o/p: 3

→ Soln: Using 2D - DP matrix.



by default 0.

use bottom up.

if i & j match

1+ val of bottom diagonal

: 1+ dp[i+1][j+1]

if don't match:

dp[i][j] : max(dp[i][j+1], dp[i+1][j]

O(nxm)

15) Word break Problem: Given string s and dictionary of words, return true if s can be segmented into separate words all present in dictionary.

Ex: s: 'leet code' D: ['leet', 'code'] → True

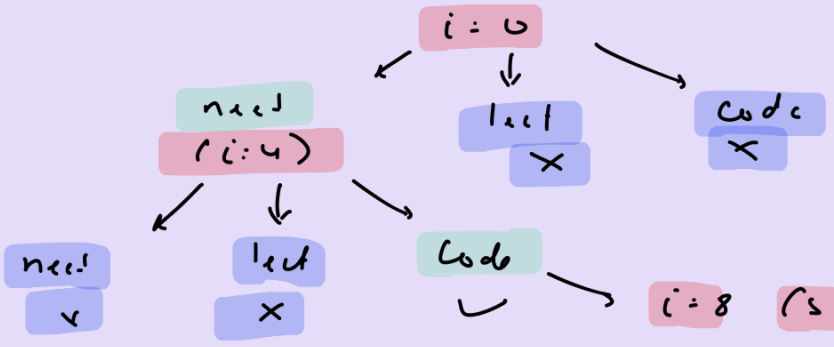
s: 'leet code' D: ['leet'] → False

s: 'leet leet' D: ['leet', 'code'] → True

→ Soln: Using DP →

To build DP: make decision tree, remove redundancy using Cache (Memo), Compress using DP.

1) Decision Tree: s: 'need code' D: ['need', 'leet', 'code']



DP[8]: False

19) Solving with DP

→ DP should be one length extra i.e len(DP): 9

→ base case: DP[8] = True last always True

'e' cant be compared with any word.

DP[7]: False

DP[6] = DP[5]: False

DP[4] = match found → check if after this match word we have True.

ie DP[4+4] = DP[8]: True

↓ matched word length

DP[3]: DP[2]: DP[1]: False

DP[0]: match found → check if

DP[0 + wordlength]: True

→ Check if position has enough words to match

Pos + word length < length of sentence

eg: 7 + 4 = 11 fails as sentence length: 8

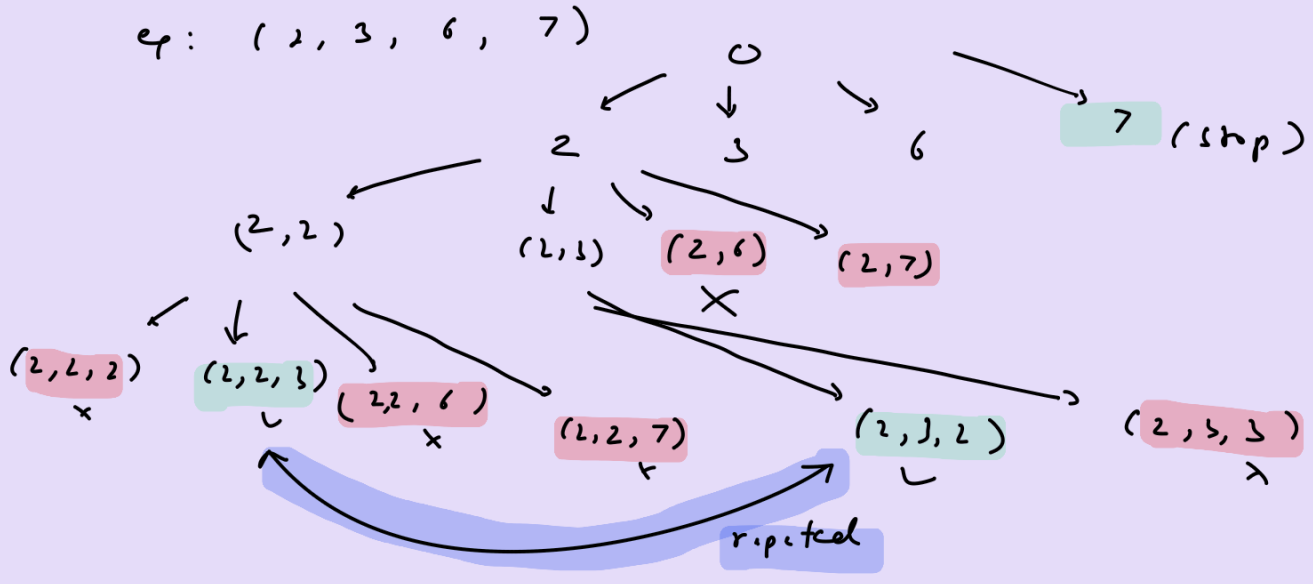
4 + 4: 8 pass as length = 8

20) Combination Sum: Given array of distinct candidate, & target. list all unique combination of candidate that sum up to target.

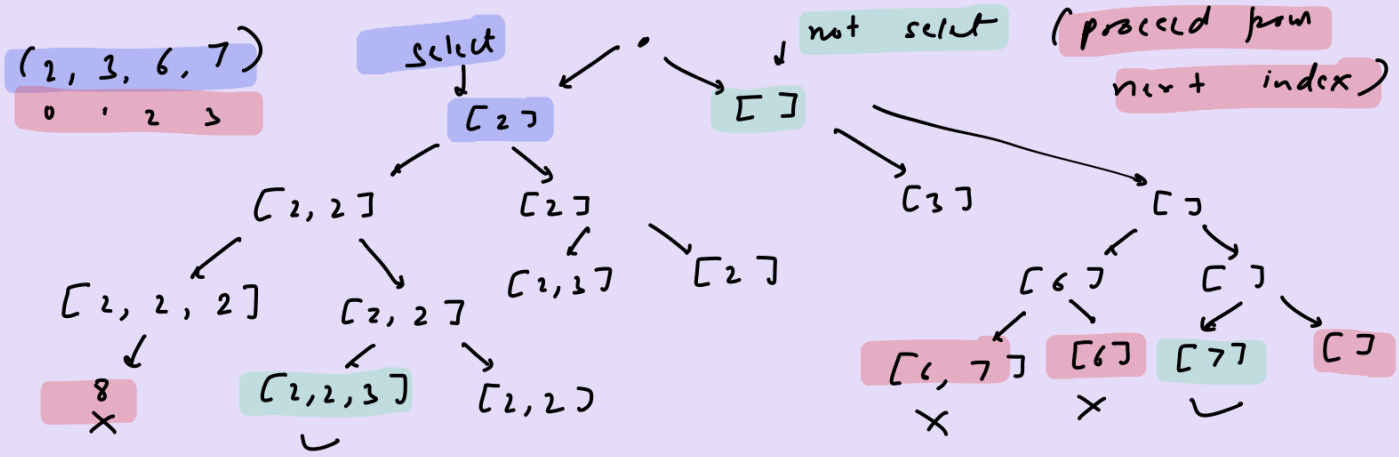
eg: can = [2, 3, 6, 7] target: 7 o/p: [[2, 2, 3], [7]]

i) using simple decision Tree will have repeated.

eg: (2, 3, 6, 7)



1) Decision Tree without repetition.



→ Using DFS & recursion we solve.

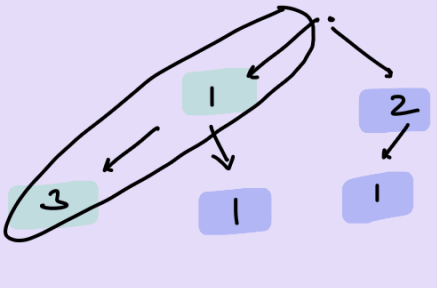
```
dfs (i, cur, total)
→ if total == Target:
      res.append (cur.copy())
  if i ≥ len (candidates) or total > target:
      return
  cur.append (candidate[i])
  dfs (i, cur, total + candidate[i])      ← select candidate & process.
  cur.pop()
  dfs (i+1, cur, total)                   ← don't select candidate & proceed from next position
```

dfs (0, [], 0) ← start from 0th position with total as 0 since no element

2) House Robber Problem: Array with value of money.
The robber can't rob consecutive houses.

eg. num: [1, 2, 3, 1] o/p = 4 = 1+3

- soln: Binary Tree:



(rob either of Consecutive)

DP: [1 ; 2 ; 3 ; 1]

rob 1 = 1

rob 2 = 2 else 1

rob = 3 1+3 = 4 else = 2

rob = 1 2+1 = 3 else 4

→ max: 1 2 4 4 → Soln

- Use 2 variables rob1 & rob2 and make 0

list look like [rob1, rob2, n, n+1 ...]

: if you went to rob nth house

: (rob1 + n)

if you don't went to rob nth house

: rob 2

→ max obtained : max (rob1+n), rob 2)

→ update rob1 : rob2

rob2 : max ((rob1 + n), rob 2)

22) House Robber 2: Same question as above but now houses arranged in circular fashion.

eg: [2, 3, 2] 2 → 3 → 2 o/p: 3

Soln: list of last house connected

- run house robber 1 code on list excluding last
- run house robber 1 on list excluding first
- take max.

