S. No.	Date	Title

Q1 $O(mn)$. $O(1)$

LC.73: Set Matrix Zeroes

Given an mxn integer matrix, if its a[i][j]=0 then set entire row & col to 0. [Constant Space Soln]

0★	1	2	0★
3	4	5	2
1	3	1	5

⇒

0	0	0	0
0	4	5	0
0	3	1	0

APPROACH ⇒ • We will use the first cell of each row & each col. to note if a zero has been encountered in that row or column.
• The problem then becomes having an extra variable to store state of row 0 as a[0][0] would already be used for col 0 state storage. And then do a reverse sweep to set zeroes [reverse because we don't want to lose state holding cells]

```
for row in range(m):
    for col in range(n):
        if mat[row][col] == 0:
            if row == 0 : row0 = True
            else  mat[row][0] = 0
            mat[0][col] = 0
```

Next ↗

```
for row in [m-1, -1):
    if row == 0:
        if row0 == True:
            for col in [0,n): mat[row][col]=0
    else:
        for col in [n-1, -1):
            if mat[0][col]=0 or mat[row][0]==0 : set mat[r][c]=0
```

classmate

Q2. $O(n^2)$ $O(1)$

LC·118 Pascal's triangle

Print the first n rows of Pascal triangle

APPROACH

> There are n levels from [1→n]
> each level i has, i elements [1-based]
> The first & last element has always value 1
> every other index of current level would be sum of prevlvl[index-1] + prevlvl[index]

CODE ⇒

```
ans = []
For level → [1, n]:
    for elem → [1, level]
        if elem = 1 or elem = n:
            curlvl[elem-1] = 1
        else
            curlvl[elem-1] = prevlvl[elem-2]
                           + prevlvl[elem-1]
    ans.append[curlvl]
return ans.
```

Q3 $O(n)$ $O(1)$

LC·53 Maximum Subarray [Kadane's Algo·]

Return the largest sum of a contiguous subarray [with one or more element] from a given array.

[5, 4, -1, 7, 8] ⇒ 23
[-2, 1, -3, 4, -1, 2, 1, -5, 4] ⇒ 6 i.e. [4, -1, 2, 1]

APPROACH ⇒ • a negative previous sum can only worsen further sums so no point carrying it further, reset it
• Initialise both current & max as first element as the answer would atleast have 1 element.

CODE

```
curSum, maxSum = arr[0], arr[0]
for num in arr[1:]:
    if curSum < 0:
        curSum = 0
    curSum += num
    maxSum = max(maxSum, curSum)
return maxSum
```

Q4 O(n) O(1)

LC:31 Next Permutation

Print next lexicographically greater permutation of given array.

[1,2,3] ⇒ [1,3,2] [3,2,1] ⇒ [1,2,3]

Approach.

- Every permutation has some increasing prefix subarray. [Except for totally descending permutation like 3,2,1]
- In totally descending case we return nums.sort()
- In every other case, we replace the very last element of increasing subpart with an element JUST GREATER than it from the non-increasing subpart.
- After swapping, we reverse the non-increasing subpart to get it in its lowest lexicographical state.

CODE ⇒ → traverse from back

```
for i in [n-2, 0]:
    if a[i] < a[i+1]:
        for j in [n-1, i):
            if a[j] > a[i]: swap a[j], a[i]
        reverse a[i+1:]
        return a
else return a.sort() → only runs if totally descending
```

Q5 O(n) O(1)

Sort array of 0's 1's and 2's [Dutch National Flag]
[Sol" using pointers]

APPROACH → Use 3 pointers low, high & mid

low → eventually has indices before which 0's are present.

mid → indices before which 0's & 1's are present and from & beyond which 2's are present.

End → index from which 2's are present.

Now keep moving mid from 0 till end pointer.

CODE

```
while mid <= end:        ← equal is needed in case last iter has cur value 0 in place
    if nums[mid] == 0
        nums[mid], nums[low] swap
        mid++, low++
    else if nums[mid] == 1
        mid++
    else if nums[mid] == 2
        swap nums[mid], nums[end]
        end--
```

Q6 O(n) O(1)

LC. 121 Best Time to Buy and sell Stock

Return max achievable profit (by buying & then selling)

Approach → > It makes sense to pick buy date as the minimum price date encountered in past while considering current day as date of sale.

> Then store the maximum possible profit encountered in all cases.

CODE:-

```
maxProfit, minSeenBefore = 0, inf

for price in prices:

    maxProfit = max(maxProfit, price - minSeenBefore)
    minSeenBefore = min(minSeenBefore, price, minSeenBefore)

return maxProfit
```

Q7 O(n log n) O(n) ← sorting needs space

L.C. 56 Merge Intervals

Given array of intervals with intervals like [start, end], merge overlapping intervals and return non-overlapping merged ones.

[[1,4],[4,5]] ⇒ [[1,5]] we should

Approach ⇒ > Consider intervals which start earlier than others before we process the rest of intervals.

> To do that sort the intervals on basis of start time.

> If for current interval start time > end time of previous, then no overlap is possible

> for other case merge the two intervals and note the merged interval down by adjusting prev.

CODE:-

```
intervals.sort(key = start-time)
ans, prev = [], intervals[0]
for interval in intervals[1:]:
    if interval.start > prev.end:
        ans ← push prev, prev = interval
    else: prev = [prev.start, max(prev.end, interval.end)]
ans.append(prev) # 1 interval always left in prev
return ans.
```

$O(n^2)$ $O(1)$

L.C. 48 Rotate Image

Given a nXn matrix, Rotate it in-place

1	2	3
4	5	6
7	8	9

⇨

7	4	1
8	5	2
9	6	3

Approach ⇒ The main observation is to note that Rotation operation is equivalent to transpose and then flipping across vertical axis of matrix.

Code :-

```
for i ∈ [0, n-1]
    for j ∈ [i+1, n-1]:
        swap a[i][j] and a[j][i]
for col ∈ [0, n//2):
    for row in [0, n-1]
        swap a[row][col], a[row][n-1-col]
```

$O(n+m)$ $O(n+m)$

Knuth Morris Pratt Algorithm (LPS Based)

LPS → Longest Proper Prefix which is also suffix.

string	a	a	b	a	a	c	a	a	b	a	a	d
LPS	0	1	0	1	2	0	1	2	3	4	5	0

The value here is 2 because aa on both ends.

> For algorithm to work lps[index 0] = 0 always.

ALGO →

```
i = 1, len = 0
while (i < length of string):
    if s[i] == s[len]:
        len += 1
        lps[i] = len
        i++
    else
        if (len > 0):
            len = lps[len-1]
        else
            lps[i] = 0
            i++
```

Multiply by 2 while subtracting to account for pattern appended before # & also to find index of starting point

where Kmp[i] is length of pattern

To actually search pattern →

```
Kmp = getLPS(pattern + "#" + string); return i - len(pattern)*2
```

where i is idx in Kmp

O(n+m) O(1)

Merge two sorted arrays in Constant Space

Two arrays a & b are given with length m & n respectively. 'a' also has n extra positions in end. Place both array's elements in 'a' in sorted order.

Approach → Keep a last index and place the greatest elements of tail end of both array's in that last position and then decrement indices accordingly.

CODE →

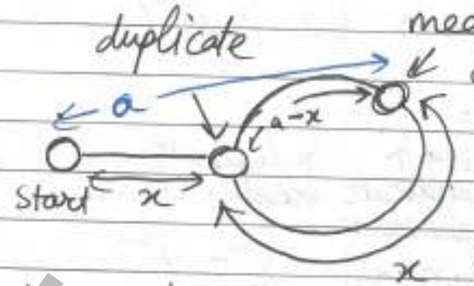
```
i = m-1 , j = n-1
k = m+n-1
while j >= 0
    if i >= 0 and a[i] >= b[j]:
        swap a[k], a[i]
        i--
    else:
        swap a[K], b[j]
        j--
    always K--
return a
```

O(n) O(1)

Find duplicate in assay of integers

There is only one repeated number in a given array of integers (can be repeated multiple times though). Find the repeated element and don't modify the array and use only constant space

Approach



'a' is always divisible by length of cycle

Distance by slow pointer → a

~~fast pointer → 2a~~

so move one pointer back to beginning and travel one by one x times to reach start of cycle.

CODE →

```
slow, fast = nums[0], nums[0]
while True:
    fast, slow = nums[nums[fast]], nums[slow]
    if fast == slow: break
slow = nums[0]
while fast != slow: fast, slow = nums[fast], nums[slow]
return slow (or fast)
```

O(n) O(1)

Repeat and Missing Numbers

a N length array has elements from 1 to N except one is missing and some other element is duplicated. Find both the missing & duplicated element.

APPROACH → Two approaches.

① Mathematical approach.

let two target values be X & Y

$\frac{n(n+1)}{2} - sum(array) = X - Y$

(missing elem ↑, duplicate ↑ missing elem)

$\frac{n(n+1)(2n+1)}{6} - sum(squares(array)) = X^2 - Y^2$

Now we can solve for X & Y

② XOR approach

(i) we find $X \wedge Y$ by doing $xor(array) \wedge xor[1 \dots n]$

(ii) now find any set bit in $X \wedge Y$, the two numbers differ here.

(iii) Now divide array in two buckets, in those buckets also put numbers from 1 to n which have bitset from step 2.

(iv) the values of two buckets in end would be X & Y.

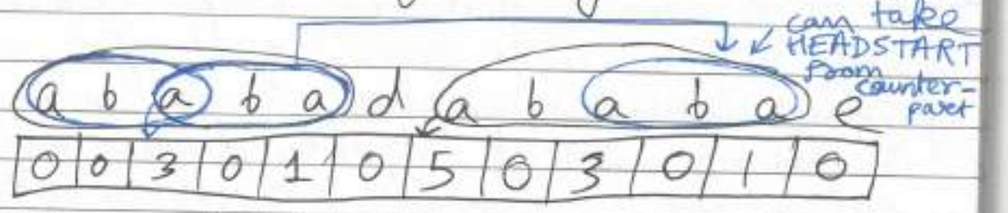
classmate

O(n+m) O(n+m)

Z-Algorithm

APPROACH → Z-array → length of longest substring which is also prefix of the whole string starting at THAT INDEX

> Z Array[0] = NA always (Not defined) we initialize it to 0 generally but it's not used.



> Obviously headstart can only be given till end of range as values beyond r are unknown. So headstart has an upper limit. ↳ right boundary.

> Algo has 3 steps → ① take headstart from known ② Grow beyond known ③ Define new known

CODE ⇒ l, r = 0, 0

```
for i = 1 ... n :
headstart [ if i <= r : z[i] = min(z[i-l], r-l+1)        (from counterpart; can't go beyond known)
from unknown [ while i+z[i] <= n and s[i+z[i]] == s[z[i]]:
                   z[i] ++
grow known [ if i+z[i]-1 > r:
                 l = i, r = i+z[i]-1
```

For pattern search return i - needle.size() - 1 (needle → pattern)

classmate

O(n) O(1)

Majority Element (Moore's Voting Algo)

There's a number which is present more than the rest of the numbers combined in an array. Find this special number in O(1) space

APPROACH → We can use Moore's counting algorithm
The idea is that we keep getting answer to new value once count of previous answer expires. If the answer is majority element, it would cancel out other elements in the array with its own count and eventually emerge as the answer.

CODE →
```
ans, count = nums[0], 1
for i in range [1, n):
    cur = nums[i]
    if count == 0:
        ans = cur
    if ans == cur:
        count++
    else:
        count--
return ans
```


O(log(n)) o(log(n))

Recursive Exponentiation

Efficiently implement pow(x,n)

APPROACH :- > We use recursion and use divide and conquer by splitting given problem to 2 equal smaller subproblems wherever possible.
> We also take reciprocal of x if n is negative.

CODE :-
```
if n<0:
    x = 1/x
    n = -n
def powerRecursive(x,n):
    if n == 0: return 1
    if n%2 == 1:
        return x * powerRecursive(x, n-1)
    else:
        return powerRecursive(x,n/2)**2
return powerRecursive(x,n)
```

$O(\log(m)+\log(n))$ $O(1)$

Search in a sorted 2-D Matrix

1	(3)	5	7
10	11	16	20
23	30	34	60

target = 3

Write an efficient search algorithm

APPROACH →> We can see that minimum element and max element of each row are present in first and last index respectively.
>∴ we can use binary search twice to first decide row index & then to decide column index.
m×n matrix

ALGO → • Set $l=0, r=m-1$
• do binary search to get row by comparing min & max val with target
• If row is defined and present, then do binary search again now in that specific row by setting $l=0$ and $r=n-1$
• If found target report True
• If row not defined or target not found, return False.

$O(n\log n)$ $O(n)$

Inversions of Array

Form1: Find number of pairs in array such that $i<j$ & $a[i]>a[j]$
Form2: Find the minimum number of swaps of consecutive elements to sort array.

APPROACH :-> Merge sort Modification: Do normal merge sort but also have a inversion count which gets incremented every time elements in left subpart > elements in right subpart
>NOTE :- Only declare minimal space temp array instead of cloning full array to not get TLE!

CODE :-

```
ans = 0
def mergeSort(l,r):
    if l<r:
        mid = (l+r)//2
        mergeSort(l, mid), mergeSort(mid+1, r)
        temp = [None]*(r-l+1), k=0, i=l, j=mid+1
        while i<=mid and j<=r:
            if a[i]<=a[j] => temp[k++] = a[i++]
            else : ans += (mid-i+1), temp[k++] = a[j++]
        while i<=mid : temp[k++] = a[i++]
        while j<=r : temp[k++] = a[j++]
        Copy temp into a[l:r+1] (end index not inclusive)
mergeSort(0,n-1), return ans.
```

O(n) O(1)

Majority Element-2 (Modified moore's voting)

Find those numbers in array which occur more than $\lfloor n/3 \rfloor$ times.

APPROACH → At max only 2 numbers can be present more than $\lfloor n/3 \rfloor$ times. so we find possible answers by modifying moore's voting algorithm.

▶ We will need to verify the possible answer and also check for duplicates.

CODE :-
```
ans, n1, n2, ct1, ct2 = set(), -1, -1, 0, 0
for i in nums:
    if i==n1 : ct1++
    else if i==n2: ct2++
    else if ct1==0: n1=i, ct1=1
    else if ct2==0: n2=i, ct2=1
    else:
        ct2--, ct1--
verify(n1)
verify(n2)
return ans
```

```
def verify(num):
    if nums.count(num) > n//3:
        ans.add(num)
```

classmate

O(m)+n) O(1)

Unique Paths

m = 3, n = 7

Print number of possible paths from top left to bottom right comprising of down or right moves.

