Leetcode 75 Grind:

2130. Maximum Twin Sum of a Linked List:

1. Converting Linked List into an array then solving:

class Solution(object):

def pairSum(self, head):

result = []

curr = head

while curr:

result.append(curr.val)

curr = curr.next

print(result)

l, r = 0, len(result)-1

maxi = 0

while l<r:

c\_max = result[l] + result[r]

maxi = max(maxi, c\_max)

l+=1

r-=1

return maxi

2. Using Fast and slow pointers:

# Definition for singly linked list.

# class ListNode(object):

# def \_\_init\_\_(self, val=0, next=None):

# self.val = val

# self.next = next

class Solution(object):

def pairSum(self, head):

"""

:type head: Optional[ListNode]

:rtype: int

"""

slow, fast = head, head

prev = None

while fast and fast.next:

fast = fast.next.next

tmp = slow.next

slow.next = prev

prev = slow

slow = tmp

res = 0

while slow:

res = max(res, prev.val + slow.val)

prev = prev.next

slow = slow.next

return res

3343. Count Number of Balanced Permutations

Optimized  *def count\_balanced\_permutations(num: str) -> int:  
 MOD = 10 \*\* 9 + 7  
  
 # Count total length and partition into even/odd positions.  
 N = len(num)  
 cEven = (N + 1) // 2 # number of even-index positions (0,2,4,...)  
 cOdd = N - cEven # number of odd-index positions  
  
 # Count frequencies of each digit and compute total sum.  
 freq = [0] \* 10  
 total\_sum = 0  
 for ch in num:  
 d = int(ch)  
 freq[d] += 1  
 total\_sum += d  
  
 # If total sum is odd, cannot split evenly -> 0 ways.  
 if total\_sum % 2 == 1:  
 return 0  
 target = total\_sum // 2  
  
 # Precompute factorials and inverse factorials up to N.  
 fact = [1] \* (N + 1)  
 invfact = [1] \* (N + 1)  
 for i in range(1, N + 1):  
 fact[i] = fact[i - 1] \* i % MOD  
 invfact[N] = pow(fact[N], MOD - 2, MOD) # Fermat's little theorem  
 for i in range(N, 0, -1):  
 invfact[i - 1] = invfact[i] \* i % MOD  
  
 # dp[c][s] = ways (mod M) to pick c digits summing to s  
 # from the digits processed so far, weighted by inverse factorials.  
 dp = [[0] \* (target + 1) for \_ in range(cEven + 1)]  
 dp[0][0] = 1  
  
 # Process each digit value d = 0..9  
 for d in range(10):  
 f = freq[d]  
 if f == 0:  
 continue  
 # Precompute weight[x] = 1/(x! \* (f-x)!) for x=0..f  
 weight = [0] \* (f + 1)  
 for x in range(f + 1):  
 weight[x] = (invfact[x] \* invfact[f - x]) % MOD  
  
 # Build new dp after including digit d  
 newdp = [[0] \* (target + 1) for \_ in range(cEven + 1)]  
 for c in range(cEven + 1):  
 for s in range(target + 1):  
 if dp[c][s] == 0:  
 continue  
 ways\_here = dp[c][s]  
 # Try putting x copies of digit d into the even positions  
 maxChoose = min(f, cEven - c)  
 for x in range(maxChoose + 1):  
 new\_sum = s + d \* x  
 if new\_sum > target:  
 break # further x will only increase sum  
 newdp[c + x][new\_sum] = (newdp[c + x][new\_sum]  
 + ways\_here \* weight[x]) % MOD  
 dp = newdp  
  
 # dp[cEven][target] holds the sum of products of invfact terms for all valid distributions.  
 ways\_dist = dp[cEven][target]  
 # Multiply by E! and O! to count full permutations  
 result = ways\_dist \* fact[cEven] % MOD \* fact[cOdd] % MOD  
 return result  
  
  
# Example usage:  
print(count\_balanced\_permutations("123")) # Output: 2  
print(count\_balanced\_permutations("112")) # Output: 1  
print(count\_balanced\_permutations("4567")) # Output: 8*

Brute Force:

def string\_permutations(nums: str):  
 perms = [''.join(p) for p in permutations(nums)]  
 print(perms)  
 count = **0** for p in perms:  
 eSum = **0** oSum = **0** for i in range(len(p)):  
 if i % **2** == **0**:  
 eSum += int(p[i])  
 else:  
 oSum += int(p[i])  
 if eSum == oSum:  
 count += **1** return count  
  
# Example usage  
if \_\_name\_\_ == '\_\_main\_\_':  
 input\_string = "4567"  
 result = string\_permutations(input\_string)  
 print(result)