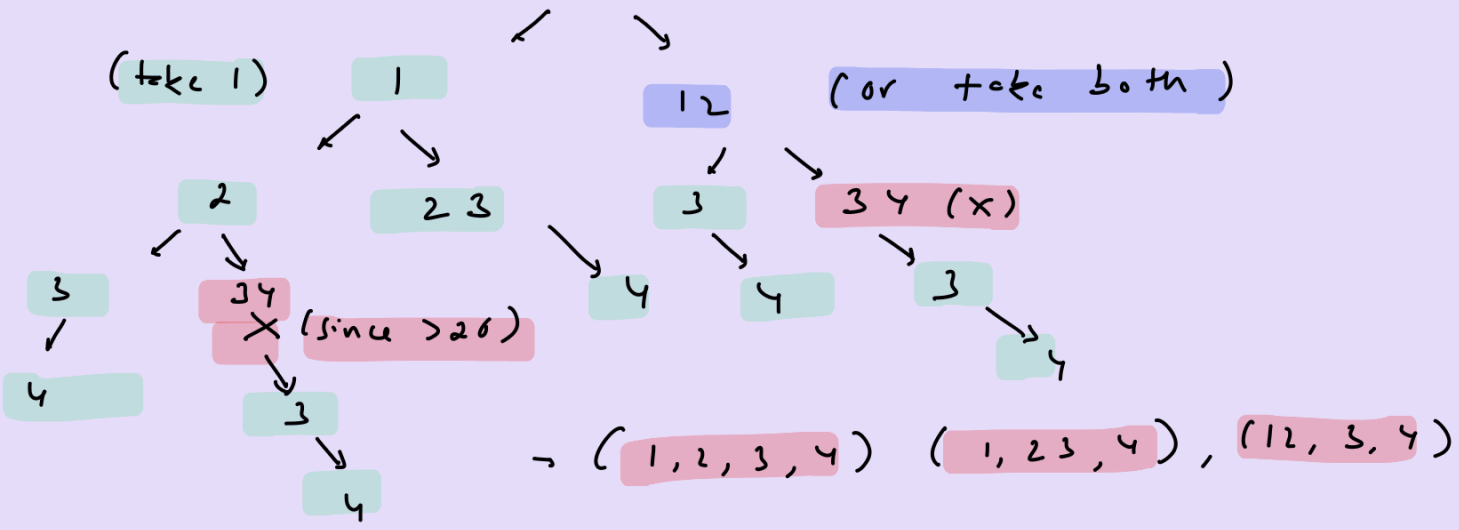
23) Decode Ways: Message is encoded as string of digits. find all possible ways to decode.

here A : 1 . . . . Z : 26

eg: '12' → '1, 2' : AB & '12' : L

o/p = 2

soln - binary Tree for 1 2 3 4



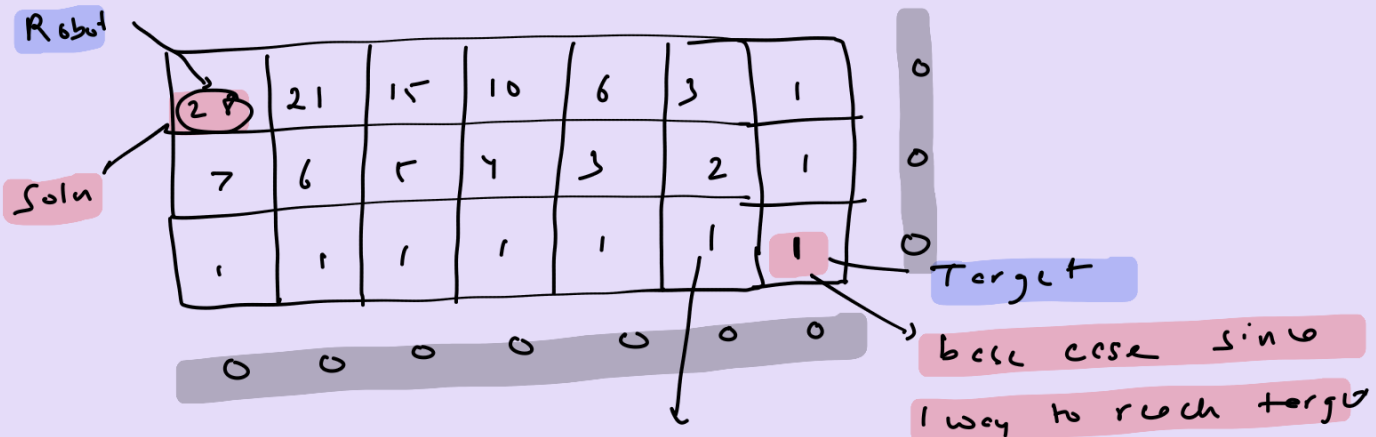

```
def numdecode(s):

    def dp(i)

base case {
        if i ≥ len(s): return 1        (reached end)
        elif s[i] == '0': return 0     (invalid cond)

    # check if we can take two values.

    if i < len(s)-1 and int(s[i:i+2]) <= 26:
        return dp(i+1) + dp(i+2)

    else:
        return dp(i+1)

    return dp(0)
```

24) Unique paths: robot located on top left of m×n grid & target at bottom left.

find # of unique paths.

Q: m: 3, n: 7 o/p: 28

→ Soln: DP



DP(i,j): DP(i+1, j) + DP(i, j+1)

25) Jump Game: Given array of non negative integers nums, initially at first index position.

return True if we can reach the last index.

value at the index is max jump we can do.

Q: nums: [2, 3, 1, 1, 4] o/p: True [2, 3, 4]

nums: [2, 3, 1, 0, 4] o/p: false

Soln → Goal post method.

Goal post

e.g. [2, 3, 1, 1, 4]

(i) you can move to goal post

(ii) can move → shift post

(n) ..... continue moving post until post at loc 0

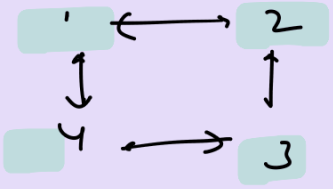
26) Clone the graph: Return a deep copy.

node contains a val & neighbours (list of nodes)

→ soln: BFS using hashmap.

hashmap

old	new
1	1
2	2
3	3
4	4



1) if node not in hashmap, create a copy

copy: Node (node.val)

2) start filling neighbours & follow same

```
def dfs(node):
    if node not in hashmap:
        copy = Node(node.val)        ← if not made clone then make it
    else: return hashmap[node]       ← clone already made
    for nei in neighbours:
        copy.neighbours.append(dfs(nei))
                                     ↓
                                adding neighbour & also cloning it before adding.
    return dfs(node)
```

27) Course Schedule - Total numCourses labled 0 to numCo-1

Some courses may have a prerequisite eg.

[0, 1] is 0th course require course 1 (0 → 1)

- find if it's possible to finish all courses.

eg: numCourses = 2 pre = [[1, 0]] o/p = True

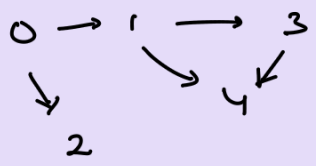
1 → 0 1st do 0 then 1

: numCourses = 2 pre = [[1, 0], [0, 1]] o/p: false.

forms cycle.

→ solu: use a visited set to detect cycle & do dfs to find and fill another map.

eg: n = 5 pre: [[0, 1], [0, 2], [1, 3], [1, 4], [3, 4]]



(i) pre map

course	pre-requisite
0	[1, 2]
1	[3, 4]
2	[ ]
3	[4] → one check make [ ]
4	[ ]

complexity : O(n + p)

(ii) start visit set :

- set it empty set
- byin with 1st course
  - → if that course already in visited set then cycle detected & return false
  - → if pre map [course] = [ ] no prerequisite means True for that course
    - → add this course to visited set (so that we can detect any cycle in prerequisites)
  - → if pre map not empty. then dfs the pre rq course.
  - → if fully traversed the pre requisite then simply make pre map (course) = [ ] & remove curse from visited set.

base case for dfs

Note: In case of cycle we have visited set failing

eg: [[0, 1] [1, 0]]

visited set

(i) 1

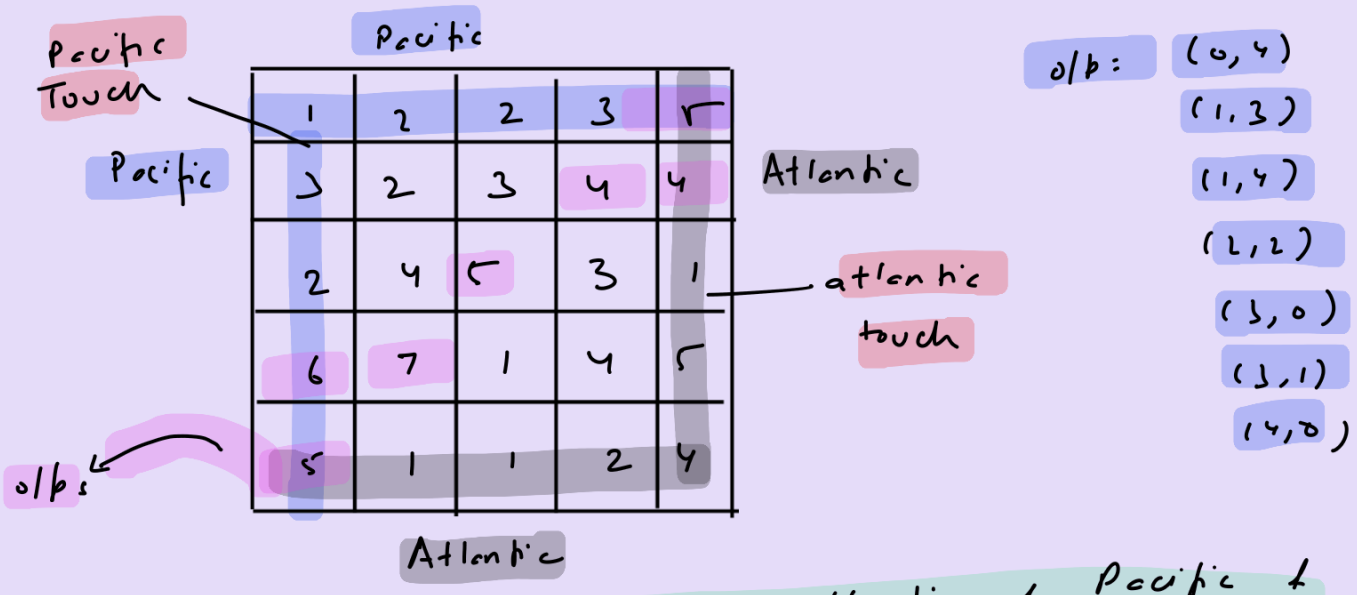
(ii) 0

(iii) again try to include 1 (cycle detected)

28) Pacific Atlantic water:

Given mxn matrix with nights find all indices that can reach atlantic as well as pacific ocean.