APPROACH :-> If m == 1 or n == 1, there is only one way (all down or all right)

> In other cases our routes comprise of m-1 downs and n-1 rights. So we can apply combinatrics to find answer.

```
if m==1 or n==1: return 1
m-=1, n-=1
return factorial(m+n) // [factorial(m)*factorial(n)]
```

classmate

O(n log n) O(n)

Reverse Pairs

Find number of pairs (i,j) in array where $0 <= i < j < n$ and $num[i] > 2 * nums[j]$

APPROACH :> We modify merge sort and do the calculation of inversions in middle of a merge operation.

CODE :-
```
ans = 0
def ms(l,r):
    if l<r:
        mid = (l+r)//2
        ms(l,mid), ms(mid+1,r)
        i,j = l,mid+1
        temp = [0]*(r-l+1)
        while i <= mid:
            while j<=r and nums[i] > nums[j]*2:
                j += 1
            ans += j-mid - 1, it = 1
Now do rest of normal merge sort process :-
    ① populate temp with smallest
    ② Copy rest of left subpart & right subpart
    ③ Copy temp to array back.
return ans after increasing it on 0 to n-1
```

O(n) O(n)

LC.1 Two Sum

Return pair of indices in array having sum equal to given target

nums = [2,7,11,15], target = 9
Ans → [0,1]

APPROACH →> Store already encountered numbers and their corresponding occurences in an unordered map. From that do lookups of target - current val to see if it exists

CODE :-
```
for idx, num in enumerate(nums):
    if target-num in map:
        return [idx, map[target-num]]
    map[num] = idx
```

$O(n^3)$ $O(1)$

LC·18 4Sum

Return distinct quadruplets from array summing up to given target, using constant space.

APPROACH :-> We will need to sort the array first if we want constant space (quick sort)

> We will need to jump indices with repeating values to avoid duplicates without having to use hashset.

CODE → nums.sort(), i=0
while i<n:
  I = nums[i], j = i+1
  while j<n:
    J = nums[j], l = j+1, r = n-1, pre = I+J
    while l<r:
      L, R = nums[l], nums[r], temp = L+R
      if temp == target - pre:
        ans.append([nums[i], nums[j], L, R])
        move l forward dodging duplicates
        and r back dodging duplicates.
      else: move one of l and r to adjust
    move j right dodging duplicates
  move i right dodging duplicates

$O(n)$ $O(n)$

LC·128 Longest Consecutive Sequence

Given an unsorted array, return length of longest consecutive elements sequence

APPROACH 1 > sort nums and count max consecutive. (nlogn)

APPROACH 2 → O(3n) using unordered set

Only start from elements that don't have a predecessor, and do a linear traversal starting from those numbers and count longest streak.

CODE → hashset = set(nums)
longest = 0
for num in nums:-
  if (num-1 not in hashset):
    cur = num, curstreak = 1
    while curr+1 in hashset:
      cur += 1
      curstreak += 1
    longest = max(longest, curstreak)
return longest

O(n) O(n)

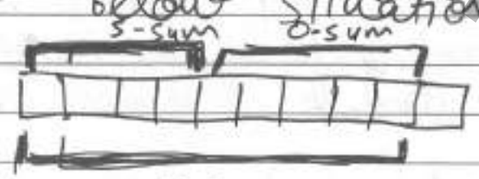
Largest Subarray with 0 sum

print length of longest subarray with 0 sum.

15, -2, 2, -8, 1, 7, 10, 23

Answer → 5

APPROACH :- Assume any subarray with below situation.



this initialisation is needed for 0sum subarrays starting from beginning of array

If prefix-sum of 2 subarrays from start is equal the difference of those 2 subarrays is a sub-array of 0 sum

CODE →
```
prefix, earliestOccurence = 0, {0: -1}
longest = 0
for idx, num in enumerate(arr):
    prefix += num
    if prefix in earliestOccurance:
        longest = max(longest, idx - earliestOccurance[pre])
    else
        earliest[pre] = idx
return longest
```

O(n) O(1)

Middle of Linked List

Given a linked list, find its middle, in case of even length list, return second middle node.

APPROACH :- Use fast and slow pointer

CODE →
```
slow, fast = head, head
while fast and fast.next:
    slow = slow.next
    fast = fast.next.next
return slow
```

O(n) O(1)

Reverse Linked List

Given the head of a linked list, reverse it and return new head.

APPROACH ⇒ Keep track of previous node and assign next of cur to prev, and keep repeating the process.

CODE

```
prev = None, temp = head
while temp:
    nex = temp.next
    temp.next = prev
    prev = temp
    temp = nex
return prev.
```


O(n) O(1)

LC.3 Longest Substring without repeated characters

Find length of longest substring without any repeated characters

APPROACH ⇒ Sliding window

> Keep growing right boundary if no duplicates are found.
> Remove elements from left until duplicacy is removed.

CODE ⇒

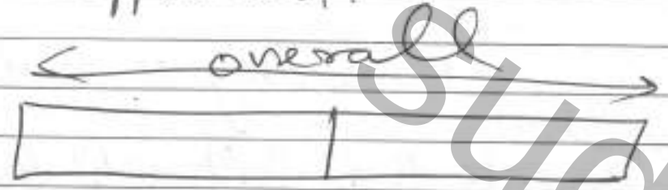
```
vis = set(), l = 0, r = 0, ans = 0

while r<n:
    if s[r] not in vis:
        vis.add(s[r])
        r += 1
    else:
        vis.remove(s[l])
        l += 1
    ans = max(ans, r-l)
return ans
```

O(n) O(n)

Subarray with given XOR value

Return number of subarrays possible from given arr with provided XOR values.

APPROACH ⇒ prefix computation based approach:

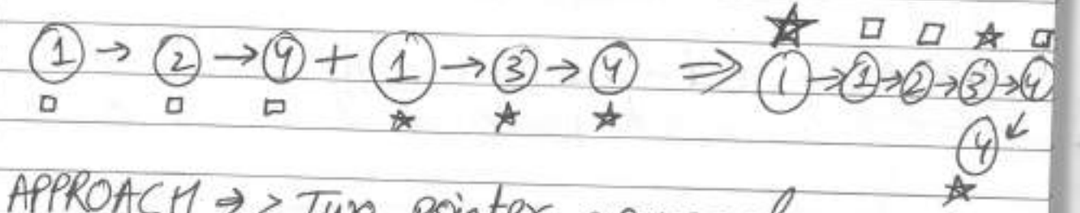



```
xorr = 0, ans = 0, cache = defaultdict(0)
for num in arr:
    xorr ^= num
    ans += cache[xorr ^ target]
    if xorr == target:        → account for case where overall == target
        ans++
    cache[xorr] += 1
return ans
```

O(n1+n2) O(1)

Merge two sorted Linked Lists

Merge two sorted linked lists inplace by modifying next pointers of present nodes.



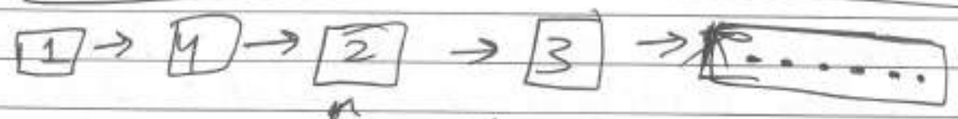
APPROACH ⇒ > Two pointer approach
> Keep a dummy head
> And a dummy prev → which goes through both lists combined in sorted order and is used to add new entries to that sequence using its next pointer.

CODE ⇒

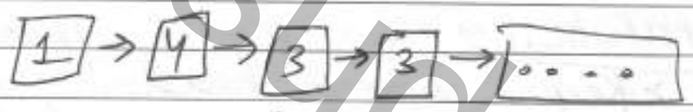
```
p1, p2, dummy = list1, list2, ListNode()
prev = dummy
while p1 or p2:
    if p1 != None and p1.val <= p2.val:
        prev.next = p1
        prev = p1
        p1 = p1.next
    else:
        prev.next = p2
        prev = p2
        p2 = p2.next
return dummy.next
```

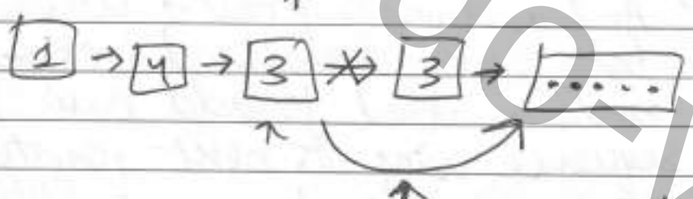
O(1) O(1)

Delete Given Node in Linked List



APPROACH ⇒ Hint - You can copy value of next node

step1 ⇒ 

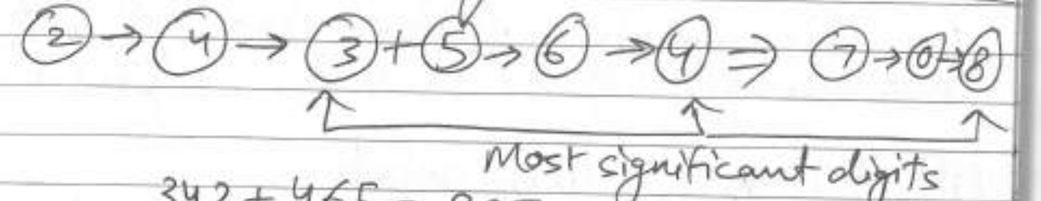
Step 2 ⇒ 

CODE ⇒
```
node.val = node.next.val
node.next = node.next.next
```

In case if node.next == null:
just set node = null

O(n1+n2) O(max(n1,n2))

Add two Numbers given in Linked List Form



342 + 465 = 807

APPROACH ⇒ maintain a dummy node for head of answer and keep adding answer terms as you keep generating them

> Also maintain carry variable for carry forwards

CODE ⇒
```
dummy = ListNode(), temp = dummy
p1, p2 = l1, l2
carry = 0
while p1 or p2
    cur1 = p1.val or 0, cur2 = p2.val or 0
    total = cur1 + cur2 + carry
    curVal = total % 10, node = ListNode(curVal)
    temp.next = node, temp = temp.next
    if total >= 10: carry = 1
    else: carry = 0
    if p1: p1 = p1.next
    if p2: p2 = p2.next
# In the end account for carry outside loop
if carry: temp.next = ListNode(carry)
return dummy.next
```

O(n) O(1)

Delete nth node from end of linked list

You are not given total size of list and you have to do in only 1 pass.

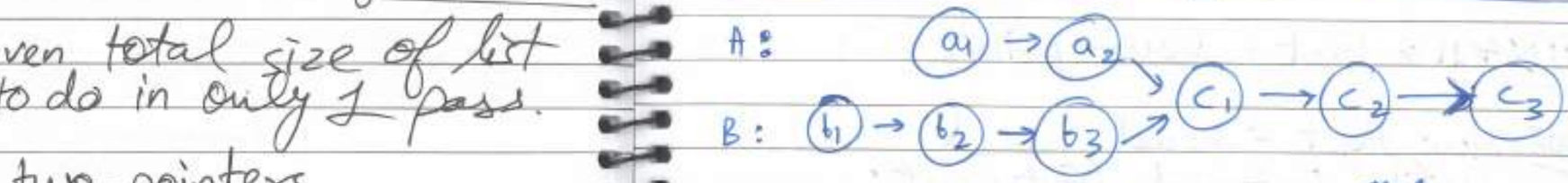
APPROACH ⇒ use two pointers.
> iterate 1 of those n times
> Now start iterating both together till one reaches end of list.
> the next of other pointer is node to be deleted, delete it as usual
> In case n == size of list, just return head.next (as head itself will be removed)

CODE ⇒
```
p1, p2 = head, head
while n: p2 = p2.next, n--
if not p2: return head.next
else:
    while p2.next:
        p2 = p2.next
        p1 = p1.next
p1.next = p1.next.next
return head
```

n=len case ←
delete step ←

O(m+n) O(1)

LC. 160 Intersection of 2 Linked Lists



APPROACH > start iterating both lists, one will end first if length is unequal. Lets say A ends first. Now make this null pointer as head of B [This will help in compensating for extra length of B, now iterate both pointers, till one ends, this marks successful completion of excess length process. Now make null pointer equal to head of shorter list, and MOVE BOTH POINTERS TOGETHER TO GET ANSWER

CODE :-
```
t1, t2 = headA, headB
while t1 and t2: t1, t2 = t1.next, t2.next
if t1 ends: shorterHead = headA
            t1 = headB
else: shorterHead = headB
      t2 = headA
while t1 and t2: t1, t2 = t1.next, t2.next
if t1 == None: t1 = shorterHead
else: t2 = shorterHead
while t1 != t2:
    t1, t2 = t1.next, t2.next
return t1
```

O(n) O(1)

Linked List Cycle Detection

APPROACH ⇒ Fast & slow pointers

CODE ⇒
```
slow, fast = head, head
while fast and fast.next:
    slow = slow.next
    fast = fast.next.next
    if slow == fast:
        return True
return False
```

O(n) O(1)

Check if linked list is palindrome

APPROACH ⇒ 1) Find middle (Ignore middle element is list is odd length for palindrome)
2) reverse second part
3) Compare first & reversed second part

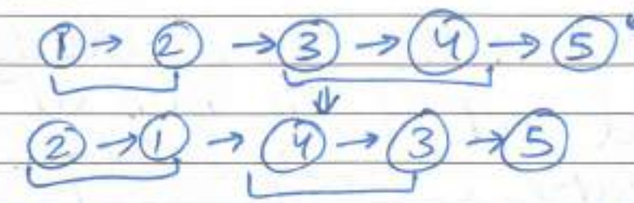
CODE:
```
def isPalindrome(head):
    middle, evenLength = getMiddle(head)
    if evenLength:
        p2 = middle
    else
        p2 = middle.next
    p1 = head
    p2 = reverse(p2)
    while p1 and p2:
        if p1.val != p2.val: return False
        p1, p2 = p1.next, p2.next
    return True
```

```
def reverse(head):
    prev = None, temp = head
    while temp:
        nex = temp.next
        temp.next = prev
        prev = temp
        temp = nex
    return prev
```

```
def getMiddle(head):
    slow, fast = head, head
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
    return slow, fast == None
    check for even length
```

O(n) O(1)

Reverse Nodes in K-Group



left out

K = 2

note: don't reverse left out nodes

APPROACH => > flip each subpart
> store last node of each step's reversed part [as that has to point at head of next reversed subpart]
> Use node of next subpart as delimiter to check end of current sublist.