938. Range Sum of BST

# Definition for a binary tree node.

# class TreeNode(object):

# def \_\_init\_\_(self, val=0, left=None, right=None):

# self.val = val

# self.left = left

# self.right = right

class Solution(object):

def rangeSumBST(self, root, low, high):

"""

:type root: Optional[TreeNode]

:type low: int

:type high: int

:rtype: int

"""

if not root:

return 0

# If current node value is within the range, add it.

if low <= root.val <= high:

return root.val + self.rangeSumBST(root.left, low, high) + self.rangeSumBST(root.right, low, high)

# If value is less than low, move to the right subtree.

elif root.val < low:

return self.rangeSumBST(root.right, low, high)

# If value is greater than high, move to the left subtree.

else:

return self.rangeSumBST(root.left, low, high)

**Converting Any Recursive Function to Iterative (Stack Trick)**

**🚀 Trick Explanation:**

* Recursion works using a **call stack**: each function call is pushed onto the stack, and when the function returns, it is popped off.
* We can **mimic this manually** using an **explicit stack (list)** in Python.
* Each time we would make a recursive call, we instead push the **current state (arguments)** onto the stack.
* Each time we return from a function, we pop the stack and continue from the stored state.

**✅ Example 1: Fibonacci (Recursive to Iterative)**

**📌  Recursive Fibonacci (Inefficient):**

def fibonacci\_recursive(n):

if n <= 1:

return n

return fibonacci\_recursive(n - 1) + fibonacci\_recursive(n - 2)

print(fibonacci\_recursive(5)) # Output: 5

* Each call breaks down into two further calls.
* This leads to an **exponential (2^n) time complexity**.

**📌  Iterative Fibonacci using Stack:**

def fibonacci\_iterative(n):

if n <= 1:

return n

stack = [(n, 'call')]

result = 0

memo = {}

while stack:

value, state = stack.pop()

if state == 'call':

if value <= 1:

memo[value] = value

else:

# Push return state

stack.append((value, 'return'))

# Simulate recursion: push next two calls

stack.append((value - 1, 'call'))

stack.append((value - 2, 'call'))

elif state == 'return':

memo[value] = memo[value - 1] + memo[value - 2]

return memo[n]

print(fibonacci\_iterative(5)) # Output: 5

**📌  How This Works:**

* We use a stack to simulate the call stack.
* Each time we “call” a Fibonacci function, we push its arguments to the stack.
* When returning, we compute the result and store it in memo.
* This is now **linear (O(n))** due to memoization.

**✅ Example 2: Binary Tree Inorder Traversal**

**📌 Recursive Inorder Traversal:**

def inorder\_recursive(root):

if root:

inorder\_recursive(root.left)

print(root.val)

inorder\_recursive(root.right)

**📌  Iterative Inorder Traversal with Stack:**

def inorder\_iterative(root):

stack = []

current = root

while stack or current:

while current:

stack.append(current)

current = current.left

current = stack.pop()

print(current.val)

current = current.right

**📌  How This Works:**

* We use an explicit stack to store nodes.
* We simulate the recursive left traversal by pushing all left children onto the stack.
* We process each node and move to its right child.

**✅  Handling Recursion Limit in Python**

* Python has a default **recursion limit of 1000** (or slightly more, depending on the environment).
* This prevents stack overflow on very deep recursive calls.
* We can increase this limit using sys.setrecursionlimit().

**📌  Example: Setting Recursion Limit:**

import sys

sys.setrecursionlimit(10\*\*6)

def deep\_recursive(n):

if n == 0:

return 0

return 1 + deep\_recursive(n - 1)

print(deep\_recursive(100000)) # Works without stack overflow

**⚠️  Warning:**

* Increasing the recursion limit can cause a **stack overflow** if your system’s memory cannot handle the large number of calls.
* Always prefer the **iterative stack method** for extremely deep recursive problems.

**✅  How to Convert Any Recursive Function to Iterative:**

1. Create a **stack** to simulate the function call stack.
2. Push the **initial arguments and state** onto the stack.
3. Use a while loop:
   * Pop the stack for the **current state/arguments**.
   * If the base case is met, store the result.
   * If not, push the recursive calls onto the stack.
4. Manage a **state marker** (like 'call' and 'return' in the Fibonacci example) to differentiate between a function call and return.