- water can only flow in 4 dirn



Soln → we need traversal for atlantic & Pacific & then union of both.

- we start from outer ends & look for neighbours which are same or more height
- we use recursive fn & traverse all 4 direction.

for traversing:

i) all point reaching to pacific:
- we go all column of row 0
- we go all rows for coln 0

ii) all points reaching Atlantic:
- we go all Columns of last row (Row - 1)
- we go all row of last colum (Col - 1)

- stopping condition -

point inside visited_set, row, col out of bounds or height of adjacent is less.

29) Number of Island : Given binary m x n grid.

- Connected is one island.

q:

1	0	0	1
1	1	0	1
0	0	1	0
1	0	1	0

o/p : 4

→ soln: Can be done with dfs or bfs.

- we need a visited set
- run for all row & col the bfs (r, c) fn if (r, c) not in visited & grid[r][c] == 1
- visited set can be filled up with bfs or dfs.

30) longest Consecutive Sequence -

Given unsorted array find the longest consecutive sequence length.

q: [100, 4, 200, 1, 3, 2] o/p: 4 [1, 2, 3, 4]

→ soln: (i) if num is start of sequence then (num - 1) not there.

(ii) if we found start we must iterate unitl the last consecutive element is found

(iii) use a set to store array.

- we get 3 sequences (1, 2, 3, 4), (100) (200)

longest

31) Alien Dictionary - Alien language use english Alphabet but not same order.

Given sorted list of words return a string of unique letters in new alien language.

- if no soln exists return ""
- if multiple soln return any of them

Note: if word1 < word2 & have same alphabet then false. eg. ape, apes

eg: ['wrt', 'wrf', 'er', 'ett', 'rftt']

o/p = wertf

→ soln:

wrt
wrf
er
ett
rftt

look for 1st difference (-t4f) since ordered t < f ∴ t → f

w → e
e → r
r → t

Graph becomes -

w → e → r → t → f

- Cases where issue comes -

{we, ew} - w ⇄ e return "" (loop)

- other case:

A
BA
BC
C

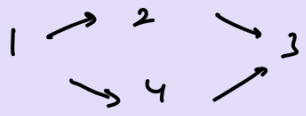
A → B → C

- DFS but option in all case but here it will give

ACB

⇒ we use post order DFS.

Post order DFS



Post order : 3 4 2 1

- all children then finally the parent node

- in 

Code DFS : C B A

for o/p we reverse : A B C

→ Soln :

1) form Graph using the list
   a) initialize adjacency dict
   b) for num matching alphabet fill us adjacency
2) perform post order dfs
3) reverse the o/p.

post order dfs :

visit = { } dict : False if visited

res = [ ] True : if in current path & again visited (loop)

```
def dfs(c):
    if c in visit:
        return visit[c]   ← can be True (loop) or False (visited)

    visit[c] : True  → if not present we let mark it True

    for nei in adj[c]:
        if dfs(nei):
            return True   } if any loop return True

    visit[c] : False  → it means we have visited & is not in current path

    res.append(c)

for c in adj:
    if dfs(c) : ""   →   rev(res)
```

- case of loop : w ⇄ e

w not in visit : visit [w] : True

neib (w) : e → visit [e] : True

neib (e) : w → w in visit & True

means loop detected.

- case of no loop but multipath :

A not in visit : visit [A] : True

→ neib (A) : B, &

→ B not in visit : visit [B] : True

neib [B] : C → C not in visit : visit [C] : True

→ neib (C) : empty. (I)

visit [C] : false res : [C]

→ visit [B] : false res : [C, B] (II)

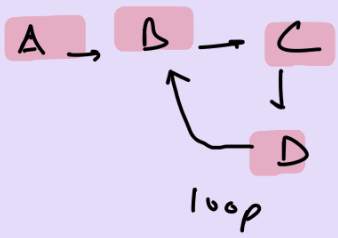
→ neib (A) : ~~&~~, C

C in visit → return false

C was made false so not fully detect loop.

visit [A] : false res : [C, B, A] (III)

eg :



if we do for A

visit [A] : True

visit [B] : True

visit [C] : True

visit [D] : True

— again B appear & True so loop & fail.

4: A → B → C → D (Simple)

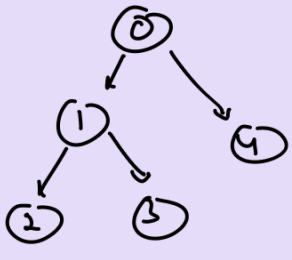
visit [A]: True — [ ]
visit [B]: True — [ ]
visit [C]: True — [ ]
visit [D]: True — [ ]
— nowhere to go
visit [D]: False — [D]
visit [C]: False — [D, C]
visit [B]: False — [D, C, B]
visit [A]: False — [D, C, B, A]

Note: This loop search only happens in unidirected graphs.

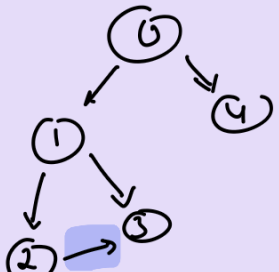
32) find wether graph is valid Tree.

- if empty it Tree
- no loops
- no disconnection.

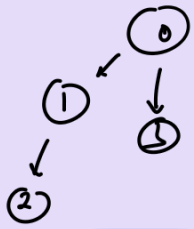
4:



Tree



not Tree



not Tree

→ soln: use parent info to avoid checking loop in bidirectional graph

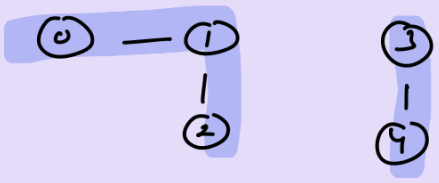
dfs(node, parent)

parent: -1
when node: 0

- if n (Total nodes) = len(visited) → no disconnection.

32) Number of Connected Components in Undirected graph.

q:



o/p = 2

edges: [[0,1], [1,2], [3,4]]

→ soln:

1) simple soln:

iterate all nodes:

perform dfs(node) if node not in visited:

increment counter.

11) Union-Find Algorithm:

- performs find() - find root parent
- performs union() - merge smaller set to larger.

eg:

- performs path compression

start: Consider all node to be un connected.

parent: [0 1 2 3 4] → all nodes are parent of itself

0 1 2 3 4

rank: [1 1 1 1 1] - since all disconnected rank = 1 for all

0 1 2 3 4

edges: [[0,1], [1,2], [3,4]]

iteration 1: [0, 1]

parent: [0 ~~1~~ 0 2 3 4] → update root node

rank: [2 1 1 1 1]

↑

increment rank.

iteration 2: [1, 2]

parent(2): 1 but parent(1): 0 ∴ parent(2): 0

parent: [0 0 ~~2~~ 0 3 4]

rank: [3 1 1 1 1]

iteration 3: [3, 4]

parent(4): 3 → 3 is parent of itself

parent: [0 0 0 3 ~~4~~ 3]

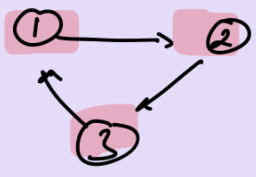
rank: [3 1 1 2 1]

→ we did merge operation 3 times.

connected components: n - # of times merge

: 5 - 3 = 2

→ Detecting cycle: if union fails then cycle.



[[1, 2], [2, 3] [3, 1]]

p: [1 2 3] rank: [1, 1, 1]

0 1 2

[1, 2]: p: [1 ~~2~~ 3] rank: [2, 1, 1]

[2, 3]: p: [1 1 ~~3~~] rank: [3, 1, 1]

[3, 1]. p(1) = 3 => p(3): 1 fails.

→ Code logic:

```
def find(n1):
    res: n1
    while res != par[res]:
        par[res]: par[par[res]]
        res: par[res]
    return res.
```

- in the beginning when parent[i]: i while loop not executed & find(n): n

```
def union(n1, n2)
p1, p2 = find(n1), find(n2)
if p1 == p2:
    return 0
if rank[p2] > rank[p1]:
    parent[p1] = p2
    rank[p2] += rank[p1]
else:
    par[p2] = p1
    rank[p1] += rank[p2]
return
```

1st time this will not be true since p[i] = i
→ if this condition happens, means n1 & n2 same set

1st time else will happen since p1 = n1 & p2 = n2
r[p1] = 1 r[p2] = 1

34) Insert Interval: Given intervals & new Interval

e.g.: interval: [[1,3],[6,9]] newInterval = [2,5]

o/p: [[1,5],[6,9]]

e.g.: interval: [[1,2],[3,5],[6,7],[8,10],[12,16]]

newInterval: [4,8]

o/p: [[1,2],[3,10],[12,16]]

→ Soln: 1) if new interval before interval

newinterval[1] < interval[i][0]

- return new interval + intval[i:]

2) if new interval after interval

res.append(interval[i])

3) else:

new interval: [min(newintval[0], interval[i][0]), max(newinterval[1], interval[i][1])

soln: [[1,2], [3,5], [6,7], [8,10], [12,16]]

new interval: [4, 8] res: [ ]

1: [1,2] → cond 2 satisfy

res: [[1,2]]

2: [3,5] → cond 3 satisfy

new interval: [3, 8] , res: [[1,2]]

3: [6,7] → cond 3 satisfy

new interval: [3, 8] , res: [[1,2]]

4: [8,10] → cond 3 satisfy

new interval: [3, 10] , res: [[1,2]]

5: [12,16] → cond 1 satisfy

res: [[1,2], [3,10]]

return res + interval[i:] = [[1,2], [3,10], [12,16]]

35) Merge Interval: array of intervals (might not be sorted) merge overlapping interval.