CODE =>
```
l, r = head, head
ans, prevStepLastNode = head, None
while r:
    cnt = 0
    while r and cnt<K:
        r, cnt = r.next, cnt+1
    if cnt!=K  # leftout nodes
        if prevStepLastNode:
            prevStepLastNode.next = l
        return ans  # head by default if all nodes are leftout
    else: newHead = self.reverse(l,r)
        if prevStepLastNode: pSLN.next = newHead
        else: ans = newHead
    l = r  pSLN = l
on the end return ans
```

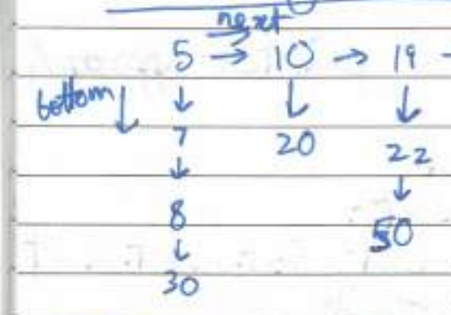

O(n) O(1)

Find starting point of loop of linked list

APPROACH => > fast and slow pointer approach

```
slow, fast = head, head
while fast and fast.next:
    slow, fast = slow.next, fast.next.next
    if slow == fast: break
else: return None
slow = head
while slow != fast:
    slow, fast = slow.next, fast.next
return slow
```

Flattening a Linked List



answer will be checked by traversing through bottom pointer

APPROACH > Use heap to keep track of next smaller node.
> Insert garbage value to deal with equal value nodes as of nodes are not comparable.

CODE ⇒

```
pointers = [], temp = root, dummy = Node()
i=0  # for preventing Node object comparison
while temp:
    heappush(pointers, (temp.data, i, temp))
    temp = temp.next , i+=1
temp = dummy
while pointers:
    cur_, curNode = heappop(pointers)
    temp.bottom = curNode
    temp = curNode, nex = curNode.bottom
    if nex != None:
        heappush(pointers, (nex.data, i, nex))
        i+=1
    curNode.bottom = None # clean up, not needed though
return dummy.bottom
```

Rotate List by K

K can be much greater than list length.

APPROACH ⇒ Find length of list and do K%n to reduce computation iterations.
> Change tail to point to head.
> now travel n-K times and break the chain there and repeat new head.

① → ② → ③ → ④ → ⑤, K=2 ⇒ ④ → ⑤ → ① → ② → ③

CODE ⇒

```
if not head: return head
lastnode, n = head, 1
while lastnode.next:
    lastnode = lastnode.next, n = n+1
lastnode.next = head  # join end and begin
K %= n
keepFromBegin = n-K
while keepFromBegin:
    lastNode, keepFromBegin = lastnode.next, keepfrombegin-1
newhead = lastnode.next
lastnode.next = None
return newhead.
```

O(n+K%n) O(1)

O(n) O(1)

Copy a Linked List with Random Pointer

Approach > Use next pointers of old list nodes as hashmap.
> Then establish random pointers for new nodes
> separate next pointers of two lists.

CODE:-
```
if not head: return None
temp = head
while temp:                                   # setup new nodes
    temp.next = Node(temp.val, temp.next)
    temp = temp.next.next
temp = head
while temp:
    temp.next.random = temp.random.next if temp.random else None
    temp = temp.next.next
newHead = head.next, cur = head
while cur:
    nextOldNode = cur.next.next, curNewNode = cur.next
    curNewNode.next = nextOldNode.next if nextOldNode else None
    cur.next = nextOldNode
    cur = cur.next
return newHead
```

O(n²) O(1)

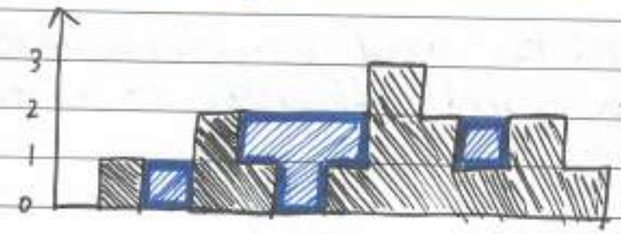
Three Sum (Avoid duplicate triples)

> Use an iter pointer and four new pointers
> Sort first to efficiently be able to avoid duplicate vals.

CODE:-
```
nums.sort(), ans = []
i, n = 0, len(nums)
while i<n
    l = i+1, r = n-1, iVal = nums[i]
    while l<r:
        L, R, moveL, moveR = nums[l], nums[r], False, False
        tot = iVal+L+R
        if tot>0:
            moveR = True
        elif tot<0:
            moveL = True
        else:
            ans.append((iVal, L, R))
            moveL, moveR = True, True
            while moveL and l<n and nums[l]==L:
                l++
            while moveR and r>=0 and nums[r]==R:
                r--
    while i<n and iVal == nums[i]: i++
return ans
```

O(n) O(n)

LC. 42 Trapping rainwater



Given the elevations, what's the most amount of water you can trap.

Approach → at every index, only think about your max storage possible
→ Use monotonic stack or DP to find max height on left and right of current index.
→ there can never be negative contribution by a index, so keep 0 minimum contribution.

CODE

```
for i in range(n):
    r = highestonRight(i)
    l = highestonLeft(i)
    ans+ = max(0, min(r,l) - height[i])
return ans.

@cache
def highestonLeft(i)
    if i==0: return height[i]
    else:
        return max(height[i], hol(i-1))

@cache
def hor(i)
    if i==n-1: return height[n-1]
    else:
        return max(height[i], hor(i+1))
```

height needs to be returned instead of 0 for dp to work for rest of indices.

O(nlogn) O(n) or O(1)

Fractional Knapsack

APPROACH → > greedy, pick element with max val per unit weight
> PICK AS MUCH of THE ITEM AS YOU CAN

CODE

```
Items.sort(key = lambda x: x.value/x.weight, reverse=True)
                                  → highest to lowest
for item in Items:
    value, weight = item.value, item.weight
    maxCanTake = min(weight, W)
                          → knapsack capacity
    W -= maxCanTake
    ans+ = value * (maxCanTake/weight)
                    Fractional Knapsack
return ans
```

O(n) O(1)

Remove duplicates from sorted Array

Modify array inplace, place answer in beginning and return number of uniques

[1,1,2] ⇒ 2, [1,2,_]
↳ 2 unique.

APPROACH ⇒ > 2 pointers approach
> Keep a third pointer, like K, which stores index till valid length.

CODE ⇒
```
l, r, K, n = 0,0,0, len(nums)
while r<n:
    while r<n and nums[r]==nums[l]:
        r+=1
    nums[K] = nums[l]
    K+=1
    l=r
return K
```

O(n) O(1)

Max consecutive Ones

Given an array of 0's and 1's, return max consecutive 1's in array

APPROACH → > greedy approach
> reset count whenever you see 0

CODE ⇒
```
cur, highest = 0,0
for num in nums:
    if num==1:
        cur+=1
    else
        cur=0
    highest = max(highest, cur)
return highest
```

Depending on sorting algo ⇒ O(nlogn) or O(n²) O(n) or O(1)

N meetings in one Room

Maximum number of meetings that can be conducted in one room without overlap.

APPROACH ⇒ do that meeting first which ends first
⇒ if non-overlapping meet is found, make that the new prev.

CODE ⇒

```
meets.sort(key = lamba x : x.end)
prev = meets[0], ans = 1
for meet in meets[1:]:
    if prev.end < meet.start : # no overlap
        prev = meet
        ans++
    else:
        continue
return ans
```


Time ⇒ O(coinTypeCount X target)
Space ⇒ O(target)

Minimum Number of coins

You have infinite supply of every coin type
If its impossible to get target value, return -1.

APPROACH ⇒ Just because complexity of dp recursion is mXn, doesn't mean its memo table size also has to be mXn. This problem's sol^n is an example of this. You should use for loop inside recursive function instead of relying on 2-dimensional recursion.

CODE ⇒

```
@cache
def minCoinCount(target):
    if target == 0 : return 0
    ans = inf
    for coin in coins:
        if coin <= target:
            ans = min(ans, 1 + minCoinCount(target-coin))
    return ans
res = minCoinCount(target)
return -1 if res == inf else res
```

$O(n2^n)$ $O(2^n)$

Subset Sums

Print all possible subset sums from a given array of unique and positive numbers

APPROACH => > build curStep result from all previous steps

CODE => ans = [0] → You can always get 0 by ignoring all nums.

```
for num in arr:
    n = len(ans) # all previous steps
    for i in range(n):
        ans.append(ans[i]+num)
return ans
```

Combine previous step with next step.

$O(N*2^N)$ $O(N*2^N)$

Subsets with Duplicate

APPROACH => > Sort the array
> Use normal subset generation logic of using result from all previous steps except for the case where cur element is same as last element.
> In repeated case, only use results from previous step.

CODE =>

```
nums.sort()
ans = [[]]
prev = [[]] # this value will get overwritten and never be used as it is
for i in range(len(nums)):
    tempans = []
    if i>0 and nums[i] == nums[i-1]:
        for subset in prev:
            tempans.append(subset+[nums[i]])
    else:
        for subset in ans: # use all prev steps
            tempans.append(subset+[nums[i]])
    prev = tempans
    ans.extend(tempans)
return ans
```

O(n log(n)) O(n)

Job scheduling Problem

each job is of ~~deadline~~ type (id, deadline, profit)
return maxprofit possible if each job takes 1 slot and can only be completed on or before deadline.

APPROACH ⇒ > sort jobs from highest deadline to lowest, calculate slots between current and next highest deadline
> while slots are available, assign highest possible profit job to schedule using heap.

CODE ⇒
```
doneJobs=0, totProfit=0
jobs.sort(key=deadline)
for i in range(n-1,-1,-1): #(highest to lowest deadline)
    if i==0:
        slots = jobs[i].deadline
    else: slots = jobs[i].deadline - jobs[i-1].deadline
    heappush(maxHeap, (-jobs[i].profit))
    while slots and maxHeap:
        profit = - heappop(maxHeap)
        slots--, doneJobs++, totProfit+=profit
return [doneJobs, totProfit]
```

O(n) O(2361) or O(1)

Minimum Train platforms

given arrival and departures, find min platforms needed.

APPROACH 1 ⇒
NOTE ⇒ You can use heap approach of getting running meeting with quickest end time of MINIMUM MEETING ROOMS NEEDED PROBLEM (LC-253). But that approach is nlogn

APPROACH 2 ⇒ increment cur if train arrives and decrement if train leaves [keep arrivals and departures sorted in one single array itself] this approach also nlogn but the most optimised sol^n is specific to this problem and assumes all train data is of same day

```
trains = [0 for i in range(2361)]   # 1 extra sec for next day ka 00:00
for time in arrival:
    trains[time]++ → inc if train needed at a timestamp
for time in dep:
    trains[time+1]-- → dec if train not needed
cur, ans = trains[0], trains[0]
for i in trains[1:]
    cur += trains[i]; ans = max(ans, cur)   # max Prefix sum
return ans
```

TC → $O(2^{target} \times avg\ length\ of\ combination)$ SC → $K \times x$ (x = combination count (unpredictable))

L.C.39 Combination Sum

Each candidate has infinite supply, return unique combinations which sum to target.

APPROACH ⇒ ▶ Recursion, Unbounded Knapsack
▶ use path data structure to keep track of nums used so far.

CODE ⇒
```
N = len(candidates), ans = []

def solve(idx, target, path):
    if idx == N:
        if target == 0:
            ans.append(list(path))   # Make copy of list
    else:
        cur = candidates[idx]
        # solve without picking
        solve(idx+1, target, path)
        # try solving with picking (not always possible)
        if cur <= target:
            path.append(cur)
            solve(idx, target-cur, path)   # don't do idx+1 here as nums can be repeated
            path.pop()
solve(0, target, [])
return ans
```

TC $O(2^n \times k)$ SC $O(k \times x)$ (n = number of comb (ignoring recursive stack space); k = avg length of comb.; x = storing answers)

LC 40. Combination Sum II

Each number can be used only once. Generate all unique combinations with target sum.

APPROACH ⇒ ▶ Think like each recursive call is to pick current position's chosen number.
▶ If you're able to ensure non-duplicacy for 1 current pick, it will be ensured throughout the solution.
▶ Ignore duplicacy of the first index possible since its previous actually belongs to last step and not current step.

CODE ⇒
```
candidates.sort(), N = len(candidates), ans = []
def solve(idx, target, path):
    if target == 0: ans.append(list(path))
    else: for i in range(idx, N):
        if i > idx and candidates[i] == candidates[i-1]: continue
        if candidates[i] > target: break   # Make sure to break here to avoid TLE
        path.append(candidates[i])
        solve(i+1, target-candidates[i], path)
        path.pop()
solve(0, target, [])
return ans
```

$O(n! \times n)$ $O(n! \times n)$ (for storing answer)

Print all permutations of array with Distinct nums

APPROACH ▸ Any number can come at current index in permutation

▸ Assume each call to solve function is to determine number to be taken at current position.

CODE :-
```
ans = [], N = len(nums)
def solve(idx):
    if idx == N:
        ans.append(list(nums))
    else:
        for i in range(idx, N):
            nums[i], nums[idx] = nums[idx], nums[i]   # Swap to simulate being in current position
            solve(idx+1)
            nums[i], nums[idx] = nums[idx], nums[i]   # Swap back to clean up for next step
solve(0)
return ans
```

$O(n^2)$ loop × remove from list per iter. $O(n)$ to store answer

K^{th} permutation Lexicographically

Given all permutations of nums 1 to n, return K^{th} such permutation.

APPROACH :- If N nums left in list, and N! permutations of those nums, each number will be at beginning of (N-1)! of those permutations. So we can find which bucket K belongs to for current step and keep recalculating K and moving forward.

CODE :-
```
nums = list(range(1, n+1))
ans = []
while len(nums) > 0:
    N = len(nums)
    perHead = factorial(N-1)
    pickedIdx = (K-1) // perHead
    pickedNum = nums[pickedIdx]
    ans.append(str(pickedNum))
    nums.remove(pickedNum)
    K = K % perHead
return ''.join(ans)
```

$O(n)$ ← find index of current pick as nums is always sorted.

$O(n)$

$O(N*2^N)$ $O(N)$ for recursion

Palindrome Partitioning

Print all partitions of string s, such that all subparts are palindromes
'aab' ⇒ [['a','a','b'],['aa','b']]

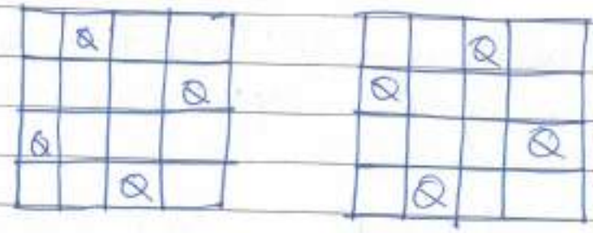
APPROACH ⇒ > recursion
> try every possible substring possible starting from current index and recur further if a palindrome is found possible.

