12/11/20

12/29/20

1/9/21, 1/25/21

2/17/21

3/2/21

Leetcode:

1. Array-based
2. Link-list

Review + new problem

**Need\_to\_review:**

* for i,n in enumerate(**itertools.accumulate**(nums)

**#1- Array**

**----------------------------------------------------------------------------------------------------------------#**

1. **Array - based**
2. **1054: Distant Barcode**

**Input:** barcodes = [1,1,1,1,2,2,3,3]

**Output:** [1,3,1,3,1,2,1,2]

Rearrange the barcodes so that no two adjacent barcodes are equal.

1. Use heapq as **priority Q**.
2. def rearrangeBarcodes(self, barcodes: List[int]) -> List[int]:
3. # 9:53 --> 10:00 --> 10:12 12/11/20
5. cnter = collections.Counter(barcodes)
6. pq = []
8. for v, cnt in cnter.items():
9. heapq.heappush(pq, [-cnt, v])
11. res = []
12. while pq:
13. top = heapq.heappop(pq)
14. res.append(top[1])
15. if pq:
16. top\_next = heapq.heappop(pq)
17. res.append(top\_next[1])
18. top\_next[0] += 1
19. if top\_next[0] < 0:
20. heapq.heappush(pq, top\_next)
22. top[0] += 1
23. if top[0] < 0:
24. heapq.heappush(pq, top)
26. return res
27. **299. Bulls and Cows**

**Input:** secret = "1807", guess = "7810"

**Output:** "1A3B"

Use collections.Counter()

One Linear Scan

def getHint(self, secret: str, guess: str) -> str:

# 3\_2\_21, 3:47 🡪 4:07

gs = collections.Counter(guess)

x, y = 0, 0

for i, v in enumerate(secret):

if secret[i] == guess[i]:

x += 1

gs[v] -= 1

if gs[v] < 0:

gs[v] = 0

y -= 1

elif v in gs and gs[v] > 0:

y += 1

gs[v] -= 1

return str(x) + 'A' + str(y) + 'B'

1. **Increasing Subsequences**
2. **300. Longest Increasing Subsequence**

Simply dp.

**dp = [1] \* n**

**max\_len = 1**

# Recursive back-track

**for i in range(n):**

**for j in range(i):**

**if nums[i] > nums[j]:**

**len\_new** = dp[j]+1

**dp[i]** = len\_new if len\_new > dp[i] else dp[i]

**max\_len** = max(max\_len, dp[i])

**return max\_len**

**#**

**n = len(nums)**

**if n < 1:**

**return 0**

**dp = [1]\*n**

**res = 1**

**for i in range(1,n):**

**max\_len = dp[i]**

**for j in range(i):**

**if nums[i] > nums[j] and dp[j] + 1 > max\_len:**

**max\_len = dp[j] + 1**

**dp[i] = max\_len**

**res = res if res >= max\_len else max\_len**

**return res**

1. **673. Number of Longest Increasing Subsequence**

def findNumberOfLIS(self, nums: List[int]) -> int:

# 10:43 --> 11:00 --> 11;16 9/1/20 Num of longest increasing subsequence

**dp** = [ **[1, 1]** for i in range(len(nums))]

max\_len = 0

max\_cnt = 0

**for i, num in enumerate(nums):**

**for j in range(i):**

**if num > nums[j]:**

len\_new = dp[j][0] + 1

if len\_new > dp[i][0]:

**dp[i][0**], **dp[i][1]** = len\_new, dp[j][1]

elif len\_new == dp[i][0]:

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

**if max\_len == dp[i][0]:**

max\_cnt += dp[i][1]

**if max\_len < dp[i][0]:**

max\_len = dp[i][0]

max\_cnt = dp[i][1]

return **max\_cnt**

1. **1546. Maximum Number of Non-Overlapping Subarrays With Sum Equals Target**

* Can be only one array element
* Use cumulative sum as an array, find diff between elements

**DP solution**

def maxNonOverlapping(self, nums: List[int], target: int) -> int:

n = len(nums)

sums = [0]\*n

**res = [0] \* n**

max\_len = 0

**dic = {}**

**# cumulative SUM**

**for i in range(n):**

**sums[i] = sums[i-1] + nums[i] if i > 0 else nums[i]**

**for i, v in enumerate(sums):**

**val = v - target**

if val in dic:

res[i] = res[dic[val]] + 1

elif val == 0:

res[i] = 1

**dic[v] = i**

if i > 0:

res[i] = max(res[i], res[i-1])

return res[n-1]