Q: [[1,3], [2,6], [8,10], [15,18]]

o/p: [[1,6] [8,10], [15,18]]

4: [[1,4], [4,5]]: o/p = [1,5] here consider overlapping.

→ soln: op.append(interval[0])

start from interval[1]

current: op[-1]

- look for overlap:

op[-1][1]: max(current[1], interval[i][1])

else

op.append(interval[i])

36) Non overlapping intervals:

Given an array return min no of intervals we need to remove to make array non-overlap.

Ex: [[1,2], [2,3], [3,4], [1,3]] o/p: 1

remove [1,3]

Soln. cases:

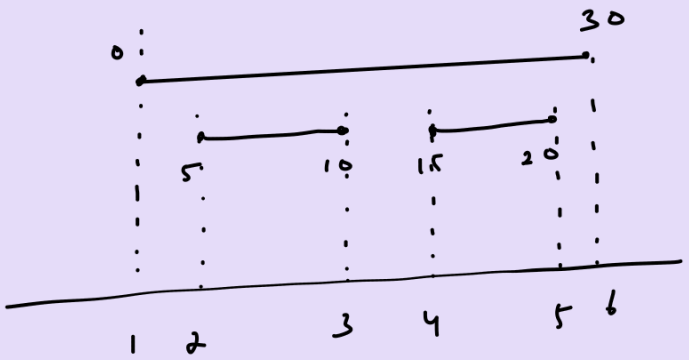
37) Meeting Rooms -

Given an array of interval return True if a person will be able to attend all meeting.

→ soln: if overlap return False.

38) Meeting Room 2: find max no of meeting rooms required

Ex: [(0,30), (5,10), (15,20)] o/p: 2



→ soln: Count based on start & end time.

at instance 1 1st meeting begin → C = 1

at instance 2 2nd meeting begin → C = 2

at instance 3 2nd meeting end

C = 1

at 4 : new meeting c = 2

at 5 : new meeting end c : 1

at 6 : meeting end c : 0

res : max (count)

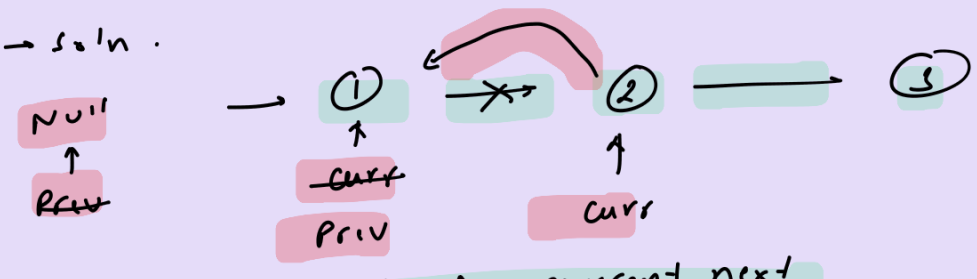
- whenever meeting begin c++ when end c--
- use 2 pointers & 2 arrays (1 for start & 1 for end)

37) Reverse linked list :

Q : 1 → 2 → 3 → NULL

3 → 2 → 1 → NULL

→ soln :



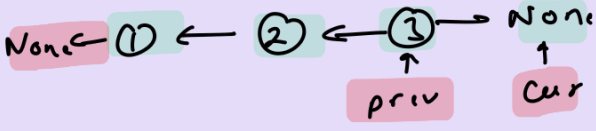
nxt_nfo = current.next

current.next = prev

prev = current

curr = nxt_nfo.

→ return prev of 3rd



40) find loop in Linked list O(n)

Soln → Floyd Hare & Tortoise soln

- use 2 pointers one slow (+1) one fast (+2)

why O(n) →

Total length = n

slow moves => dist blw fast & slow : n+1

fast moves → " " " " : n+1-2

= n-1

∴ after n times fast = = slow.

4) Merge two sorted list:

→ ip: 1 → 2 → 4 , 1 → 3 → 4

o/p: 1 → 1 → 2 → 3 → 4

Soln: initialize a list with dummy node so that we don't insert to dummy node.

1 → 2 → 4

1 → 3 → 4

list: dummy tail: dummy

- if val(l1) > val(l2)

  tail.next = l2

  l2 = l2.next

- if val(l2) > val(l1)

  tail.next = l1

  l1 = l1.next

tail = tail.next

This stops when one of the list finishes.

for the remaining part of list

if l1: (l1 remain)

  tail.next = l1

elif l2: (l2 remain)

  tail.next = l2

return dummy.next.

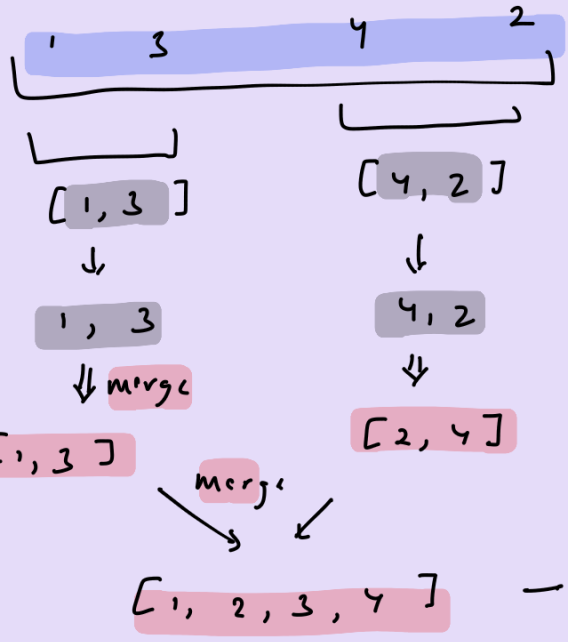
42 Merge k sorted lists:

Soln → use technique of merge sort

1) recursively divide the list in 2 parts

2) merge divided list.

4:



divided Greedily

```
merge (l1, l2):          # use pointer
    p1, p2 = 0, 0
    sorted_list = []
    while p1 < len(l1) and p2 < len(l2):
        if l1[p1] < l2[p2]:
            sorted_list.append(l1[p1])
            p1 += 1
        else:
            sorted_list.append(l2[p2])
            p2 += 1
    # append remaining
    sorted_list.extend(l1[p1:])
    sorted_list.extend(l2[p2:])
    return sorted_list
```

fn is similar to previous question.

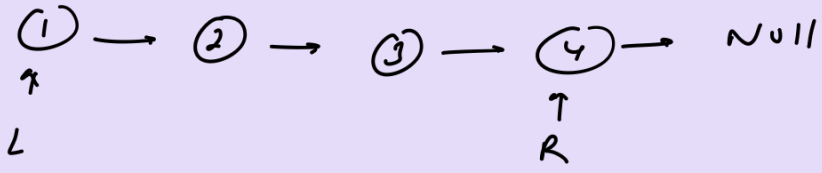
43) Remove nth node from end of list:

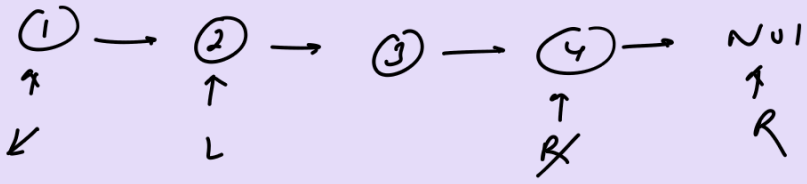
q: n: 2

① → ② → ③ → ④ → Null

① → ② → ④ → Null

soln: take 2 pointers with separation of (n+1) & traverse until final pointer reaches null.





L.next: L.next.next

44) Reorder list:

1 → 2 → 3 → 4

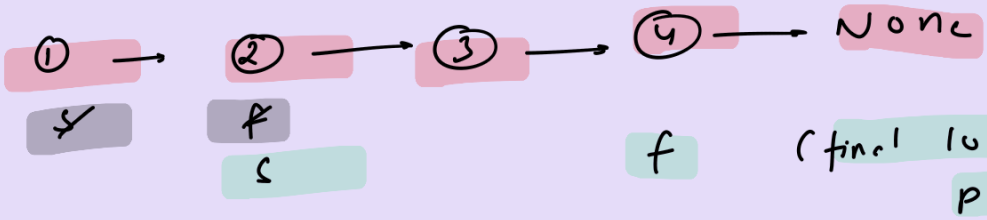
o/p: 1 → 4 → 2 → 3

→ soln: Divide list into 2 using slow & fast pointer

slow += 1 fast += 2

Note:
slow: head
fast: head.next

– Reverse the 2nd list



→ get 1st & 2nd list

s.next: ③ → ④ → None.

first: ll ① → ② → ③ → ④

if we make slow.next: None

first: ll ① → ② → None

=> lst.next: 2nd

2nd.next: tmp1 (tmp1: lst.next)

lst, 2nd: tmp1, tmp2

4 r) set matrix:

q: [[1,1,1],[1,0,1],[1,1,1]] → [[1,0,1],[0,0,0],[1,0,1]]

row & col to 0

→ soln: Time complexity O(mxn)

Soln: Space complexity O(mxn)

[[1,1,1],[1,0,1],[1,1,1]] copy → [[1,1,1],[1,0,1],[1,1,1]]

Traverse here change here

Soln: Space complexity O(m+n)