CODE ⇒
```
def solve(idx, path):
    if idx == len(s):
        res.append(list(path))
    else:
        cur = ""
        for i in range(idx, len(s)):
            cur += s[i]
            if isPalin(cur):
                path.add(cur)
                solve(i+1, path)
                path.pop()
res = []
solve(0, [])
return res
```

$O(N!)$ $O(N^2)$ for board excluding space for answer

LC-51 N-Queens

N=4 gives



2*(n)-1 upperdiagonal (0 to 2*n-2)

APPROACH → as we are placing queens left to right, column by column, we only need to worry about conflicts from the left side as right side is empty
[left of current row, upper diagonal, lower diagonal]

- The way to get to get most optimised soln for interview is to come up with efficient checks [O(1)] for above 3. [Map row to 0 to n-1 and both diagonals b/w 0 to 2n-2

CODE ⇒
```
def solve(col, n):
    if col == n: ansBoards.append(deepcopy(board))
    else: for row in range(n):
        if leftrow[row] == empty and lowerDiag[row+col] == empty and upperDiag[n-1+col-row] == empty:
            board[row][col] = 'Q'
            leftrow[row] = full, lowerDiag[row+col] = full,
            upperDiag[n-1+col-row] = full;
            solve(col+1, n)
```

Hash for
row ⇒ row
for lowerDiag → row+col
for upperDiag → n-1+col-row

CLEANUP ⇐
```
board[row][col] = '.'
leftrow[row] = empty, lowerDiag[row+col] = empty,
upperDiag[n-1+col-row] = empty
```

$O((N!)^N)$

LC-37 Sudoku Solver

APPROACH ▸ recursion and backtracking.
▸ Try each num in each empty space and check for conflicts.

CODE ▸

```
def willConflict(row,col,num):
    for c in range(n):
        if b[row][c]==num: return True
    for r in range(n):
        if b[r][col]==num: return True
    R,C = row//3, col//3  #for 3x3 square
    for r in range(3*R, 3*R+3):
        for c in range(3*C, 3*C+3):
            if board[r][c]==num: return True
    return False
def solve(row,col):
    if col==n: return True  # filling complete
    nextRow = (row+1)%n, nextCol = col+1 if row==n-1 else col
    if board[row][col]!=empty: return solve(nextRow,nextCol)
    else: for num in range(1,n+1):
            if not willConflict(row,col,num):
                board[row][col]=str(num)
                temp= solve(nextRow,nextCol)
                if temp: return True
                board[row][col] = empty
    return False
solve(0,0)
```

$O(3^{N^2})$
n^2 cell and each has 3 unvisited cells.

$O(\frac{L}{\text{avg. length of path}} * \underset{\text{num. of paths}}{X})$

Rat in a Maze

free = 1, blocked = 0 :- Print all paths from 0,0 to n-1,n-1 which don't have a blocked cell and visit cells only once.

APPROACH :- ▸ Depth first search recursive
▸ Mark visited nodes on the way and clean up when stepping out of recursion.

```
moves = [0,-1,0,1,0]
mapping = {(0,1):R,(0,-1):L,(-1,0):U,(1,0):D}
vis = [[False for _ in range(n)] for _ in range(n)]
def dfs(i,j,path):
    if i<0 or j<0 or n-1<i or n-1<j or vis[i][j]
       or m[i][j]==blocked: return
    else: vis[i][j] = True
    if i==n-1 and j==n-1:
        ans.append(str(path))
    else:
        for di,dj in zip(moves[1:],moves[:-1]):
            I,J = i+di, j+dj
            dfs(I,J, path+mapping[(di,dj)])
    vis[i][j] = False # CLEANUP
dfs(0,0,"")
return ans.
```

$O(M^N)$ $O(N)$

M-Coloring Problem

return yes or no if all vertices of graph can be colored with $\leq M$ colors with no two neighbors having same color.

APPROACH → Don't use BFS or DFS
⇒ each ~~edge~~ vertex starting from 0 will check if [i+1, n) can be colored given its current color choice

CODE ⇒
```
color = [0 for _ in range(n)]
def isSafe(node, col):
    for dest in adj[node]:
        if color[dest] == col: return False
    return True
def canAssign(node):
    if node == n: return True
    else: for col in range(1, m+1):
        if isSafe(node, col):
            color[node] = col
            if canAssign(node+1):
                return True
            color[node] = 0
    return False
return 1 if canAssign(0) else 0
```

$O(\log n)$ $O(1)$

Single Non-Repeat Element in Sorted Array

1, 1, 2, 3, 3, 4, 4, 8, 8 ⇒ Answer should be 2

APPROACH → Binary Search
Couple of observations → ① If you are standing before the target element, all first occurences of nums would be at even indices and 2nd occurences would be at odd indices.
② Ily first occurences are at odd indices and second occurences are at even indices after the target element.

CODE ⇒
```
l, r = 0, n-1
while l<=r:
    mid = l+(r-l)//2
    cur = nums[mid]
    if mid>0 and nums[mid-1]==cur:
        # means this is second occurence
        if mid%2==0: # means currently we are at after target
            r=mid-1
        else: l=mid+1
    elif mid<len(nums)-1 and nums[mid+1]==cur:
        # first occurence.
        if mid%2==1: r=mid-1 # first occurence at odd means after target
        else: l=mid+1
    else: return cur # target element
```

$O((\log n) \times \log(m \times 10^p))$ $O(1)$

for pow operations → precision digits

N^{th} Root of a Number Using Binary Search

Find $(M)^{1/N}$

APPROACH ⇒ we can keep a search space of ans for 1 to M, and keep an eps (diff value) of 10^{-7} (because answer will only be checked for 6 digits)

CODE
```
def findNthRootOfM(n, m):
    eps = 10**(-7)
    l, r = 1, m

    while r-l > eps:
        mid = l + (r-l)/2
        v = mid**n  # O(logn)
        if v > m:
            r = mid
        else
            l = mid

    return "%.6f" % l
```

Note :- total search space is
1.000000 to m
so $(m \times 10^6)$ total possible nums
so binary search will have $\log(m10^6)$ steps

$O(N * 2^N)$ $O(N * 2^M)$

Word Break Problem

given a dictionary of words and a string of chars, put spaces in possible places such that valid words from dictionary are obtained. Return all such valid sentences

APPROACH ⇒ ▶ Recursion
▶ Very similar to palindrome partitioning

CODE :-
```
words = set(dictionary), ans = []
def solve(idx, path):
    if idx == len(s):
        ans.append(" ".join(path))
    else:
        cur = ""
        for i in range(idx, len(s)):
            cur += s[i]
            if cur in words:
                path.append(cur)
                solve(i+1, path)
                path.pop()

solve(0,[])
return ans.
```

TC→ $O(\log(10^{**}9) * (n\log(m)))$

$n \times m_{matrix}$ → S.C → O(1)

Matrix Median with sorted rows with odd total elements

APPROACH → definition of median → even counting duplicates (even of median itself), matrix would total have >= (tot+1)//2 elements which are less than or equal to it.

- we should try to find the smallest num val which satisfies this property.
- In the end I will always have the correct value.

CODE :-

```
tot = len(A) * len(A[0])
lessThanEqlCnt = (tot+1)//2
l, r = 1, 10**9

while l<=r:
    mid = l+(r-l)//2
    v = getCount(mid)
    if v >= lessThanEqlCnt:
        r = mid-1
    else:
        l = mid+1
return l
```

```
def getCount(num):
    ans = 0
    for row in A:
        ans += bisect_right(row, num)
    return ans
```

classmate

O(logN) O(1)

L.C. 33 Search in a Rotated Sorted Array

[4, 5, 6, 7, 0, 1, 2] , target = 0

↑ Ans → 4

Approach > Binary Search

- No matter where your mid lies, either the left subpart or the right subpart WILL BE sorted.
- The approach relies on identifying IF ITS POSSIBLE for the target to be in sorted subpart or not.

CODE :-

```
l, r = 0, N-1
while l<=r:
    m = l+(r-l)//2
    if nums[m] == target:
        return m
    elif nums[l] <= nums[m]:   # sorted subpart l to m
        if nums[l] <= target <= nums[m]:
            r = m-1
        else:
            l = m+1   will
    else:   # right part be sorted
        if nums[m] <= target <= nums[r]:
            l = m+1
        else:
            r = m-1
return -1
```

classmate

O(log(min(m,n))) ~~O(1)~~

LC.4 Median of two sorted Arrays in log(min(m,n)) complexity

APPROACH :- ① Basically we do binary search to decide number of elements picked from arr1, from this we can also calculate number of elements picked in arr2 for that iteration.
② Then check if this is a valid split by comparing CROSS ELEMENTS.
③ In case of odd length combined, only consid the MAXIMUM ELEMENT of LEFT SUBPART.

CODE ⇒
```
m,n = len(nums1),len(nums2)
if m>n: swap nums1 and nums2 and m,n
tot = m+n, onLeft = (m+n+1)//2, l=0, r=m
while l<=r:
    takenFromFirst = l+(r-l)//2
    takenFromSecond = onLeft - takenFromFirst
    if takenFromSecond > n: l = takenFromFirst+1
                            continue
    elif takenFromSecond<0: r = takenFromFirst-1
                            continue
    l1 = -inf if takenFromFirst==0 else nums[takenFromFirst-1]
    l2 = -inf if tfs==0 else nums2[tfs-1]
    r1 = inf if tff==m else nums1[tff], r2 = inf if tfs==n else nums2[tfs]
    if l1<=r2 and l2<=r1:
        return max(l1,l2) if tot%2==1 else (max(l1,l2),min(r1,r2))/2
    else: if l1>r2: r=tff-1 else l=tff+1
```

(check if cross elements are valid as $l_1 <= r_2$ and $l_2 <= r_1$ and automatically)

O(log(min(m,n))) O(1)

Kth element of 2 sorted arrays

APPROACH → ▸ same as previous problem, just try keep ~~first~~ first K elements in left subpart.
▸ Also swap arrays if first one is bigger by default
By default → arr1 → n elements arr2 → m elements

CODE :-
```
if n>m:
    n,m = m,n
    arr1,arr2 = arr2,arr1
l,r = 0,n
while l<=r
    pickedFrom1 = (l+r)//2
    pickedFrom2 = K - pickedFrom1
    if pf2>m: l = pf1+1; continue;
    if pf2<0: r=pf1-1; continue.
    l1 = -inf if pf1==0 else arr1[pf1-1]
    l2 = -inf if pf2==0 else arr2[pf2-1]
    r1 = inf if pf1==n else arr1[pf1]
    r2 = inf if pf2==m else arr2[pf2]
    if l1<=r2 and l2<=r1:
        # valid possibility
        return max(l1, l2)
    else:
        if l1>r2: r=pf1-1
        else: l=pf1+1
```

O(nlogn) O(1)

Aggressive Cows

Given possible stall locations and number of cows, return max possible min distance between cows.

APPROACH → ▶ binary search, try to get max possible answer

CODE ⇒
```
l, r = 0, stalls[-1]   # possible distance space.
while l<=r:
    mid = (l+r)//2
    if isPos(mid):
        ans = max(ans, mid)
        l = mid+1
    else:
        r = mid-1
return ans

def isPos(dist):
    allotted = 0, prev = -inf
    for stall in stalls:
        if stall - prev >= dist:
            allotted++, prev = stall
            if alloted >= cows: break
    return alloted >= cows
```


O(nlogn) O(1)

Allocated Books

Return min. possible max book allocation such that books are allocated continuously and each student gets atleast one book.

APPROACH → ▶ Binary Search

▶ CODE ⇒
```
if len(books) < students:
    return -1
else: l, r = 0, sum(books)
    ans = inf
    while l<=r:
        mid = (l+r)//2
        if isPos(mid):
            ans = min(ans, mid)
            r = mid-1
        else: l = mid+1
    return ans
def isPos(maxi):
    prev, assigned = maxi, 0
    for book in books:
        if prev+book > maxi:
            prev = book, assigned++
            if prev > maxi: return False
        else: prev += book
    return assigned <= students
```

O(n log K) O(K)

Kth Largest Element in an Array

APPROACH :- heap (minimum) [Largest finding needs min heap]

- Keeping adding to heap, as soon as number of elements becomes >K, kick one out
- In the end return the topmost element.

CODE ▸
```
heap = []
for num in nums:
    heappush(heap, num)
    if len(heap) > K: heappop(heap)
return heappop(heap)
```


O(NlogN) O(N)

Maximum Sum Combination

Given two arrays, comb-sum takes one elem from either array and returns top C of them

APPROACH ↴

- Sort the arrays individually first
- Then there is no predictable way to tell next best possible combo after first, so just push all possibilities to the heap.
- Also keep track of visited set to prevent duplicates.

CODE :-
```
ans, h, seen = [], [], set()
A.sort(), B.sort()
heappush(h, (-(A[N-1]+B[N-1]), N-1, N-1))
for _ in range(C):
    negTot, i1, i2 = heappop(h)
    ans.append(-negTot)
    if i1-1 >= 0 and (i1-1, i2) not in seen:
        seen.add((i1-1, i2))
        heappush(h, (-(A[i1-1]+B[i2]), i1-1, i2))
    if i2-1 >= 0 and (i1, i2-1) not in seen:
        seen.add((i1, i2-1))
        heappush(h, (-(A[i1]+B[i2-1]), i1, i2-1))
return ans
```

Find median -> O(logN) space -> O(N)

Find Median from Data Stream

APPROACH ▶ Use two heaps, minimum elements would be stored in maxHeap and maximum elements in minHeap.
▶ Also keep elements balanced, maxHeap should have one element extra, in case of odd total elements. Also use this while calculating median, taking elements from both heaps for even total length.

CODE :-
```
def addNum(num):
    heappush(maxHeap, -num)
    heappush(minHeap, -heappop(maxHeap))
    if len(minHeap) > len(maxHeap):
        heappush(maxHeap, -heappop(minHeap))
def findMedian():
    if len(maxHeap) > len(minHeap):
        return -maxHeap[0]
    else:
        return (-maxHeap[0] + minHeap[0])/2
```


O(nlogK) O(n)

Top K Frequent Elements

APPROACH -> ▶ Make a dictionary of num to freq
▶ Keep highest freqs (k of them) in heap, and kick out when this count goes > K.