108 array to BST

# Definition for a binary tree node.

# class TreeNode(object):

# def \_\_init\_\_(self, val=0, left=None, right=None):

# self.val = val

# self.left = left

# self.right = right

class Solution(object):

def sortedArrayToBST(self, nums):

def build\_bst(left, right):

if left > right:

return None

mid = (left + right) // 2

root = TreeNode(nums[mid])

root.left = build\_bst(left, mid - 1)

root.right = build\_bst(mid + 1, right)

return root

root = build\_bst(0, len(nums) - 1)

return root

109. linked list build BST

1. converting into array – EASY way

class Solution(object):

def sortedListToBST(self, head):

"""

:type head: Optional[ListNode]

:rtype: Optional[TreeNode]

"""

result = []

curr = head

while curr:

result.append(curr.val)

curr = curr.next

def build\_bst(left, right):

if left > right:

return None

mid = (left + right) // 2

root = TreeNode(result[mid])

root.left = build\_bst(left, mid - 1)

root.right = build\_bst(mid + 1, right)

return root

root = build\_bst(0, len(result) - 1)

return root

2. slow and fast pointer

def sortedListToBST(self, head):

if not head:

return None

if not head.next:

return TreeNode(head.val)

# Find middle using slow-fast pointer

slow, fast = head, head.next.next

while fast and fast.next:

slow = slow.next

fast = fast.next.next

mid = slow.next

slow.next = None # Break the list

root = TreeNode(mid.val)

root.left = self.sortedListToBST(head)

root.right = self.sortedListToBST(mid.next)

return root

3. Global Pointer + In-Order Simulation (Best)

def sortedListToBST(self, head):

self.current = head

length = 0

while head:

length += 1

head = head.next

return self.buildBST(0, length - 1)

def buildBST(self, left, right):

if left > right:

return None

mid = (left + right) // 2

left\_child = self.buildBST(left, mid - 1)

root = TreeNode(self.current.val)

self.current = self.current.next

root.left = left\_child

root.right = self.buildBST(mid + 1, right)

return root

501.Modes of a BST ---- Brute Force using stack

# Definition for a binary tree node.

# class TreeNode(object):

# def \_\_init\_\_(self, val=0, left=None, right=None):

# self.val = val

# self.left = left

# self.right = right

class Solution(object):

def findMode(self, root):

"""

:type root: Optional[TreeNode]

:rtype: List[int]

"""

stack = []

current = root

current\_val = None

current\_count = 0

max\_count = 0

modes = []

while stack or current:

while current:

stack.append(current)

current = current.left

current = stack.pop()

if current.val == current\_val:

current\_count += 1

else:

current\_val = current.val

current\_count = 1

if current\_count > max\_count:

max\_count = current\_count

modes = [current\_val]

elif current\_count == max\_count:

modes.append(current\_val)

current = current.right

return modes

2. using recursion

class Solution(object):

def findMode(self, root):

self.current\_val = None

self.current\_count = 0

self.max\_count = 0

self.modes = []

def in\_order(node):

if not node:

return

in\_order(node.left)

# Count the value

if node.val == self.current\_val:

self.current\_count += 1

else:

self.current\_val = node.val

self.current\_count = 1

if self.current\_count > self.max\_count:

self.max\_count = self.current\_count

self.modes = [node.val]

elif self.current\_count == self.max\_count:

self.modes.append(node.val)

in\_order(node.right)

in\_order(root)

return self.modes

2918. Minimum Equal Sum of Two Arrays After Replacing Zeros

class Solution(object):

def minSum(self, nums1, nums2):

"""

:type nums1: List[int]

:type nums2: List[int]

:rtype: int

"""

c1, c2, sum1, sum2 = nums1.count(0), nums2.count(0), sum(nums1), sum(nums2)

# If one side has no zeros and is smaller than the other’s sum+zeros, no solution:

if (c1 == 0 and sum1 < sum2 + c2):

return -1

if (c2 == 0 and sum2 < sum1 + c1):

return -1

# Otherwise the minimal equal sum is:

return max(sum1 + c1, sum2 + c2)

2300. Successful Pairs of Spells and Potions

class Solution(object):

def successfulPairs(self, spells, potions, success):

"""

:type spells: List[int]

:type potions: List[int]

:type success: int

:rtype: List[int]

"""

ans = []

potions.sort() # Sort potions to enable binary search

m = len(potions)

for spell in spells:

# The minimum potion strength needed

min\_potion = (success + spell - 1) // spell # Equivalent to math.ceil(success/spell)

# Find the first potion that meets the requirement

left, right = 0, m

while left < right:

mid = (left + right) // 2

if potions[mid] < min\_potion:

left = mid + 1

else:

right = mid

ans.append(m - left)

return ans

Transformation of a string 1

DP method:

class Solution(object):

def lengthAfterTransformations(self, s, t):

mod = 10\*\*9 + 7

nums = [0]\*26

for ch in s:

nums[ord(ch) - 97] += 1

for \_ in range(t):

cur = [0]\*26

z = nums[25]

if z:

# 'z' → 'a' and 'b'

cur[0] = (cur[0] + z) % mod

cur[1] = (cur[1] + z) % mod

for j in range(25):

v = nums[j]

if v:

cur[j+1] = (cur[j+1] + v) % mod

nums = cur

res = 0

for v in nums:

res = (res + v) % mod

return res