[x] col marker

[x] [[1,1,1],[1,0,1],[1,1,1]]

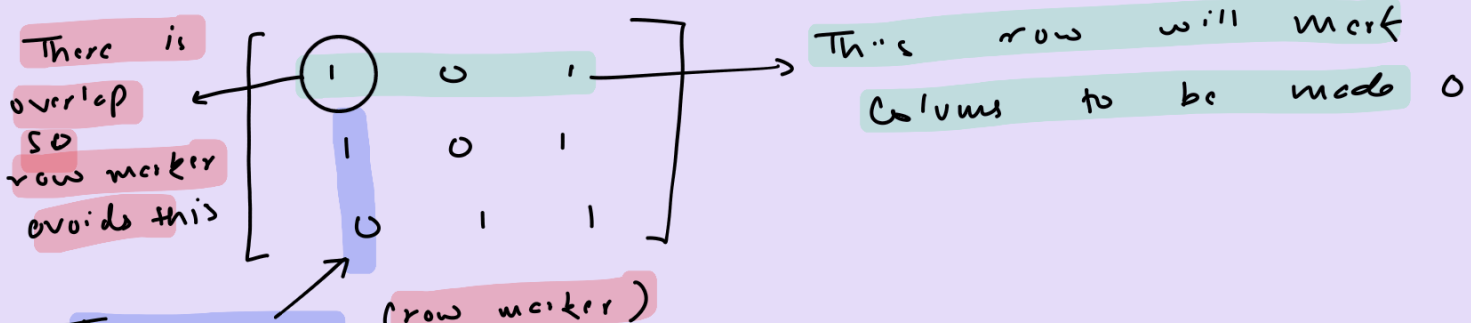
row marker

→ use row marker & column marker to mark.

Soln: Space Complexity O(1)

- we use 1st row & 1st colum of this matrix as a marker
- we process arr[1:row, 1:colum] using this marker

- we maintain a variable to decide whether this row needs to be marked.



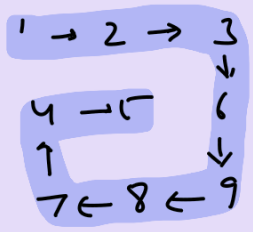
- row marker avoids 1st element so we keep 0-th to decide whether to mark 1st row as 0.

Algo:

```
row_zero = False

for r in rows:
    for c in cols:
        if mat[r][c] == 0:
            mat[0][c] = 0        (column marker)

        if r > 0:
            matrix[r][c] = 0     (row marker)
                                 except 1st row
        else:
            row_zero = True

Start marking rows & colums of array except
1st row & column.

for r in (1, rows):
    for c in (1, colum):
        if mat[0][c] == 0 or mat[r][0] == 0:
            mat[r][c] = 0

mark 1st colum →
if mat[0][0] == 0
    for r in rows:
        mat[r][0] = 0

mark 1st row →
if row_zero:
    for c in col:
        mat[0][c] = 0
```

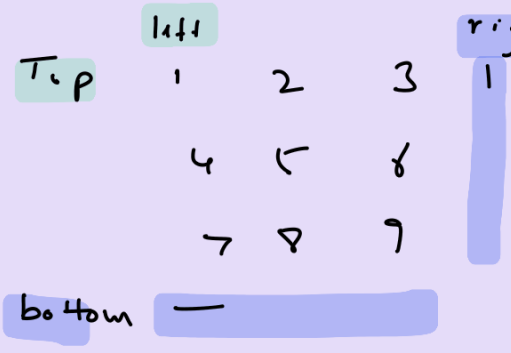
41) Spiral matrix:

Given matrix traverse in spiral way:

Q: 

o/p: [1, 2, 3, 6, 9, 8, 7, 4, 5]

Soln: we need 4 pointers. Top, bottom & left right.



- we keep bottom & right outside the bounds for code.

top = left : 0
right: len(mat)
left: len(mat[0])

Traversal:

1) left to right:

```
i in range (left, right):        (column)
    res.append (mat[top][i])  → top row
top += 1
```

2) top to bottom:

```
i in range (top, bottom)  → (rows)
    res.append (mat[right-1][i])  → right-1 col
right -= 1
```

3) right to left:

```
i in (right-1, left-1, -1)  → (col)
    res.append (mat[bottom-1][i])  → bottom-1 row
bottom -= 1
```

4) bottom to top:

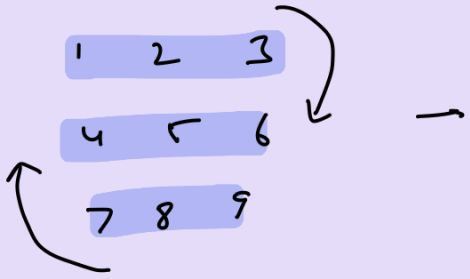
i in (bottom-1, top-1, -1) → (rows)

res.append(mat[i][left])

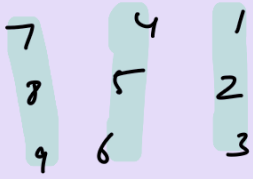
left += 1

47) Rotate Image : rotate 90° clockwise.

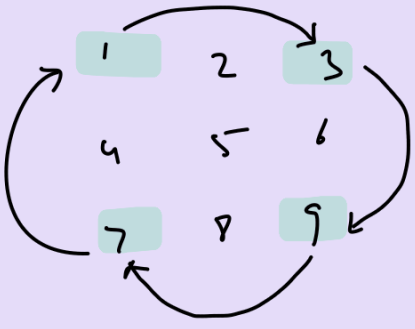
Q:



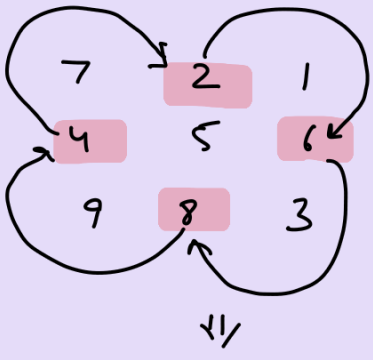
→



Soln:

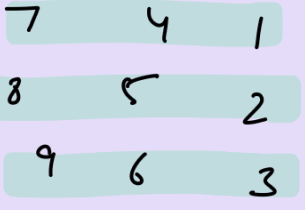


⇒

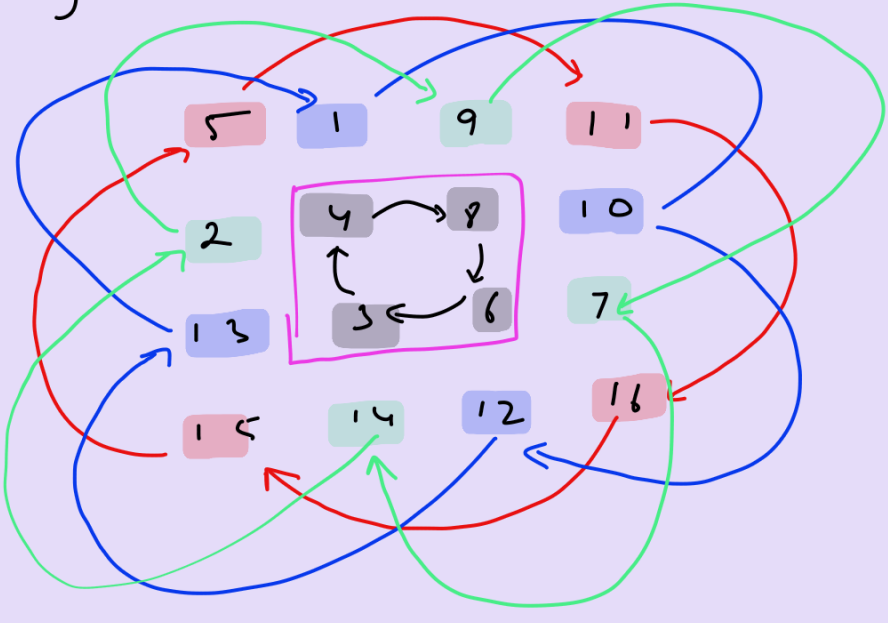


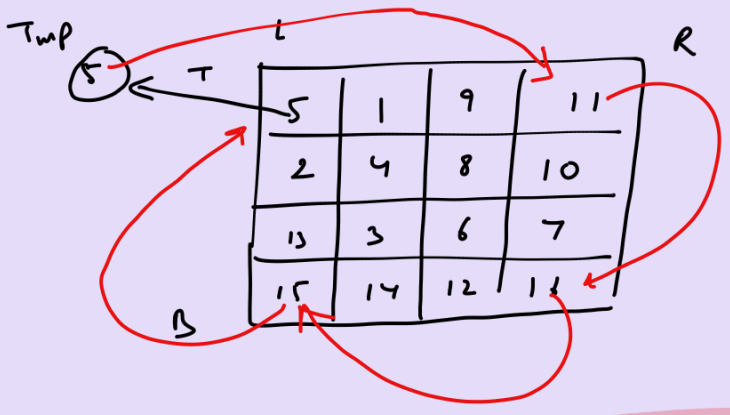
1st i +

5 no new. ⇐



- big example





- we will do in revse ordr, i.e anticlockwise
- we will need to save only 5 in tmp variable

∴

Logic → The matrix is n x n

L : T & R : B

→ save TL element in tmp

tmp : mat [T] [L]

→ move Bottomleft to Top left

mat [T][L] : mat [B] [L]

→ move Bottom Right to Bottom left

mat [B][L] : mat [B] [R]

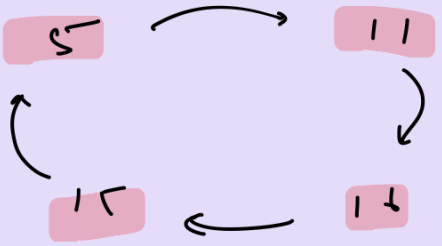
→ move Top Right to Bottom Right

mat [B] [R] : mat [T] [R]