CODE :-
```
ctr = Counter(nums)
minH = []
for key in ctr:
    heappush(minH, (ctr[key], key))
    if len(minH) > k:
        heappop(minH)

return [x[1] for x in minH]
```

N → total elements in all arrays O(n log K) O(n)
K → Num of arrays

Merge K sorted Arrays

APPROACH → Keep a pointer for each index of each array, and push the next index of an array once current one is processed and pushed to answer.

CODE ▶
```
minH = [], ans = []
for i in range(K):
    heappush(minH, (KArrays[i][0], i, 0))
while minH:
    val, arrIdx, idx = heappop(minH)
    arrLen = len(KArrays[arrIdx])
    ans.append(val)
    if idx+1 < arrLen:
        heappush(minH, (KArrays[arrIdx][idx+1],
                        arrIdx, idx+1))
return ans.
```


O(n) O(n)

Check Valid Parenthesis

APPROACH ⇒ ▶ use stack for matching
▶ Return False whenever any mismatch occurs.

CODE ▶
```
st = deque() # to simulate stack
pair = {')':'(', ']':'[', '}':'{'}

for c in s:
    if c in '[({':
        st.append(c)
    else:
        if len(st)==0 or st[-1] != pair[c]:
            return False
        st.pop()
return len(st)==0 # Final check.
```

O(n) O(n)

LC.496 Next Greater Element - I

APPROACH → monotonous stack.

- Given nums array and query array, store the result for each unique num in map as they are popped out of stack.
- The answer for each num is that number which 'pops it out' from monotonic stack.

CODE:-

```
grtr = defaultdict(lambda: -1)
st = []
for num in nums:
    if len(st)==0 or st[-1]>num:
        st.append(num)
    else:
        while st and st[-1]<num:
            grtr[st.pop()] = num
        st.append(num)
return [grtr[num] for num in qry]
```


O(1) amortised all ops O(n)

L.C. 232 Implement Queue Using Stacks

APPROACH →
- Use two stacks
- Keep all operations amortised O(1)
- we do push operation fullfilment from st1 and pop operation fullfilment from st2.

CODE:-

```
def __init__(self): self.st1, self.st2 = [], []
def push(self, x):
    self.st1.append(x)
def empty(self):
    return not (self.st1 or self.st2)
def pop(self):
    if not self.st2:
        while self.st1:
            self.st2.append(self.st1.pop())
    return self.st2.pop()
def peek(self):
    if not self.st2:
        while self.st1:
            self.st2.append(self.st1.pop())
    return self.st2[-1]
```

O(1) for push O(n) for pop and peek O(n) space

Implement Stack Using Queues
(Using Only 1 queue)

APPROACH ▶ append or push just like normal
▶ But while popping or peeking, we basically wrap around the whole queue and return or pop the last element.

CODE :-
```
def push(x): self.q.append(x)
    def size(self): return len(self.q)
    def empty(): return self.size()==0
    def pop():
        n = self.size()
        for _ in range(n-1): # repeat all except last
            self.q.append(self.q.popleft())
        return self.q.popleft()
def top():
    n = self.size()
    for _ in range(n-1):
        self.q.append(self.q.popleft())
    temp = self.q.popleft()
    self.q.append(temp)
    return temp #  ↳ # send into queue
```

O(n²) O(n)

Sort Stack

APPROACH
▶ We need 2 recursive function, one sorts and other appropriately inserts inside a presorted stack.
▶ We remove current element and sort rest of stack.
▶ And then assuming stack has been sorted, we go and insert current value in correct place.

CODE :-
```
def sortStack(stack):
    if stack:
        cur = stack.pop()
        sortStack(stack)
        insertInStack(stack, cur)

def insertInstack(stack, elem):
    if len(stack)==0 or stack[-1]<=elem:
        stack.append(elem)
        return
    else:
        temp = stack.pop()
        insertInStack(stack, elem)
        stack.append(temp)
```

$O(n * \log(\log(n)))$ $O(n)$

Sieve of Eratosthenes

Returns the prime numbers in a given range.

```
# initialize Array                    Each index will tell if given index num is ↓ prime
prime = [True for _ in range(n+1)]

# start from 2
p = 2
while p*p <= n:
    if prime[p]: # Only try for prime nos.
        for i in range(p*p, n+1, p):   ↳ skip p steps every time
            prime[i] = False
    p += 1

# Finally prime assay will have status of all nums [prime or not]
```

$O(n)$ $O(n)$

LC 901 Online Stock Span

Number of days in a row where stock price was <= current day price including today.

```
def __init__(self):
    self.st = deque()
    self.cnt = 1

def next(self, price):
    while self.st and self.st[-1][0] <= price:
        self.st.pop()

    prevHigher = self.st[-1][1] if self.st else 0
    ans = self.cnt - prevHigher
    self.st.append((price, self.cnt))
    self.cnt += 1
    return ans
```

APPROACH ⇒ Keep popping elements which have lesser or equal price, and store day also in stack.

answer is given by currentDay - prev day with higher value.

▸ Remember to push current value in end and update count.

O(n) O(n)

Prev Smaller

- Monotic Stack, Keep Kicking greater and equal elements, store current elem when you're done.

CODE :-

```
st, n = [], len(ars)
ans = [-1]*n
for idx, num in enumerate(ars):
    while st and st[-1] >= num:
        st.pop()
    if st:
        ans[idx] = st[-1]
    st.append(num)
return ans
```


O(1) O(n)

Min Stack

Develop special stack that supports getting minimum element in O(1)

```
def __init__(self):
    self.st = []

def push(self, val)
    self.st.append((val, min(val, self.st[-1][1]) if self.st else val))

def pop(self):
    return self.st.pop()[0]

def getMin(self):
    return self.st[-1][1]
```

O(m·n) O(m·n)

LC994 Rotting Oranges

Find minimum time to spread rotting to all fresh oranges using bfs.

APPROACH
- Push all rotten oranges position in queue.
- From those keep spreading and increment time when one iteration of spreading is over for all rotten oranges.

```
# First mark totalOranges, and rottenOranges
                    ↙ queue
while infected and totInfected != totOranges:
    N = len(infected)
    for _ in range(N):
        i,j = infected.popleft()
        for di,dj in moves:
            I,J = i+di, j+dj
            if 0<=I<m and 0<=J<n
                    and grid[I][J]==fresh:
                grid[I][J] = rotten
                totInfected+=1
                infected.append((I,J))
    time+=1
return time if totInfected!=totOranges else -1
```

O(3N) O(N)

LC.84 Largest Rectangle in Histogram

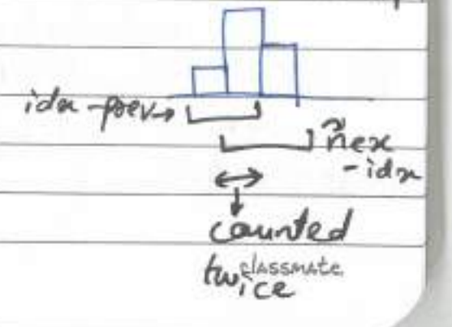
- Keep track of next smaller and prev smaller.

NOTE → always keep out of index range value as default in case AND THEN subtract any double countings !!
For example -1 for prevSmaller and n for nextSmaller while range is 0 to n-1

CODE→

```
# use stack to get nextSmaller &
# prevSmaller with defaults n & -1 resp.
# then-
ans = 0
for idx in range(n):
    cur = heights[idx]
    nex,prev = nextSmaller[idx], prevSmaller[idx]
    L,R = (idx-prev)*cur, (nex-idx)*cur
    ans = max(ans, L+R-cur)
return ans
```

remove overlap



O(N) O(1)

LC-277 Find the Celebrity

A celebrity knows No-one but all know him/her. Tell if celebrity exists or not in <=3N calls to knows(i,j) [returns bool]

APPROACH ▸ Start with a candidate and every call will rule out 1 candidate (by either establishing new candidate and rejecting old or keeping current candidate and reject current other person being checked)

- Final confirmation check is needed however
- Totally makes around 3N calls. [2 celeb]

CODE:

```
candidate=0
for i in range(1,N):
    if knows(i,candidate):
        pass
    else:
        candidate = i
if isCeleb(candidate):
    return candidate
else: return -1
```

```
def isCeleb(i):
    for j in range(n):
        if knows(i,j) or
        not knows(j,i):
            return False
    return True
```

classmate

O(n) O(n)

Reverse Words in a String

"..hello...world.." ⇒ "world_hello"
(Also remove extra spaces)

Straightforward Code using inbuilt

```
return ' '.join(s.strip().split()[::-1])
```

A note on reducing space complexity
The below steps won't remove extra space but will solve rest of problem

① Reverse whole string "dlrow olleh"
② Reverse word by word "world hello" → Use sliding window concept

This makes space complexity O(1) possible

classmate

Since roman numerals are finite (3999) ← [O(1) O(1)

Roman to Integer

APPROACH → if a character with smaller value occurs before a character with higher value, the subtract the smaller else add value

CODE →
```
st = deque()
valMap = {I:1, V:5, X:10, L:50, C:100,
          D:500, M:1000}
ans = 0
for i in s:
    if not st or st[-1] >= valMap[i]:
        pass
    else:
        ans -= st.pop()
    st.append(valMap[i])
ans += st.sum() #sum of unpopped elements.
return ans
```

O(n²) O(1)

Longest Palindromic Substring

APPROACH ⇒ If you don't use dp, you can do in constant space
- Assume each index as starting point and assume both odd or even length.
- save the longest palindrome encountered so far.

CODE ⇒
```
ans = s[0]
for startingPoint in range(n):
    oddLen = equalsCount(startingPoint, startingPoint)
    evenLen = equalsCount(startingPoint, startingPoint+1)
    if 2*oddLen-1 > len(ans):
        ans = s[startingPoint-oddLen+1 : startingPoint+oddLen]
    if 2*evenLen > len(ans):
        ans = s[startingPoint-evenLen+1 : startingPoint+evenLen+1]
return ans
def equalsCount(l,r):
    diff = 0
    while l-diff>=0 and r+diff<len(s) and s[l-diff]==s[r+diff]:
        diff += 1
    return diff
```

O(n) O(n)

Sliding window Maximum

Find maximum value of every window of size K.

▶ Approach ⇒ use deque(), remove elements that go out of sliding window scope or have value smaller or equal to current idx value, this ensures max value for each window is on left front of queue.

CODE ⇒

```
ans = deque(), [0]
for idx, num in enumerate(nums):
    while q and q[0][0] == idx-K:
        q.popleft()
    while q and q[-1][1] <= num:
        q.pop()
    q.append((idx, num))
    if idx >= k-1:
        ans.append(q[0][1])
return ans.
```

classmate

O(n) O(n)

Maximum of minimum of every Window Size

APPROACH ⇒ Naive approach would be $O(n^3)$ time and O(1) space.

▶ Optimised approach requires finding nextSmaller and prevSmaller.

▶ Other imp. facts → ① you are a potential minimum for the window size of (yournextsmaller - yourprevsmaller - 1)

② For the unfilled slots from previous step, answers are not directly available, so use ans[size] = max(ans[size], ans[size+1])

∴ Answer of smaller window size ≥ Ans of a larger window size

CODE →

```
tempans = [-inf for _ in range(n+1)]
for i in range(n):
    windowLen = nextSmaller[i] - prevSmaller[i] - 1
    tempans[windowLen] = max(tempans[windowLen], a[i])
                                                  curr elem
for i in range(n-1, 0, -1):
    tempans[i] = max(tempans[i], tempans[i+1])
return tempans[1:]  # final answer.
```

classmate

total number of chars

O(S) O(1)

LC-14 Longest Common Prefix

- given a set of strings return the longest common prefix among them all.
- Use vertical scanning and check if those chars match.

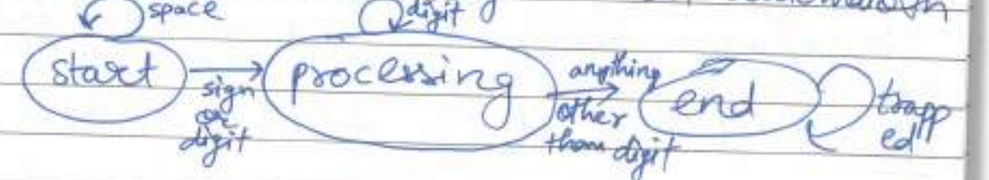
CODE ⇒

```
N, idx = len(strs), 0
while True:
    char = None
    for i in range(N):
        if len(strs[i]) <= idx:
            return strs[i][:idx]
        else:
            if char == None:
                char = strs[i][idx]
            else:
                if char != strs[i][idx]:
                    return strs[i][:idx]
    idx += 1
```

O(L) O(1)

LC-8 Automaton based atoi

Approach
- The logic can get quite complex if written through if statements and loops alone.
- So it makes more sense to progress character by character and encode logic in an automaton



- To store number, keep a val and multiplier variable each to store magnitude and sign
- Keep clamping the value to 2^31 to prevent overflow!
- Before returning results, make sure to account that there is -2**31 in 32-bit integer but not 2**31

O(m+n) O(1)

LC·242 ~~Correct~~ Valid Anagrams

- Two anagrams are equal if the number of each alphabet in both is same.
- We could use sorting also but that increases time complexity

CODE ⇒ def check(s,t)
    return Counter(s) == Counter(t)

Not easy to predict complexity, varies based on N

LC·38 Count And Say

countAndSay(n) = 1 if n==1 else countAndSay(n-1)
↓ and we process like this → '3322251' ⇒ '2332 15 11'
count+num

APPROACH ⇒ Sliding window, Keep growing till char same as start, then append to ans and reset window with cnt = 0

CODE →
```
def countAndSay(n):
    if n==1: return '1'
    base = countAndSay(n-1)
    cur, r, cnt, N, ans = None, 0, 0, len(base), ''
    while r<N:
        cur = base[r]
        while r<N and base[r]==cur:
            r+=1
            cnt+=1
        ans += str(cnt)+cur
        cnt = 0
    return ans
```

$O(max(M,N))$ $O(1)$

LC-165 Compare Version Numbers

APPROACH

- Sliding window goes over both strings and resets on finding a period symbol.

```
i, j, m, n = 0, 0, len(v1), len(v2)

while i<m or j<n:
    val1, val2 = 0, 0

    while i<m and v1[i] != '.':
        val1 *= 10
        val1 += int(v1[i])
        i += 1
    i += 1 # move ahead of decimal
    while j<n and v2[j] != '.':
        val2 *= 10
        val2 += int(v2[j])
        j += 1
    j += 1

    if val1 < val2: return -1
    elif val1 > val2: return 1

return 0
```


$O(n)$ $O(n)$

Minimum Characters Required to make String Palindrom

- Modification of Knuth Morris Patt (Longest Proper prefix which is also suffix)
- The answer comes from subtracting last value in lps array from length of string
- we feed A+#+A[::-1] into Kmp.

AACECAAAA $\xrightarrow{Kmp}$ AACECAAAA#AAAACECAA

base input

these 2 characters will need to be added (9-7)

val of lps here will be 7

CODE → `return len(A) - kmp(A+'#'+rev(A))[-1]`

O(n) O(n)

Path to Given Node in Tree

Return path from root to node, target will always be present

APPROACH :-
- Do a VLR traversal (pre-order)
- Append your current node to path.
- If current node is target return path else recurse further.

```
def vlr(root, target, path):
    if not root: return False
    path.append(root.val)
    if root.val == target: return path
    l, r = vlr(root.left, target, path),
           vlr(root.right, target, path)
    if l != False:
        return l
    elif r != False:
        return r
    path.pop()
    return False

return vlr(root, target, [])
```


O(1) O(capacity)

LRU cache

- Involves using a hashmap and doubly linked list.
- DLLNode {
  int key;
  int val;
  DLLNode prev;
  DLLNode next;
  }
  oldest node near head
  newest near tail

DLL Class
- Keep head and tail nodes and store list between them.
- Maintain a size variable or use hashmap size to know when to popleft.

Functions to use in Doubly linked list?
- moveToBack(Node) ⇒ clear and reset pointers of prev and next node then call append(node)
- append ▸ modify tail.prev and tail node and the node itself.
- popfront → reset head and head.next pointers; also remove key from hashmap afterwards

O(N) O(1) [if not counting ans array]

Morris Traversal [IN-ORDER] (And preorder)

APPROACH ▶ Uses the space in right pointers of leaf nodes to traverse and not use any stack or queue
▶ Since its in-order traversal:- ① if there is no temp.left, insert value and move right
② Now if left subtree exists, find its right most node or detect if cycle is present in tree
(i) If cycle is present, push current and reset pointer and move to right subtree.
(ii) Else set left.right to node and move to left subtree for traversal

CODE :-

```
temp = root, ans = []
while temp:
    if not temp.left:
        ans.append(temp.val)
        temp = temp.right
    else:
        rightMostLeaf = temp.left
        while rightMostLeaf.right and
              rightMostLeaf.right != temp:
            rightMostLeaf = rightMostLeaf.right
        if rightMostLeaf.right == temp:
            ans.append(temp.val)
            rightMostLeaf.right = None, temp = temp.right
        else:
            rightMostLeaf.right = temp
            temp = temp.left
return ans
```

In-order case we append to ans when traversal of left side is over i.e. when rightMost leaf ka right matches temp, in pre-order case append to ans before left traversal starts

add ans.append line to other side of if statement and you get pre-order.

O(m×n) O(mn)

LC 733. Flood Fill

Starting from source cell, fill all nearby cells with newColor (4-directionally adjacent).

APPROACH :- ▶ Use DFS,

CODE

```
m, n = len(image), len(image[0])
baseColor = image[sr][sc]
if baseColor != newColor:
    def dfs(r, c):
        if 0<=r<m and 0<=c<n and image[r][c] == baseColor:
            image[r][c] = newColor
            for dr, dc in moves
                R, C = r+dr, c+dc
                dfs(R, C)
    dfs(sr, sc)
return image
```

O(nlogK) O(K)

LC·215 Kth Largest Element In an Array

APPROACH → Two common approaches :-
1) QuickSelect, 2) Heap (minHeap)
constant space but n^2 worst case time so don't use.

HEAP APPROACH CODE

```
heap = [ ]
for num in nums:
    heappush (heap, num)
    if len (heap) > K:
        heappop(heap)
return heap[0]
```

Similar Question → LC·703
Kth Largest element in a stream.
Only difference is elements are inserted dynamically.

O(nlogn) O(n)

LC·300 Length of Longest increasing Subsequence

APPROACH ▸ Keep a res array, it won't contain the l.i.s. but its length will be same as L.I.S.
▸ If num in nums is largest element so far, then append it to end or res.
▸ otherwise it will just replace another older element.

REASONING → res represent tails where tails[i] (for res[i]) is the smallest tail (end element) of all possible increasing subsequences of i+1 length.

For example nums = [4,5,6,3] ⇒ tails = [3,5,6]
tails (or res) is clearly an increasing array, and we can do binary search to pick which index to update.

[4] [5] [6] [3] len 1
[4,5] [5,6] len 2
[4,5,6] len 3

CODE :-

```
tails = [ ]
for num in nums:
    temp = bisect_left (tails, num)
    if temp == len (tails): tails.append (num)
    else:
        tails[temp] = num
return len (tails)
```

O(n) O(n)

Left View (or right view) of Binary Tree

CODE :-
```
ans = [ ]
def lvr(root, depth = 0)
    if not root : return
    if depth == len(ans):
        ans.append(root.val)
    lvr(root.left, depth+1)
    lvr(root.right, depth+1)

lvr(root, 0)
return ans
```

O(n) t.c. possible [check bottom] O(nlogn) O(n)
(similar to level order traversal and Zig-Zag level order)

Bottom View of a Binary Tree

APPROACH :-
- In case of overlap, we need to consider the node occuring later in a level order traversal
- This gives hint to use LVL ORDER TRAVERSAL.
- Maintain a delta attribute which starts at 0 at root and then increase or decreases on moving right or left. This will be used to calculate Bottom view. Use map to store value against each delta

CODE :-
```
bottomView = { }, q = deque()
q.append((root, 0))
while q:
    lenQ = len(q)
    for _ in range(lenQ):
        node, delta = q.popleft()
        bottomView[delta] = node.val
        if node.left:
            q.append((node.left, delta-1))
        if node.right:
            q.append((node.right, delta+1))

Keys = sorted(list(bottomView.keys()))
ans = [bottomView[key] for key in Keys]
return ans
```

To get O(n) → you need to find how left the leftmost node goes and then add that to every delta in (how right the rightmost goes) traversal, this removes need of map, can use array

O(nlogn) O(n)

Top View of Binary Tree

- Just like bottom view, this also needs level order traversal and hashmap.
- No need to overwrite once a key, value is set since top elements are already first due to Level order traversal.

CODE:-

```
q = deque(), tv = {}
q.append((root, 0))
while q:
    lenQ = len(q)
    for _ in range(lenQ):
        node, delta = q.popleft()
        if delta not in tv:
            tv[delta] = node.data
        if node.left:
            q.append((node.left, delta-1))
        if node.right:
            q.append((node.right, delta+1))
keys = sorted(list(tv.keys()))
ans = [tv[key] for key in keys]
return ans
```


O(n) O(n)

LC·104 Depth of Binary Tree

- Approach → Use recursion
  - add one for each non null node and take max depth of child subtrees recursively

```
def maxDepth(root)
    if not root: return 0
    return 1 + max(maxDepth(root.left),
                   maxDepth(root.right))
```

O(n) O(n)

L.C.110 Balanced Binary Tree

APPROACH :- This is similar to previous question of calculating height/depth, except with a few more conditions.

- Check if condition is true for current subtree and also check if it holds for child subtrees. Only say true if holds for all three times.
- Recursive Basecase: always balanced for empty tree.

CODE :-

```
def lvr(root):
    if not root: return (0, True)
    l = lvr(root.left)
    r = lvr(root.right)
    isBal = l[1] and r[1] and abs(l[0]-r[0]) <= 1
    return (1+max(l[0],r[0]), isBal)

return lvr(root)[1]
```


O(n) O(n)

LC.100 Same Trees

- Base Case: If one root is null, other should be null as well.
- Otherwise current value should be same and recursively same Tree property should hold for corresponding children.

CODE :-

```
def isSame(p, q):
    if not p or not q:
        return p == q
    else:
        return p.val == q.val and
            self.isSame(p.left, q.left) and
            self.isSame(p.right, q.right)
```

O(nlogn) O(n)

LC.987 Vertical Order traversal in Binary Tree

▶ If row and col are same for a node, output in sorted manner for those nodes.

APPROACH:- ① Use a sorted map, (col, row) → sorted list of values. (basically multiset)
② Now do any traversal and append nodes to the correct sorted set.
③ Then just unpack the values from sorted set into answer.

CODE:-

```
def verticalTraversal(root):
    colMap = SortedDict()
    def vlr(node, row=0, col=0):
        if not node: return
        if not col in colMap:
            colMap[col] = SortedDict()
        if row not in colMap[col]:
            colMap[col][row] = SortedList()
        colMap[col][row].add(node.val)
        vlr(node.left, row+1, col-1)
        vlr(node.right, row+1, col+1)
    vlr(root)
    ans = []
    for col in colMap:
        temp = []
        for row in colMap[col]:
            temp.extend(colMap[col][row])
        ans.append(temp)
    return ans
```

classmate

O(n) O(n)

LC.662 Maximum Width of Binary Tree

Basically the maximum distance between two nodes on same level of tree.

▶ Since dist. is to be found between nodes of same level, level order traversal
▶ To maintain distance calculation, children should be assigned indices of 2*idx, and 2*idx+1.
▶ But since exponential doubling can quickly exhaust 32-bit integer space and since only relative index on current level matters, we can set starting node's index to 0 and subtract starting index value from other same level nodes as relative dist. remain same

CODE:-

```
ans, q = 0, deque()
q.append((root, 1))
while q:
    ans = max(ans, q[-1][1] - q[0][1] + 1)
    lenQ, begin = len(q), q[0][1]
    for i in range(lenQ):
        cur, idx = q.popleft()
        idx -= begin
        if cur.left:
            q.append((cur.left, idx*2))
        if cur.right:
            q.append((cur.right, idx*2+1))
return ans
```

classmate

O(n) O(n)

L.C. 543 Diameter of Binary Tree

APPROACH :- ① Calculate the ans considering diameter passes through current node, but pass the recursive return value as if it does not ~~pass through~~ end at current subtree.
② So answer is given ultimately by answer variable and not by return value of recursion.

```
ans = 0
def solve(node):
    nonlocal ans
    if not node: return 0
    L, R = solve(node.left), solve(node.right)
    ans = max(ans, 1+L+R)
    return 1+max(L,R)

solve(root)
return ans
```


O(n) O(n)

Preorder, In-order, Post Order in One traversal

- Just need to be smart about when exactly you do recursive call on subtree.

CODE :-

```
preorder, inorder, postorder = [],[],[]
    def solve(root):
        if not root: return
        preorder.append(root.val)
        solve(root.left)
        inorder.append(root.val)
        solve(root.right)
        postorder.append(root.val)
solve(root)
```

O(n) O(n)

Distinct Numbers in Window

- Keep adding numbers to window and maintain a dictionary to keep track of distinct values
- Pop values from left, when your window size would be increased beyond limit.

CODE :-
```
n, ans, vis = len(A), [], {}

for r in range(n):
    cur = A[r]
    if cur not in vis:
        vis[cur]=1
    else: vis[cur]+=1

    if r>=B-1:#time to add to ans and pop from left
        ans.append(len(vis))
        endIdx = r-B+1
        endVal = A[endIdx]
        vis[endVal]-=1
        if vis[endVal]==0:
            del vis[endVal]

return ans
```


T.C. O(n) SC: Recursion O(n) iterative O(1)

LC-114 Flatten Binary Tree

Recursion approach

- Keep a track of prev and do normal pre-order traversal.

CODE :-
```
prev = None
def vlr(node):
    if not node: return
    L,R = node.left, node.right
    if prev: prev.right = node
    prev = node
    node.left = None
    vlr(L), vlr(R)
vlr(root)
```

- Iterative Approach (Morris Traversal Modification)

```
temp = root
while temp:
    if temp.left:
        rightMost = temp.left
        while rightMost.right:
            rightMost = rightMost.right
        rightMost.right = temp.right
        temp.right = temp.left
        temp.left = None
    temp = temp.right
```

O(n) O(n)

Lowest Common Ancestor in B.T.

APPROACH → we need a recursive find function that returns the node p or q if any is found but if both nodes are found in different child subtrees, it returns the parent ~~root~~ itself.

CODE ⇒

```
def find(root,p,q):
    if not root or root==p or root==q:
        return root
    L = find(root.left,p,q)
    R = find(root.right,p,q)
    if L and R:
        return root      # both present in different subtrees
    return L or R
```


O(n) O(n)

Boundary of Binary Tree

(root, then left boundary, then leaves, then right Boundary reversed)

Left Boundary → root.left and its children, prefer left child

Right Boundary → root.right and its children, prefer right child

ans = [root.val]

CODE ⇒

```
if isLeaf(root): return ans   # otherwise root is also leaf and will be double counted
else:
    ans.extend(leftBound(root.left))
    leaves = []
    getLeaves(root,leaves)
    ans.extend(leaves)
    ans.extend(rightBound(root.right)[::-1])
    return ans
```

```
def leftBound(node):
    res = []
    while node and not isLeaf(node):
        res.append(node.val)
        node = node.left or node.right
    return res
```

```
def rightBound(node):
    res = []
    while node and not isLeaf(node):
        res.append(node.val)
        node = node.right or node.left
    return res
```

```
def getLeaves(node,leaves):
    if not node: return
    if isLeaf(node): leaves.append(node.val)
    getLeaves(node.left,leaves)
    getLeaves(node.right,leaves)
```

```
def isLeaf(node):
    return node and not(node.left or node.right)
```

O(n) O(n)

Similar → from Postorder and Inorder

Construct Binary Tree from Preorder and Inorder

APPROACH :- ① First put val to idx mapping for all elements in inorder for quick retrieval.
② since its preorder, we know the root will be the first element. Similarly root of each subtree comes before its subtree. But just using pre-order, we can't tell where each subtree ends, this we can find using inorder.
③ But we need to calculate starting point of right subtree in preorder using inorder [By accounting for left subtree size]

CODE :-

```
n = len(inorder), inorderMap = {x:i for i,x in enum(inorder)}
def solve(idx, l, r):
          # idx for preorder, l, r for inorder range
    if idx < n and l <= r:
        cur = preorder[idx]
        node = TreeNode(cur)  # Generate current root node
        inOrderIdx = inorderMap[cur]  # find in inorder
```

Now l to inOrderIdx-1 is left subtree &
inOrderIdx+1 to r is right subtree

> but for further calls we need new idx value in preorder, for left subtree it can be idx+1
but for right subtree it needs to be calculated

```
        leftSubtreeSize = inorderIdx - l
        node.left = solve(idx+1, l, inorderIdx-1)
        node.right = solve(idx+1+leftSubtreeSize,
                           inorderIdx+1, r)
        return node
return solve(0, 0, n-1)
```

O(n) O(n)

LC.101 Symmetric Tree

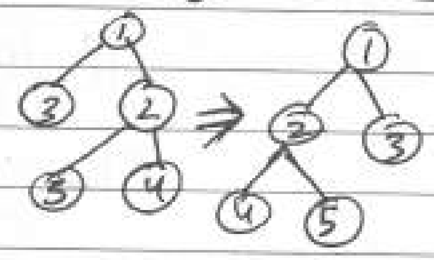
(subtrees are mirrors)

▸ Just do recursive checking.