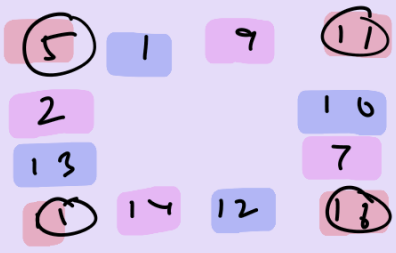
→ move saved tmp into Top Right

mat [T] [R] : tmp

This moves



- This subproblem has



To solve for 1 → 10 → 12 → 13

& 9 → 7 → 14 → 2

we use i variable to update logic

i in range (R-L)

→ save TL element in tmp

tmp = mat[T][L+i]

→ move Bottomleft to Top left

mat[T][L+i] = mat[B-i][L]

→ move Bottom Right to Bottom left

mat[B-i][L] = mat[B][R-i]

→ move Top Right to Bottom Right

mat[B][R-i] = mat[T+i][R]

→ move saved tmp into Top Right

mat[T+i][R] = tmp

Update r & left → r-- l++

4) word search:

4: A B C E
S F C S
A D E E

word: A B C C E D

o/p: True

→ soln: back tracking using dfs.

path = set()

dfs(r, c, i) i = index of word (begin with 0)

return True if i = len(word)

return False if r, c out of bounds or
word[i] != board[r][c] or
(r, c) in path

if no condition above

path.add((r,c)) → take a (r,c) & search

res = (dfs(r+1, c, i+1) or
dfs(r-1, c, i+1) or
dfs(r, c+1, i+1) or
dfs(r, c-1, i-1)

path.remove((r,c)) → remove that (r,c) so that it may be used in other (r',c')

- This way storing of (r,c) we can traverse and search around neighbours.

4) longest substring without repeating elements:

q. 'abcabcbb' o/p = 3 'abc'

soln - use sliding window & hashset.

4: abcad

l, r

hash = { } l = 0 r = 0

i+1: l:0 r:0 hash(a) → r = 1

i+2: l:0 r:1 hash(a,b) → r = 2

i+3: l:0 r:2 hash(a,b,c) → r = 3

i+4: l:0 r:3 hash(b,c,a) → l = 1 r = 4

i+5: l:1 r:4 hash(b,c,a,d) → (stop)

max = 4-1+1 = 4 bcad

4: abbc

l:0, r:0 hash(a)

l : 0, r : 1 hash(c, b) → r : 2

@ r : 2 b already in hash

— we start removing from left until b is gone

done using while loop { l : 1 hash (b) ; l : 2 hash ( ) } → now we can start from b

50) longest repeating character Replacement.

Given a string & value K

we can change characters K times &

find length of longest substring with same letters.

ex: 'AABABBA' k : 1

o/p : 4 → 'AA B A BBA' (B → A)

→ Soln. Use sliding window & hashmap.

condition on sliding window:

if len(window) − max_repeated_char ≤ K

ex: A A B A B B A

l, r

hashmap : { }

1: hash map : { A : 1 }

window length : r − l + 1 : 1

1 − max repeating char : 1 − hashmap[A]

: 1 − 1 ≤ k (satisfy)

1) l = 0 r : 1

hash : { A : 2 }

2 - 2 ≤ K satisfy

3) l : 0 r : 2

hash : { A : 2 , B : 1 }

3 - 2 : 1 ≤ K Satisfy

4) l : 0 r : 3

hash : { A : 3, B : 1 }

4 - 3 ≤ K Satisfy.

5) l : 0 r : 4

hash : { A : 3, B : 2 }

5 - 3 = 2 ≤ 2 satisfy

6) l : 0 r : 5

hash : { A : 3, B : 3 }

6 - 3 : 3 ≤ 2 fails l++

7) l : 1 r : 5

hash : { A : 2, B : 3 }

5 - 3 : 2 ≤ 2 satisfy

8) l : 1 r : 6

hash : { A : 3, B : 3 }

6 - 3 : 3 ≤ 2 fails too

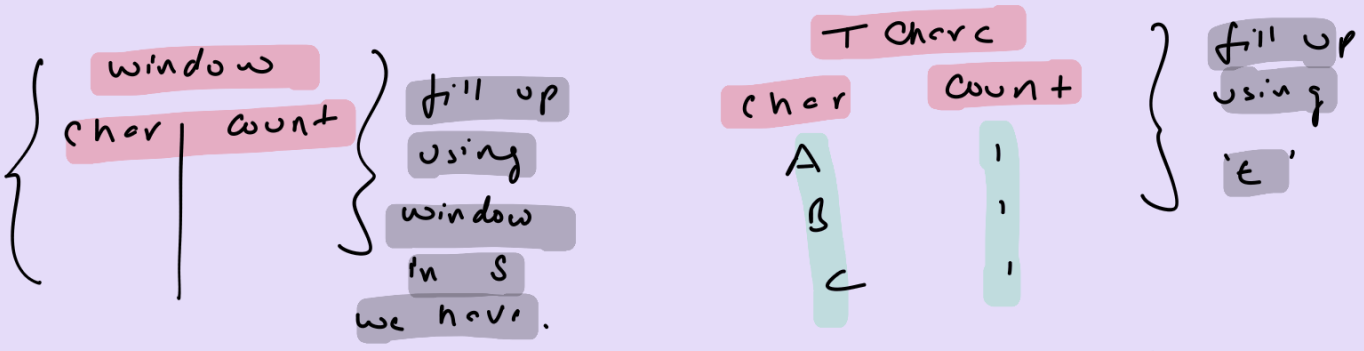
51) Min window substring:

Given 2 strings s & t. find min window in s that contains all t.

eg: s = 'ADOBECODEBANC' t = 'ABC'

o/p: 'BANC'

Soln: use 2 hashmaps & conditions.



- for calculation: maintain two variables one for each hash.

for window → have

for Tchar → need

fix need: sum (all char in t)

- here need = 3

Updation of have. start: have = 0

if window [char] == Tchar [char]

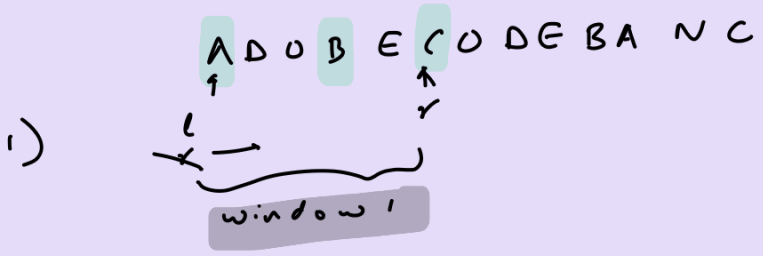
i.e in window if a character comes & the character is in T & in the window it appears some ~~#~~ of time or Tchar then have ++

- once have == need: Condition satisfy.

store res pointer

now we can shift l pointer & check.

- while shifting left counter if char is lost (which was in t) we decrement i: from window hash.



have == need.

2) r+: ADOBECODEBANC

has != need

3) r+ many time.

ADOBECODEBANC

has == need.

l+1

ADOBECODEBANC

still has == need.

ADOBECODEBANC

has == need.

(fail) → r+

Similarly we keep on going until r reaches

(2) Group Anagrams:

I/p: ["eat", "tea", "tan", "ate", "nat", "bat"]

o/p: [["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]

→ soln: make a hashmap.

for each string create an array of len 26.

[0] x26 → This will help make a fingerprint of word.

- iterate each character of word &

arr [ord(char) - ord(a)] = 1

∴ if char = a arr[0] = 1

- put this array in hashmap & append characters that follow.

51) Valid parenthesis:

eg: '( )' — True '[{ }]' - True.

'([ ] }' - False

soln: using stack.

if opening parenthesis — push in pop.

if closing parenthesis:

if stack has something &

Stack[-1]: opening of this using.

stack.pop()

else return.

(4) Valid Palindrome:

'A man, a plan, a canal: Panama' - True

'amanaplanacanalpanama'

- only alphanumeric.

Soln: def a fn for alphanum (c)

(ord(A) ≤ ord(c) ≤ ord(z)

ord(a) ≤ ord(c) ≤ ord(z)

ord(0) ≤ ord(c) ≤ ord(9))

move left & right until correct

point to alphanumeric.

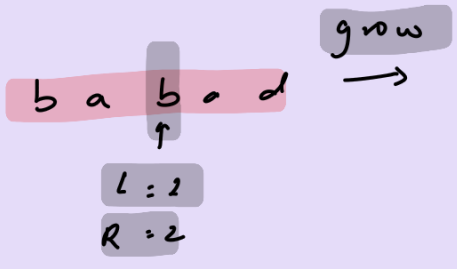
(5) longest Palindrome Subsequence:

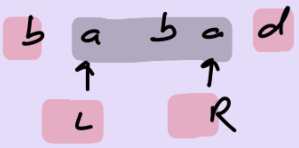
Given string s find longest palindrome:

eg: 'babad' → 'bab'

eg: 'cbbd' → 'bb'

→ soln: we start with an index & grow from there using L & R pointer.





rs: 'aba'

- similarly do for all index.

Cases: for odd → L = i & R = i

for even - L = i & R = i+1

5(i) Palindromic substrings :

q : 'abc' → op : ['a', 'b', 'c']

q : 'aaa' → op : ['a', 'a', 'a', 'aa', 'aa', 'aaa']

57) Encode Decode a string.

q : ['lint', 'code', 'love', 'you']

write encode & decode fn to get back same array.

→ soln : build a delimiter such that it can help.

q : 4# lint 4# code 4# love 3# you → encode

for decode

i : 0 → move i until we fin #

(i : j) → length.

move i to j + 1 + length.