CODE :-

```
def check(p, q):
    if not p or not q:
        return p == q
    return p.val == q.val and
        check(p.right, q.left) and
        check(p.left, q.right)

return check(root.left, root.right)
```

Similar Question → Generate Mirror of Tree



```
def mirror(root):
    if not root: return
    L, R = root.left, root.right
    root.left = mirror(R)
    root.right = mirror(L)
    return root
```

T.C. → O(n) S.C → O(n) [Skewed Tree case]

classmate

O(logn) O(1)

L.C. 235

Lowest Common Ancestor of Binary Search Tree

APPROACH :- While normal tree needs to be scanned fully, search space in BST can be efficiently pruned in half at each step.

CODE :-

```
temp = root
smaller = p if p.val < q.val else q
larger = p if smaller == q else q

while temp:
    if temp == smaller or temp == larger
       or smaller.val < temp.val < larger.val:
        return temp
    if larger.val < temp.val:
        temp = temp.left
    else:
        temp = temp.right

return temp # won't be called ideally
```


O(n) O(n) (skew-tree)

LC.124

Binary Tree Max path sum

APPROACH ⇒ ▶ Like diameter of tree, calculate ans w.r.t current subtree, but pass recursive return value as if final answer is going to be with some parent.
▶ Also if keeping only current node and discarding children increases return value, then discard those.

CODE :-

```
ans = root.val
def solve(node):
    if not node: return 0
    L,R = solve(node.left), solve(node.right)
    ans = max(ans, node.val + max(0,L) + max(0,R))
    return node.val + max(0,L,R)
solve(root)
return ans.
```

O(n) O(n)

LC-98 Validate Binary Search Tree

APPROACH :- ① We need to make sure each node meets the most restrictive constraints imposed due to parent values
② Readjust limits before calling further recursion.
③ Empty tree is also valid

```
def solve(root, maxi=inf, mini=-inf):
    if not root: return True
    cur=root.val
    return mini<cur<maxi and
        solve(root.left, min(maxi,cur), mini) and
        solve(root.right, maxi, max(mini,cur))

return solve(root)
```


O(n) O(n)

Children Sum Property

① We can only increment node values (not decrement) and sum of children values should be equal to value of parent.

APPROACH → ① If current sum of children is lesser, increase their value each to roots value, and keep recursing. This ensures root's value is always lesser then children's sum-
② Then in the end recalculate for final values

```
def changeTree(root):
    def solve(node):
        if not node: return
        L,R = 0,0
        if node.left:
            L = node.left.data
        if node.right:
            R = node.right.data
        if L+R<node.data:
            if node.left: node.left.data=node.data
            if node.right: node.right.data = node.data
        solve(node.left) ; solve(node.right)
        L,R=0,0
        if node.left: L=node.left.data
        if node.right: R=node.right.data
        if node.left or node.right:
            node.data=L+R
    solve(root)
    return root
```

similar Roof/Ceil $O(\log N)$ $O(1)$

Floor in BST

APPROACH:- We can prune search space in half each time.
- Set largest possible num smaller than X as answer.

CODE:
```
temp = root, ans = -inf
while temp:
    if temp.data == x:
        return x
    elif temp.data < x:
        ans = temp.data
        temp = temp.right
    else:
        temp = temp.left
return ans.
```

$O(n)$ $O(n)$ ($O(\log N)$ for recursion)

LC.108 Get BST from Keys / Convert sorted Array to BST

APPROACH → input keys are sorted, set middle most element as current root. and populate children recursively.
- This will ensure left subtree is always smaller and right subtree will be always greater than root.

CODE:
```
def solve(l, r):
    if l <= r:
        m = (l+r)//2
        node = TreeNode(nums[m])
        node.left = solve(l, m-1)
        node.right = solve(m+1, r)
        return node
return solve(0, len(nums)-1)
```

O(logN) O(1)

similar ⇒ successor

Find inorder predecessor of BST

APPROACH ⇒ i) Since inorder of BST is always sorted, finding predecessor means finding floor of value in BST, which we have already done ☺, just slight modifications in that.

CODE :-

```
predNode, predVal = None, -Inf
temp = root

while temp:
    if predVal < temp.val < target:
        predVal = temp.val
        predNode = temp
    if temp.val < target:
        temp = temp.right
    else:
        temp = temp.left

return predNode
```


Best ⇒ O(n) O(n)

Generate BST from Preorder

APPROACH ① ⇒ start of subarray denotes root of subtree, then bisect (logN) to find placement point of start in rest of subarray left of this point is left subtree and rest is right subtree recursively ⇒ O(nlogn) approach

APPROACH ② ⇒ pointer approach (no bisect) by setting upperbound

- first try placing value in left subtree but if upperbound doesn't allow then place in right

CODE for APPROACH ②

```
n = len(preorder), idx = 0
def helper(upperbound):
    if idx == n or preorder[idx] >= ub:
        return None
    node = TreeNode(preorder[idx++])
    node.left = helper(node.val)
    node.right = helper(ub)
    return node

return helper(inf)
```

Most operations → O(L) T.C.
Not easy to predict space. ↙ average length of word

Implement Trie

APPROACH ⇒ alphabets act as keys and similary subtrees are ~~pointed~~ pointed at and traversed

CODE ⇒ TR = lambda : defaultdict(TR) → #equivalent of #TrieNode class
WORD = 'word'
class Trie:
  def __init__(self):
    self.root = TR()

  def insert(self, word):
    tmp = self.root
    for c in word:
      tmp = tmp[c]
    tmp[WORD] = True
  def search(self, word):
    tmp = self.root
    for c in word:
      if c not in tmp: return False
      tmp = tmp[c]
    return WORD in tmp
  def startsWith(self, prefix):
    tmp = self.root
    for c in prefix: if c not in tmp: return False
      tmp = tmp[c]
    return len(tmp.keys()) > 0

T.C. O(nlogn) + O(Nl) ← for sorting, but longest ↘ average len of word
S.C. O(Nl)

Lexicographically smallest string with all prefixes present

For example ⇒ [ab, abc, a, bp] ⇒ ans ⇒ abc because a, ab, abc all present

APPROACH ⇒ Sort strings by length first so that prefixes get inserted first, and then check how many new nodes are created in each word insertion to trie.
(2) A perfect string won't have to insert any node other than the last node since all previous nodes would be pre-created by prefixes.

CODE ⇒
a.sort(key = lambda x: len(x))
trie = Trie()
longest, longestIdx = 0, -1
for idx, word in enumerate(a):
  insertsDone = insert(trie, word)
  if insertsDone <= 1 and (len(word) > longest
    or len(word) == longest and word < a[longestIdx]):
      longest, longestIdx = len(word), idx
return a[longestIdx] if longestIdx != -1 else "None"

To get insertsDone just increment count var every time node wasn't present already

$O(n^2)$ $O(n^2)$

Count Distinct Substrings of string

- This can be found in polynomial time using trie. In the end, every node of trie represents end of a unique substring. So answer is 1 + trieNode Count [1 extra for empty substring]
- Pick each index of string as potential starting point for substring and do insertion.

CODE ⇒

```
trieNodes = 1
trie = Trie(), n = len(s)
for i in range(n)
    insert(trie, s[i:])
return trieNodes

def insert(trie, word)
    tmp = trie
    for c in word:
        if c not in tmp:
            trieNodes++
        tmp = tmp[c]
```

$O(N)$ $O(2^{maxLen})$ $O(1)$

LC 421

Maximum XOR of 2 nums in an Array

① The key approach is to compare each num in array against all previous nums, and find max XOR.
② max XOR will be obtained when you do XOR of a number with its total opposite (each bit flipped)
③ However that best possible case might not always be available, so we go with closest possible case to that by searching in trie.

CODE ⇒

```
def findMaxXOR(nums):
    trie = Trie(), ans = 0, MAXLEN = len(bin(max(nums))) - 2
    for num in nums:
        self.insert(trie, num)
        bestXOR = self.findClosest(trie, num)
        ans = max(ans, bestXOR)
    return ans

def findClosest(trie, num):
    tmp, ans = trie, 0
    for i in range(MAXLEN-1, -1, -1):
        flippedBit = 1 - (target >> i) & 1
        if flippedBit in tmp:
            ans = ans | 1, tmp = tmp[flippedBit]
        else: tmp = tmp[1-flippedBit]
        ans <<= 1
    return ans >> 1 [Compensate for extra shift]
```

O(n log n) O(1)

LC1707. Max XOR with element from array below upperBound

query structure (x, upperBound)

since queries are offline, what we can do is insert only those elements from array at any point which are smaller or equal to upperBound (By sorting queries and nums).

After that approach is identical to previous question of finding max possible XOR using trie.

In the end we should be able to map answers back to original query index since we sorted queries for our convenience initially, for that you can use hashmap or be just store original index with each query itself.

O(n²) O(n²)

DP

LC.1143

Longest Common Subsequence

APPROACH ▶ Use recursive sol^n, cache output of recursive function.

▶ If current character for both pointers is same, move both forward, otherwise pick best output received by moving only one of them.

CODE ⇒

```
m, n = len(text1), len(text2)

@cache
def lcs(i,j):
    if i==m or j==n:
        return 0
    if text1[i] == text2[j]:
        return 1 + lcs(i+1, j+1)
    else:
        return max(lcs(i+1,j), lcs(i,j+1))

return lcs(0,0)
```

~~O(n^3)~~ O(lmn) O(lmn)

LC. 474

Zeroes And Ones (0/1 Knapsack Modification)

Given max nums of 0's and 1's in total, return largest possible subset of strings which can be made

APPROACH → We need to consider both taking and not taking current string and cache recursive function output.

CODE →

```
strs = [{'0':0,'1':0,**Counter(s)} for s in strs]

@cache
def solve(idx, zeroleft, oneleft):
    if idx == len(strs): return 0
    ans = solve(idx+1, zl, ol)
    cur = strs[idx]
    if cur['0']<=zl and cur['1']<=ol:
        ans = max(ans, 1+solve(idx+1, zl-cur['0'], ol-cur['1']))
    return ans

return solve(0, zl, ol)
```


O(n²) O(n²)

LC. 72 Edit Distance

Given 2 strings you can add, update or delete characters from text1, what is min. operations needed to get text2

APPROACH → simulate recursively and cache

CODE →

```
m, n = len(text1), len(text2)

@cache
def solve(i, j):
    if i == m or j == n:
        return (n-j)+(m-i)        # account for deletions and insertions
    else: if text1[i] == text2[j]:
        return solve(i+1, j+1)
    else:
        return
        1+min(solve(i+1,j+1), solve(i,j+1), solve(i+1,j))
```

- solve(i+1, j+1) → Updating text1
- solve(i, j+1) → assumed insertion in text1
- solve(i+1, j) → assumed deletion in text1

$O(n^3)$ $O(n^2)$

Matrix - Chain - Multiplication

APPROACH ▸ This is a famous pattern in DP which basically implies having a for loop inside the recursive cached function to find the most optimal answer at each step.

▸ In this question also, we run for loop to try placing end of left matrix multiplication and start of right matrix multiplication at each possible mid.

CODE:-

```
@cache
def solve(start, end):
    if end-start >=2:
        ans = inf
        for mid in range(start+1, end):
            ans = min(ans,
                solve(start, mid) +
                solve(mid, end) +
                a[start]*a[mid]*a[end]
            )
        return ans
    else: return 0
return solve(0, N-1)
```


$O(n^2)$ $O(n)$

Increasing Subsequence Maximum Sum

APPROACH ⇒ MCM variation, except we need to consider every index as initial index and only consider next index only if next element is greater than cur.

CODE ⇒

```
@cache
def solve(i):
    ans = arr[i]
    for nex in range(i+1, n):
        if arr[nex] > arr[i]:
            ans = max(ans, arr[i]+solve(nex))
    return ans
return max([solve(i) for i in range(n)])
```

$O(n^2)$

this for loop doesn't make complexity n^3 overall because most solve queries would be served from cache

O(n) O(1)

LC.152 Maximum Product Subarray

Pick max possible non-empty contiguous product

APPROACH ⇒ Have record of max streak and minstreak,
- current number can be 0 or +ve or -ve
- both streaks can be updated, or new ones can be started from current index itself
- Overall result should be stored as well

CODE ⇒

```
n = len(nums)
ans, max_streak, min_streak = nums[0], nums[0], nums[0]

for i in range(1, n):
    cur = nums[i]
    new_max_streak = max(cur, cur*max_streak, cur*min_streak)
    min_streak = min(cur, cur*max_streak, cur*min_streak)
    max_streak = new_max_streak
    ans = max(ans, max_streak)

return ans
```

newstreak
+ve x +ve
-ve x -ve

Similarly minimum might be updated

O(m*n) O(m*n)

Minimum Path Sum in Matrix

You can only move down or right in origin to bottom right cell.

APPROACH ⇒ DFS Recursive DP, move right and down at each step to check.

CODE ⇒

```
moves = [(1,0), (0,1)]

@cache
def dfs(i,j):
    if i>=m or j>=n: return inf
    remaining = inf
    for di,dj in moves:
        remaining = min(remaining, dfs(i+di, j+dj))
    if rest==inf:
        return grid[i][j]
    else:
        return grid[i][j] + rest

return dfs(0,0)
```

this will only happen for bottom-right cell.

$O(n^2)$ $O(n)$

Word Break

Given a dictionary and a string return Yes or no if string can be exactly broken down into words from dictionary.

APPROACH → Try breaking at every point posible.

CODE →

```
dictionary = set(dictionary)
n = len(line)

@cache
def solve(i):
    if i==n: return True
    for j in range(i+1, n+1):
        if line[i:j] in dictionary and
            solve(j): return True
    return False

return solve(0)
```

S.C → $O(n \times Sum(arr))$

T.C → $O(n \times Sum(arr))$

Partition Equal Subset Sum

Return if you can partition array into two subsets of equal value, leaving no element that doesn't belong to one of the two.

APPROACH → If total sum is odd not possible.

▸ In case of even, take half of it and then try finding subsequence with half sum, remaining elements will form other subset.