51) Max Depth of a binary Tree -

→ soln :

1) recursive dfs : 1 + max(dfs(left), dfs(right))

2) iterative dfs.

3) Bfs (we count levels)

ie no of time we push in children of node.

59) Same Tree : find if both tree are same.

Soln → dfs

Time complexity O(n+m)

n : # nodes in Tree1

m : # nods in Tree2

Cases : if both are null - Tree are same

if any one null - false

if val don't match. false.

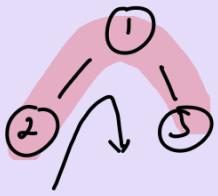
60) Invert Tree :

→ Soln : Dfs & swap left & right.

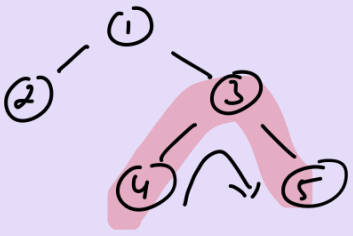
```
invert (root)
    not root : return
root.left, root.right = root.right, root.left
invert (root.left)
invert (root.right)
return root.
```

60) Binary Tree Maximum Path Sum.

Ex.



o/p: 2+1+3 = 6

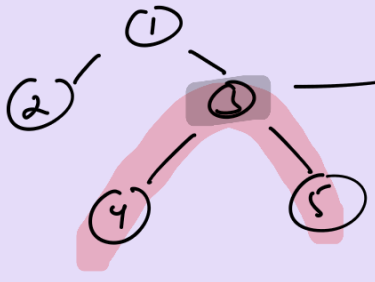


o/p: 4+3+5
= 12

→ soln: - finding sum including parent, left & right

- finding sum of the parent.

4:



here including parent

∴ root.val + (root.left) + (root.right)

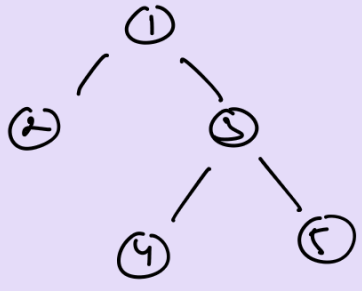
= 12

@ 3 if split then either 4 or 5 → max select

= 5

- don't take -ve values.

Breakdown:



@ 3 if consider - 4+3+5 = 12

@ 3 if find next max

so 3+5 = 8

- if we are 1

(i) 1.val + 1.left + 1.right

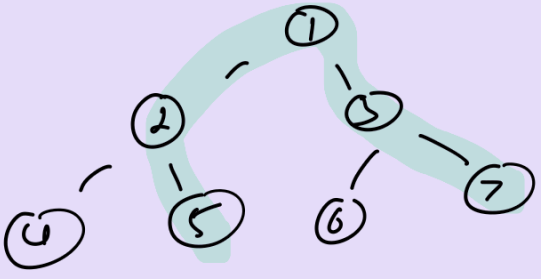
= 1 + 2 + (@ 3 without all)

= 1 + 2 + 8 = 11

∴ for should calculate considering the node + children but return only without considering

ie @ 3 → return 3+5 = 8

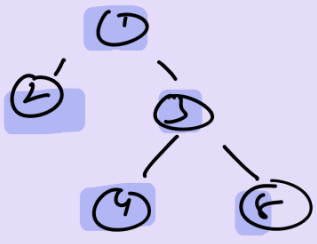
& calculate 4+3+5 = 12 & store.



- o/p = 18

(2) Binary Tree Traversal:

Ex:



o/p = [[1], [2,3], [4, 5]]

- Soln: BFS.

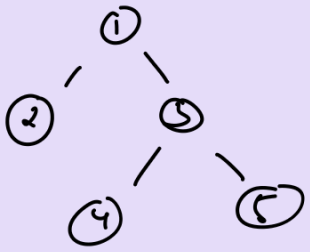
- at each iteration the queue will contain nodes at same level.

1: queue: [1] len_q = 1 → Traverse & add this node in res

2: que: [2, 3] len_q = 2 → Traverse & add the node in res & children

3: que: [4, 5] len_q = 2

63) Serialize & De serialize binary Tree -

eg: 

Serialize → '1, 2, N, N, 3, 4, N, N, 5, N, N'

↓ deserialize

- for serialization: Use dfs

- for deserialization.

make a pointer i = 0 & start recursion

if 'N' found → i++ & return None

else: node = TreeNode (int (serial [i]))

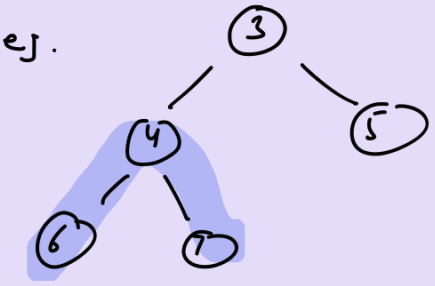
i + 1 = 1

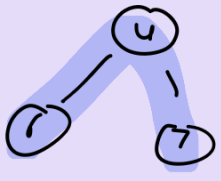
fill left right of this node

node.left = recursive-fn( )

node.right = recursive-fn( ) } i is already moving.

64) Subtree of tree. Given s and t return true if t is subtree of s.

eg. 

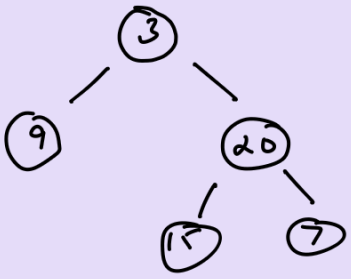
t: 

→ True.

- use same tree logic to find two trees are same.

- run the fn at each node's left & right with t & return True.

65) Construct tree when in order & preorder is given.



preorder : [3, 9, 20, 15, 7]

inorder : [9, 3, 15, 20, 7]

→ Soln : preorder : [3, 9, 20, 15, 7]

This is always root node [0]

- To decide left and right subtrees we'll look at inorder

[9, 3, 15, 20, 7]

pre[0] is here (mid)

- everything left of mid go in left subtree & right of mid go in right subtree

1st : fn ([3, 9, 20, 15, 7], [9, 3, 15, 20, 7])

node : Node (pre[0])

mid : inorder.index (pre[0])

node.left : fn (preorder [1: mid+1], inorder [:mid])

node.right : fn (preorder [mid+1:], inorder [mid+1:])

Runs: preorder: [3, 9, 20, 15, 7]

inorder = [9, 3, 15, 20, 7]

pt 1:

node: N(3)

mid: 1 (at 1st loc in inorder)

→ mid: 1 tells 0th item will go in left

& 2nd to last will go in right

This means: 1 item will go in left & 3 in right

node.left: fn([9], [9])

node.right = fn([20, 15, 7], [15, 20, 7])

node.left: N(9)

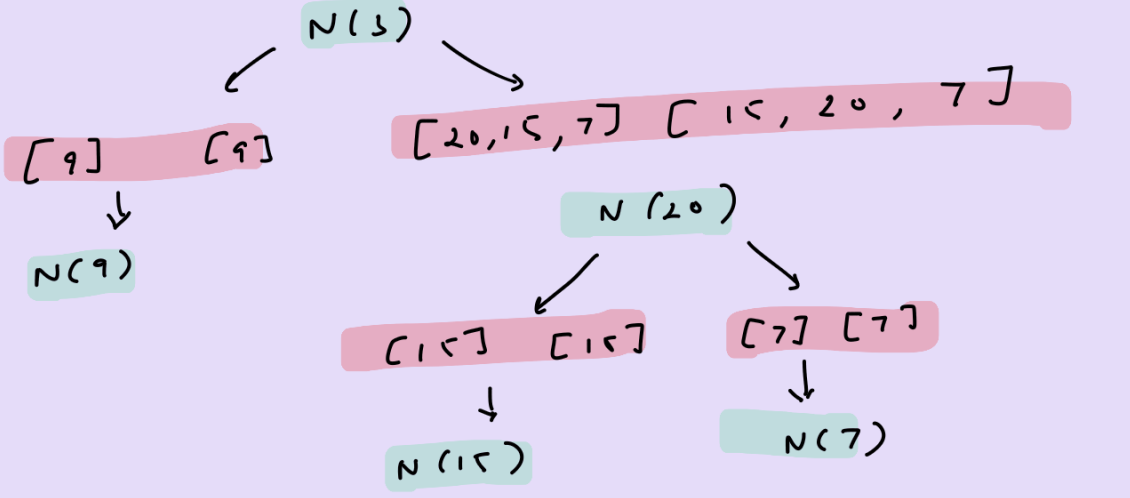
node.right: N(20) — 1 will go in left

mid: 1 1 in right

N(20).left: fn([15], [15])

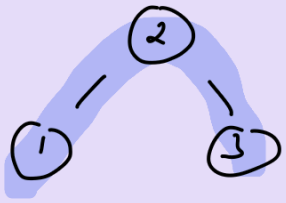
N(20).right: fn([7], [7])

[3, 9, 20, 15, 7] [9, 3, 15, 20, 7]



66) Valid Binary Search Tree.

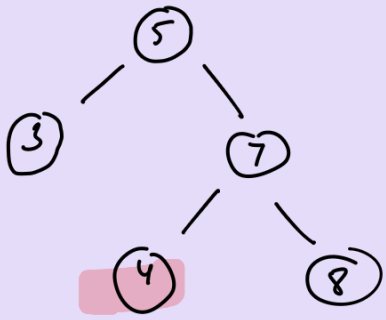
Ex:



True.

$1 < 2 < 3$

Ex:



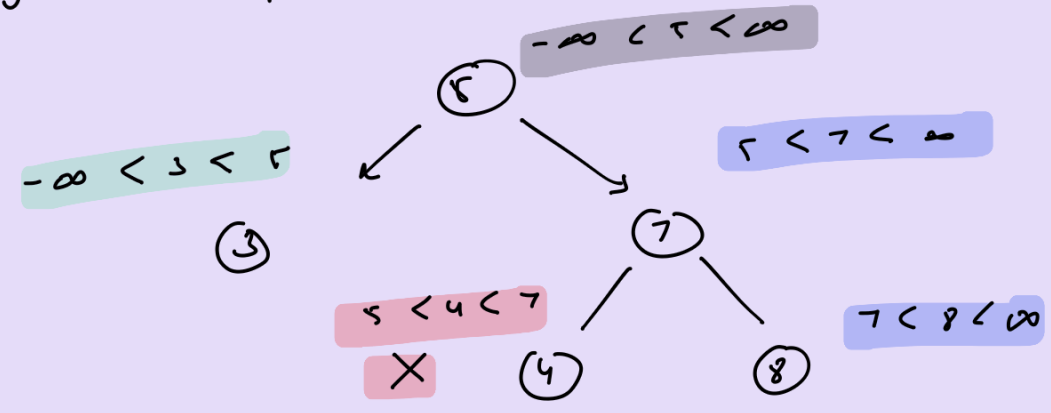
$3 < 5 < 7$

False

but every element in right subtree has to be > 5

@ 4 it fails.

logic: Compare with limit.



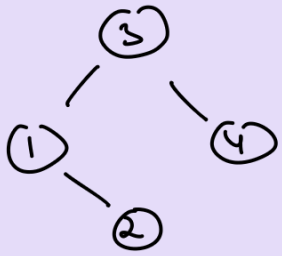
begin with $-\infty$ to ∞

on left side limit is prev left limit to prevnode

on right side limit is prevnode val to prev right node

67) Kth Smallest Element in BST.

Ex:

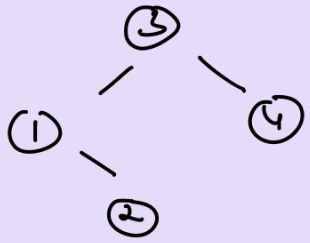


k = 1 → o/p = 1

k = 2 → o/p = 2

Soln → use inorder traversal.

- soln : use a stack.



stack = [ ]

Iterative approach.

cur = root

while cur and stack :

go to left Untill we get None.

while cur :

stack . append (cur)

cur = cur. left

once None we pop

count # pops

cur = stack . pop( )

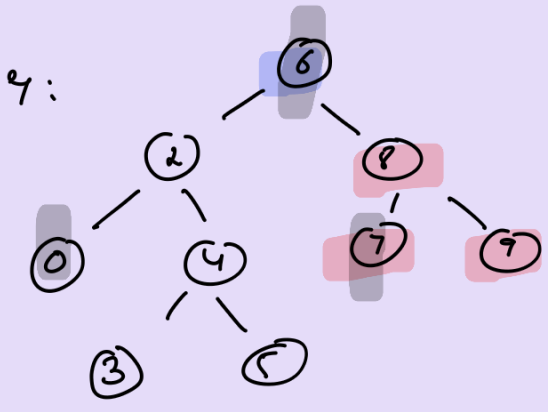
n += 1

if n = k :

return cur. val → return

cur : cur.right → go right.

68) Lowest Common Ancestor :



- 6 is LCA for itself.

p : 7 q : 9

LCA : 8

p : 0 q : 7

LCA : 6

- we need to look where break happen :

- we have to traverse to correct subtree & check

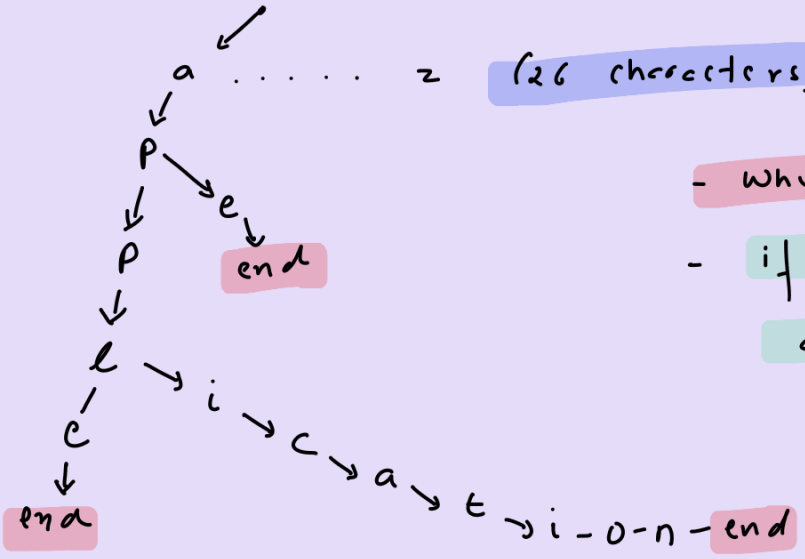
if : p & q < root.val → go to left

p & q > root.val → go to righ

if p < root.val < q → soln.

69) Trie (prefix Tree) -

implement Trie, insert, search, starts with



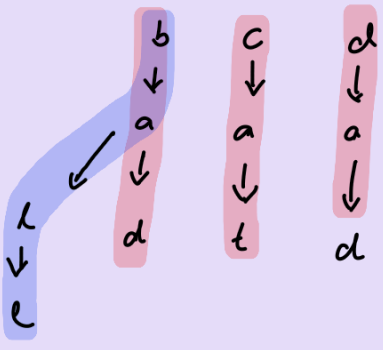
- why starts with is better ?
- if list of Im then all have to be iterated but here only 2 ?

TrieNode : have hashmap of children & endword flag.

```
class TrieNode:
    children = {}
    endword = False
```

```
class Trie:
    self.root = TrieNode()
```

70) Design a word dictionary to insert & search. the search fn can have word or dotted word.



search (bad) - True

search (b..) - True (bad)

search (.ad) - True (bad, dad)

→ soln: use above logic of Trie

- in search use combination of dfs & iterative

dfs - when dot appear

iterative - when dot not there

eg. def dfs (j, root)

. j is index of word we are at

- for iterative i → (j, len(word))

7) Word Search II: Given a list of word & grid of alphabets. find the words in list present in grid.

eg. o a a n
e t a e
i h k r
i f l v

words: ['oath', 'pea', 'eat', 'rain']

o/p: ['oath', 'eat']

→ soln: i) if we do directly dfs on grid Time complexity will be high

$O(w \cdot 4^{max})$

↓
4 dirns

ii) create a Trie using list of word.

- start dfs on grid if r, c in range and character on grid present in children of node.

72) Top K Frequent Elements:

Ip: nums: [1, 1, 1, 2, 2, 3] : K = 2

Olp: [1, 2]

→ Soln: use bucket sort

1) count values using hashmap
2) fill up freq array using hashmap
3) from back of freq array start exploring.

S1: hash: { } freq: [ ] [ ] [ ] [ ] [ ] [ ] [ ]

S2: hash: { 1: 3, 2: 2, 3: 1 }

freq: 0 1 2 3 4 5 6 (0 to 6)

3 2 1

S3: res: [1, 2]

73) Find Median from Data Stream:

[2, 3, 4] → 3 [2, 3] = 2.5

→ Soln: Use max & min heaps.

Heap: finding min value & max value in min & max heap → O(1)

- insertion & deletion - O(log n)

- Consider 2 heaps.

heap 1
max heap

heap 2
min heap

odd(3) → [3] | []

odd(2) → [3, 2] | []

- condition is: len can be max 1 different

[3, 2] | []

max value

[2] | [3]

odd(7) → [2, 7] | [3]

now max value of heap 1 ≤ min value of heap 2

[2, 7] | [3]

7 ≤ 3

[2] | [3, 7]

odd(4) → [2, 4] | [3, 7]

4 ≤ 3

[2, 4] | [3, 7]

[2] | [3, 7, 4]

now len(1) − len(2) = 2

[2] | [3, 7, 4]

[2, 3] | [7, 4]

3 | 4

⇒ (3+4)/2 = 3.5

3) Valid Anagram -

'tar' 'rat' → True

soln: 1) make 2 hashmap of count.

- Compare key value pair.

2) Sort & find.

4) Sum two numbers without + & -

→ soln: 2 + 3 → 5

2 : 0 1 0

3 : 1 1 0

5 : 1 0 1

summing →

a ^ b (XOR) = same → 0 different (1)

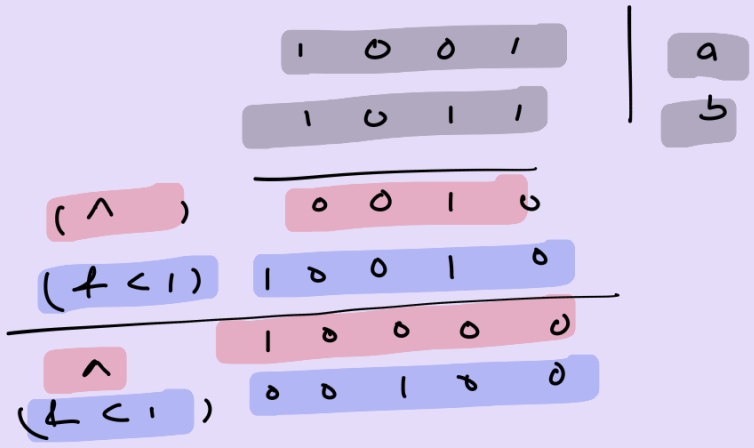
(above solns)

we also need carry.

- carry comes only when we odd 1 & 1

a & b - carry but should be left shifted by 1

4:



1 0 1 0 0 = 20 soln

(d < 1) 0 0 0 0 0 — stopping criteria.