CODE :-

```
tot = sum(nums)
if tot%2 == 1: return False
@cache
def solve(i, target):
    if i == len(nums): return target == 0
    else:
        return solve(i+1, target) or
               solve(i+1, target - nums[i])
return solve(0, tot//2)
```

$O(n^3)$ $O(n^2)$

Minimum total Cost to Cut a Stick

Cost = length of stick that you're cutting.

APPROACH ⇒ ▶ to cut ~~s~~ope from l to r, ~~both~~ we need to account for all cuts in the range of l to r [end cuts (l or r) don't count].
▶ Then we do matrix chain multiplication pattern to find lowest overall cost of all options

CODE ⇒

```
@cache(~~l,r~~):
def solve(l,r):
    ans = inf
    for cut in cuts:
        if l<cut<r:
            ans = min(ans, r-l+solve(l,cut)+solve(cut,r))
    if ans == inf: return 0
    else: return ans

return solve(0,n)
```

current cut solve

inf means no cut was present between l & r so no cost

classmate

T.C. $O(egg \times floor^2)$ S.C. $O(egg \times floor)$

Egg Dropping Problem

APPROACH ⇒ There is one ground floor and K floors on top of that [You need to check K floors only and not ground one because that's the default answer if all K floors fail].
▶ Another imp. point is that we'll have to check all floors one by one if only 1 egg is left (from bottom to top)

```
@cache(~~floors, eggs~~):
def solve(floors, eggs):
    if eggs == 1 or floors <= 1: return floors
    ans = inf
    for curDrop in range(floors):
        ans = min(ans, 1 + max(
            solve(curDrop, eggs-1),
            solve(floors-1-curDrop, eggs)
        ))
    return ans

return solve(floors, eggs)
```

consider overall minimum

case when egg breaks at curDrop so one egg ~~breaks~~ reduces and try for floors less than curDrop

egg won't break here, go check for higher floors only now

pick worst case here to get correct value

classmate

$O(n^2)$ $O(n^2)$

LC-132 Palindrome Partitioning II

get minimum partitions to have each subpart palindrom i.e.

APPROACH ⇒ at each point see if a palindrome is being formed. If yes, recurse further and find optimal answer using DP.

▶ Also we need to cache output of isPalin to make each check request O(1) amortized

```
n = len(string)

@cache
def isPalin(l,r):
    if l >= r: return True
    return string[l] == string[r] and
                    isPalin(l+1,r-1)

@cache
def solve(i):
    if isPalin(i, n-1): return 0
    else: ans = inf
        for midEnd in range(i, n-1):
            if isPalin(i, midEnd):
                ans = min(ans, 1+solve(midEnd+1))
        return ans
return solve(0)
```

avoid ending after n-1 index because there is nothing after it so it makes no sense to cut there

classmate

T.C. $O(V+E)$ S.C. $O(V)$

Topological Sort for DAGs

Basically all $u \to v$ edges should result in u coming before v in final ordering topologically.

DFS approach → Keep an answer array and only insert a node in it once all its successors in graph have already been explored through DFS (and hence already been pushed to ans). In the end reverse ans array and return.

```
ans = [], vis = set()
def dfs(node):
    if node in vis: return
    vis.add(node)
    for dest in adj[node]: dfs(dest)
    ans.append(node)
for i in range(n): dfs(i)
return ans[::-1]
```

classmate

Can be used for cycle detection — O(V+E) — O(V)

⇓

Toposort using BFS (Kahn's Algorithm) (Can also be used to detect cycle)

- Start with nodes having 0 indegree and traverse from there.

```
q = deque(), ans = [], indegree = {i:0 for i in range(v)}
for i in range(v):
    for dest in adj[i]: indegree[j] += 1
for i in range(v):
    if indegree[i] == 0: q.append(i)
cnt = 0   # ← can be used to detect cycle
while q:
    cur = q.popleft()
    ans.append(cur)
    for dest in adj[cur]:
        indegree[dest] -= 1
        if indegree[dest] == 0:
            q.append(dest)
    cnt += 1
if cnt != V: print("cycle detected")
else: return ans
```

O(V+E) O(V)

Cycle detection in Directed Graph (DFS)

Every node can be in 3 states in a visited d.s.

1. Node Key not present → Means not encountered at all yet
2. Node Key present and value false → seen once but not in current traversal
3. Node Key present and value True → seen and that too in current traversal

⇓

This third case implies a cycle in graph.

CODE:-

```
def dfs(node): # returns true if cycle present in connected component
    if node in vis:
        return vis[node]
    vis[node] = True
    for dest in adj[node]:
        if dfs(dest): return True
    vis[node] = False ← # seen once but not in current path any more
    return False
for i in range(v):
    if dfs(i): print("cycle exists"); return
print("no cycle")
```

O(V+E) O(V)

Clone an undirected graph

APPROACH ⇒ Keep a hashmap of old node to new node map. and do dfs.

```
nodes = { }

def dfs(node):
    if node not in nodes:
        nodes[node] = new Node(node.val)
        for dest in node.neighbors:
            nodes[node].neighbors.append(dfs(dest))
    return nodes[node]

return dfs(originalNode) if originalNode != None
                         else None
```


O(V+E) O(V)

LC. 785

Undirected Graph Bipartite Check (DFS)

APPROACH → We can use 2 colors 0 and 1. Assign 0 to unvisited node and start dfs of each connected component to check for conflicts.

DFS soln →

```
color = { }
def dfs(node, col):
    color[node] = col
    for dest in adj[node]:
        if dest in color:
            if color[dest] == col:
                return False
        else: if not dfs(dest, 1-col):
                return False
    return True
for node in range(len(graph)):
    if node not in color: # connected component
        if not dfs(node, 0): return False
return True #otherwise bipartite.
```

flip color like this

O(V+E) O(V)

Undirected Graph Bipartite check (BFS)

Same as previous question, but BFS implementation.

```
color = {}, q = deque()
for node in range(len(graph)):
    if node not in color:
        q.append(node)
        color[node] = 0
        while q:
            curNode = q.popleft()
            for dest in adj[curNode]:
                if dest in color:
                    if color[dest] == color[curNode]:
                        return False
                else:
                    color[dest] = 1 - color[curNode]
                    q.append(dest)
return True
```

O(V+E) O(V+E)

Kosaraju's algorithm for strongly connected components (in directed graph)

A s.c.c. is a subset of graph nodes where you can reach anywhere from anywhere.

Three steps to Kosaraju algorithm :-

① Topological sort 'like' DFS → (Algo is same but actually topological sorting is only for DAGs, but SCC (graph can have cycle))

② Construct reversed graph

③ Do DFS in reversed graph in topological-like ordering.

CODE → (check topo-sort dfs code written previously)

```
seen = set(), topoLike = []
for i in range(V): topo(i)
topoLike.reverse()
cc = [], ccCount = 0, vis = set(), revAdj = [[] for _ in range(V)]   (new set)
for i in range(V):
    for dest in adj[i]: revAdj[dest].append(i)
for node in topoLike:
    if node not in vis:
        dfs(node), ccCount++;
        print(cc) → basically all unique elements seen in last traversal of dfs
return ccCount
```

(reversing graph; done on revAdj)

O(V+E) O(V+E)

Djikstra Algorithm for shortest distance (weight) from source

Algorithm → have a priority queue and keep getting the lowest known distance edge at every point and if its presence can optimise dist. of any of its neighbors, push them aswell.

CODE →

```
heap = [], dists = [inf]*V
dists[src] = 0
heappush((0, src))
while heap:
    curDist, curNode = heappop(heap)
    if curDist > dists[curNode]:        # stale entry, no need to process
        continue
    for dest, wt in adj[curNode]:
        if curDist + wt < dists[dest]:
            dists[dest] = curDist + wt
            heappush(heap, (dists[dest], dest))
return dists
```

T.C. O(V*E) S.C. O(V)

Bellman ford algorithm for finding Minimum distances from source and also to detect -ve weight cycles

If there are n vertices in a graph, then even in worst case, there can be n-1 edges involved from any u to v node (or lesser edges) but definitelly not more than n-1
Bellman ford Negative cycle detection works on this principle itself by trying an extra n^{th} time.

```
dist = [inf]*V, dist[src] = 0
for relaxationCnt in range(1, V+1):
    for u, v, wt in edges:
        if dist[u] + wt < dist[v]:
            if relaxationCnt == V:
                print("cycle detected")
            dist[v] = dist[u] + wt
print("no cycle detected")
return dist  # minimum distances from source
```

NOTE → Djikstra can't work reliably with negative weights, while Bellman ford can but its T.C. is much worse than djikstra.

T.C. S.C.

$O(E \log E) + O(E)$ $O(E)$

for sorting, for iterating and doing union

Minimum Spanning Tree (Kruskal)

Pick edges sorted on basis of weight and use union find to only consider those edges that unify previously disjoint sets. (connected components.)

▶ Use rank compression to make union find faster

CODE ⇒

```
uf = UF(V)
ans = 0
edges.sort(key = weight)
for u, v, wt in edges:
    if uf.find(u) == uf.find(v): continue
    ans += wt
    uf.union(u, v)
return ans # weight of minimum spanning tree
```


$O(E \log V)$ $O(V+E)$

Minimum Spanning Tree (Prim's Algorithm)

Here we use priority queue to find lowest connection cost on a per node basis. So unlike Kruskal's algorithm, emphasis is on nodes in pq than edges.

▶ inMst is a set or array to account if a node is already included in M.S.T.

▶ At each step, one node from mstset would be connected to one node outside of MST (This is called cut in graph).

```
heap = [], dists = [inf]*V, parent = [-1]*V
inMst = [False]*V
dists[0] = 0, heappush(heap, (0, 0))   # (wt, edge)

while heap:
    u = heappop(heap)
    inMst[u] = True
    for v, wt in adj[u]:
        if not inMst[v] and wt < dists[v]:
            dists[v] = wt, parent[v] = u
            heappush(heap, (wt, v))
return sum(dists)
```

$O(V^3)$ $O(1)$ (not considering adj matrix)

Floyd-Warshall Algorithm

- Work for all sort of graphs (directed and undirected)
- Assume to putting an extra node 'K' between i and j node and see if overall distance improves

CODE ⇒ (Assumes mat[i][j] == inf for cases where edge doesn't exist between i & j

```
for K in range(n):
    for i in range(n):
        for j in range(n):
            mat[i][j] = min(mat[i][j], mat[i][K]+mat[K][j])
```

in case of directed graphs ← For undirected also recalculate → mat[j][i]

```
return mat   # return: min. possible distances from any node to any other node.
```


$O(n)$ $O(n)$

Serialize and deserialize binary Tree

Generally normal VLR traversal is not enough to deserialize a tree uniquely. But if we use modified serialization (means put # or some symbol instead of totally ignoring them)

```
def serialize(root):
    def vlr(root, string):
        if root is None: string += '#,'
        else:
            string += str(root.val) + ','
            string = vlr(root.left, string)
            string = vlr(root.right, string)
        return string
    return vlr(root, '')[:-1]   # ignore last comma

def deserialize(data)
    nodes = data.split(',')
    i = 0   # iterator
    def getTree():
        if nodes[i] == '#': i += 1, return None
        root = TreeNode(int(nodes[i]))
        i += 1
        root.left = getTree()
        root.right = getTree()
        return root
    return getTree()
```

T.C. → O(K) O(1) ↑ S.C

Similar to K^th Largest Element in BST

K^th Smallest element in B.S.T.

▶ APPROACH ⇒ we can do lvr traversal because bst guarantees its LVR traversal being ascending in order.
▶ Doing Morris traversal makes sure we don't take any extra space.

CODE ⇒

```
ct=0
temp=root
while temp:
    if not temp.left:
        ct+=1
        if ct==k: return temp.val
        temp=temp.right
    else:
        rightMost = temp.left
        while rightMost.right and rightMost.right !=temp:
            rightMost = rightMost.right
        if rightMost.right==None:
            rightMost.right = temp
            temp = temp.left
        else: rightMost.right =None
            ct+=1
            if ct==k: return temp.val
            temp = temp.right
```

T.C. O(1) S.C. O(h)

B.S.T Iterator

Print values from smallest to largest (lvr) when asked for using next() method

APPROACH ⇒ Keep a stack of elements yet to be included in next() response. As soon as you pop a value from stack, insert the leftmost path of it right's subtree next is stack.

CODE ⇒

```
def __init__(root):
    self.st = []
    temp = root
    while temp:
        self.st.append(temp)
        temp = temp.left
def next():
    node = self.st.pop()
    temp = node.right
    while temp:
        self.st.append(temp)
        temp = temp.left
    return node.val
def hasNext():
    return !self.st.empty()
```

T.C. O(n) S.C. O(h) or O(n) [skewed Tree]

LC-1373

Largest B.S.T subtree node sum in Tree

return max possible BST sum you can get.

APPROACH ⇒ Recurse and return if current subtree is a BST. If both children are BST and you lie in middle of values of both children, then try resetting final answer value.

CODE ⇒

```
ans = 0
isBST, minVal, maxVal, sum = range(4)   # enum

def solve(node):
    if not node: return (True, inf, -inf, 0)
    L, R = solve(node.left), solve(node.right)
    if L[isBST] and R[isBST] and
       L[maxVal] < node.val < R[minVal]:
        tempSum = L[sum] + R[sum] + node.val
        ans = max(ans, tempSum)
        return (True, min(node.val, L[minVal]),
                      max(node.val, R[maxVal]),
                      tempSum)
    else:
        return tuple([False])   # rest of value classmate dont matter

solve(root)
return ans
```

O(n) O(h)

Two Sum in a BST

APPROACH ⇒ We need l and r pointers of sorted array for two sum, but since this is a tree (we need lvr and rvl iterators. (we need to modify BST iterator code)

```
def findTarget(root, target):
    l, r = BSTIter(root, 'lvr'), BSTIter(root, 'rvl')   # just order flipped from lvr code
    while True:
        L, R = l.peek(), r.peek()   # stack top val
        if L >= R: return False
        temp = L + R
        if temp == k: return True
        elif temp < k: l.next()
        else: r.next()   # pop from stack
```

O(1) Time O(capacity)

LC. 460

LFU Cache ☺

we want ~~LRU~~ behaviour in case of a frequency tie

APPROACH ⇒ We need to maintain a dictionary of ~~LRU~~ caches, one for each frequency.

- Also need to keep track of minimum ~~Freq~~ Frequency currently present in cache.
- In case capacity exceeds, we first remove minFreq ~~LRU~~ from LRU cache
- On updation or get operation, promote node from one frequency level to next. Also increment minFreq value if ~~dict~~ cache corresponding to minFreq becomes empty at any point of time.