* **Greedy, best soln**

**dic = {0:1}**

**cnt = 0**

**cur\_sum = 0**

**for num in nums**:

cur\_sum += num

prev\_sum = cur\_sum - target

if prev\_sum in dic:

**cnt += 1**

**dic = {0:1}**

**cur\_sum = 0**

else:

dic[cur\_sum] = 1

return cnt

* **Greedy using set**

**pre\_sum = set([0])**

**cnt = 0**

**cur\_sum = 0**

**for num in nums**:

cur\_sum += num

if cur\_sum - target in pre\_sum:

**cnt += 1**

**cur\_sum = 0**

**pre\_sum.clear()**

**pre\_sum.add(0)**

else:

pre\_sum.add(cur\_sum)

return cnt

1. **Continuous SUM**
2. **523. Continuous Subarray Sum**

* Brute Force

def checkSubarraySum(self, nums: List[int], k: int) -> bool:

# // 6/11/20

n = len(nums)

**for i, v in enumerate(nums):**

total = v

**for j in range(i+1,n):**

total += nums[j]

if k != 0 and total % k == 0:

return True

if k == total == 0:

return True

return False

* Using Dict

 if sum(**nums[i:j]**) % k == 0 for some i < j, then sum(nums[:j]) % k == sum(nums[:i-1]) % k.

 So we just need to use a dictionary to keep track of sum(nums[:i]) % k and the corresponding index i. Once some later **sum(nums[:i']) % k** == **sum(nums[:i]) % k**and i' - i > 1, we return True.

[0,0]

**def checkSubarraySum(self, nums: List[int], k: int) -> bool:**

# // 6/11/20

**dic = {0:-1}**

**sum\_cur = 0**

for i, v in enumerate(nums):

sum\_cur += v

if k == 0:

if i > 0 and v == nums[i-1]==0:

return True

else:

**tmp** = sum\_cur % k

if tmp in dic:

if i - dic[tmp] > 1:

return True

else:

dic[tmp] = i

return False

-----------------------------------------------------------------------------------

**def checkSubarraySum(self, nums: List[int], k: int) -> bool:**

# // 6/11/20

dic = {0:-1}

sum\_cur = 0

for i, v in enumerate(nums):

sum\_cur += v

if k == 0:

if i > 0 and v == nums[i-1]==0:

return True

else:

**sum\_cur = sum\_cur % k**

if sum\_cur in dic:

if i - dic[sum\_cur] > 1:

return True

else:

dic[sum\_cur] = i

return False

def checkSubarraySum(self, nums: List[int], k: int) -> bool:

# // 6/11/20

dic = {0:-1}

summ = 0

for i, n in enumerate(nums):

if k != 0:

summ = (summ + n) % k

else:

summ += n

if summ not in dic:

dic[summ] = i

else:

if i - dic[summ] >= 2:

return True

return False

1. **Bucket Sort**
2. **220 Contains Duplicate III**

* def containsNearbyAlmostDuplicate(self, nums: List[int], k: int, t: int) -> bool:

# nums[i] - nums[j] <= t

# i - j <= k

if t<0 or k<0:

return False

**allBuckets = {}**

**bucketSize = t+1 #**

**#Two benefits for using t+1 as bucket size:**

**# 1. avoid case t = 0**

**# 2.To fit number of t in each bucket**

**for i in range(len(nums)):**

# m is bucket Index for nums[i]

**m = nums[i]//bucketSize**

#if there is a bucket already present corresponding to current number

if m in allBuckets:

return True

#checking two adjacent buckets m, m-1

if (m-1) in allBuckets and abs(nums[i]-allBuckets[m-1])<bucketSize:

return True

#checking two adjacent buckets m, m+1

if (m+1) in allBuckets and abs(nums[i]-allBuckets[m+1])<bucketSize:

return True

**allBuckets[m]= nums[i]**

#removing the bucket corresponding to number out of our k sized window

**if i>=k:**

**allBuckets.pop( nums[i-k]//bucketSize)**

return False

* **Sorting**

def containsNearbyAlmostDuplicate(self, nums: List[int], k: int, t: int) -> bool:

# nums[i] - nums[j] <= t

# i - j <= k

n = len(nums)

**A = list(zip(nums, range(n)))**

**A.sort()**

for i in range(n):

j = i + 1

while j < n and **A[j][0] - A[i][0]** <= t:

if abs(**A[j][1] - A[i][1]**) <= k:

return True

else:

j += 1

return False

# sorting 2

def containsNearbyAlmostDuplicate(self, nums: List[int], k: int, t: int) -> bool:

# 3/8/21 9:49

# num: t ; i-j: k

n = len(nums)

num\_index = list(zip(nums, range(n)))

num\_index.sort()

for i in range(n-1):

for j in range(i+1,n):

if num\_index[j][0] - num\_index[i][0] <= t and abs(num\_index[j][1] - num\_index[i][1] )<= k :

return True

if num\_index[j][0] - num\_index[i][0] > t:

break

return False

* **DefaultDict**

def containsNearbyAlmostDuplicate(self, nums: List[int], k: int, t: int) -> bool:

# 9:59 --> 10:12 7/24/20

# t -> nums[i] - nums[j]

# k => i - j

if k < 1 or t < 0:

return False

dic = **collections.OrderedDict()**

for num in nums:

key = num if t == 0 else num // t

for m in [dic.get(key-1), dic.get(key), dic.get(key+1)]:

if m is not None and abs(num-m) <= t:

return True

if len(dic) == k:

dic.popitem(last=False)

dic[key] = num

return False

1. **29. Divide Two Integers**

#### # 1. Repeated Subtraction

The key observation to make is that the problems are occurring because there are more negative signed 32-bit integers than there are positive signed 32-bit integers. Each positive signed 32-bit integer has a corresponding negative signed 32-bit integer. However, the same is not true for negative signed 32-bit integers. The smallest one, -2147483648, is alone. It is this number that causes the problems.

The best solution is to work with negative, instead of positive, numbers. This is allows us to use the largest possible range of numbers, and it covers all the ones we need.

def divide(self, dividend: int, divisor: int) -> int:

# Constants.

MAX\_INT = 2147483647 # 2\*\*31 - 1

MIN\_INT = -2147483648 # -2\*\*31

# Special case: overflow.

if dividend == MIN\_INT and divisor == -1:

return MAX\_INT

# We need to convert both numbers to negatives

# for the reasons explained above.

# Also, we count the number of negatives signs.

negatives = 2

if dividend > 0:

negatives -= 1

dividend = -dividend

if divisor > 0:

negatives -= 1

divisor = -divisor

# Count how many times the divisor has to be

# added to get the dividend. This is the quotient.

quotient = 0

while dividend - divisor <= 0:

quotient -= 1

dividend -= divisor

# If there was originally one negative sign, then

# the quotient remains negative. Otherwise, switch

# it to positive.

return -quotient if negatives != 1 else quotient

#### #2. Repeated Exponential Searches

#### def divide(self, dividend: int, divisor: int) -> int:

#### # Constants.

#### MAX\_INT = 2147483647 # 2\*\*31 - 1

#### MIN\_INT = -2147483648 # -2\*\*31

#### HALF\_MIN\_INT = -1073741824 # MIN\_INT // 2

#### # Special case: overflow.

#### if dividend == MIN\_INT and divisor == -1:

#### return MAX\_INT

#### # We need to convert both numbers to negatives.

#### # Also, we count the number of negatives signs.

#### negatives = 2

#### if dividend > 0:

#### negatives -= 1

#### dividend = -dividend

#### if divisor > 0:

#### negatives -= 1

#### divisor = -divisor

#### quotient = 0

#### # Once the divisor is bigger than the current dividend,

#### # we can't fit any more copies of the divisor into it anymore \*/

#### while divisor >= dividend:

#### # We know it'll fit at least once as divivend >= divisor.

#### # Note: We use a negative powerOfTwo as it's possible we might have

#### # the case divide(INT\_MIN, -1). \*/

#### powerOfTwo = -1

#### value = divisor

#### # Check if double the current value is too big. If not, continue doubling.

#### # If it is too big, stop doubling and continue with the next step \*/

#### while value >= HALF\_MIN\_INT and value + value >= dividend:

#### value += value;

#### powerOfTwo += powerOfTwo

#### # We have been able to subtract divisor another powerOfTwo times.

#### quotient += powerOfTwo

#### # Remove value so far so that we can continue the process with remainder.

#### dividend -= value

#### # If there was originally one negative sign, then

#### # the quotient remains negative. Otherwise, switch

#### # it to positive.

#### return -quotient if negatives != 1 else quotient

1. **288. Unique Word Abbreviation**

def \_\_init\_\_(self, dictionary: List[str]):  
 self.dic = collections.defaultdict(set)  
 for s in dictionary:  
 val = s  
 if len(s) > 2:  
 s = s[0] + str(len(s) - 2) + s[-1]  
 self.dic[s].add(val)  
  
  
def isUnique(self, word: str) -> bool:  
 val = word  
 if len(val) > 2:  
 word = word[0] + str(len(word) - 2) + word[-1]  
 return len(self.dic[word]) == 0 or (len(self.dic[word]) == 1 and val == list(self.dic[word])[0])

class ValidWordAbbr:  
  
 def \_\_init\_\_(self, dictionary: List[str]):  
 self.dic = defaultdict(set)  
 for wd in dictionary:  
 n = len(wd)  
 if n == 2:  
 self.dic[wd].add(wd)  
 else:  
 tmp = wd[0] + str(n - 2) + wd[-1]  
 self.dic[tmp].add(wd)  
  
 def isUnique(self, word: str) -> bool:  
 n = len(word)  
 if n == 2:  
 wd = word  
 else:  
 wd = word[0] + str(n - 2) + word[-1]  
  
 if wd not in self.dic:  
 return True  
 else:  
 if len(self.dic[wd]) == 1 and word in self.dic[wd]:  
 return True  
 return False

1. **444. Sequence Reconstruction**

**# Topological Sort**

\*\*What does a super sequence really mean? \*\*

* It means a topological sort of input graph. Therefore every sequence (within the sequences) will be a subsequences of org. How will you check this condition? Every edge (u,v) in sequence will honor this precedence in org i.e. index\_position(u) < index\_position(v)
* For a sequence [5,2,3,6], it is enough to test the edges [5,2], [2,3], and [3,6]. This automatically implies [5,3] and [2,6].

\*\*What is meant by the super sequence being unique? \*\*

* In other words, when will the topological sort be unique? If and only if every consecutive items in org are edges then we have a unique sequence. Use an example: [1,2] and [1,3] will give us two valid super-sequences: [1,2,3] or [1,3,2]. There is no unique sequence since there is no precendence defined for nodes 2 and 3. Here is a wikipedia article about

def sequenceReconstruction(self, org: List[int], seqs: List[List[int]]) -> bool:  
 # 5:51 7/26/20  
  
 if not seqs:  
 return False  
 pos, flags = {}, {}  
 n = len(org)  
 for i in range(n):  
 pos[org[i]] = i  
  
 for s in seqs:  
 for j in range(len(s)):  
 if s[j] not in pos:  
 return False  
 if j > 0:  
 # [1]  
 # [[1,1]]  
 # = sign is important since s can have same values  
  
 if pos[s[j]] <= pos[s[j - 1]]:  
 return False  
 if s[j] not in flags and pos[s[j]] == pos[s[j - 1]] + 1:  
 flags[s[j]] = 1  
  
 return len(flags) == n - 1

1. **468. Validate IP Address**

def validIPAddress(self, IP: str) -> str:  
 def is\_ipv4(str):  
 if **str.count('.'**) != 3:  
 return False  
 res = **str.split(".")**  
 for chrs in res:  
 if not chrs or not chrs.**isdigit():**  
 return False  
 val = **int**(chrs)  
 if val > 255 or (chrs[0] == '0' and len(chrs) > 1):  
 return False  
 return True  
  
 def is\_ipv6(str):  
 if str.count(':') != 7:  
 return False  
 res = str.split(':')  
 for chrs in res:  
 if not chrs or len(chrs) > 4 or not chrs.isalnum():  
 return False  
 for v in chrs:  
 if v.isdigit() or 'a' <= v <= 'f' or 'A' <= v <= "F":  
 continue  
 else:  
 return False  
 return True  
  
 if is\_ipv4(IP):  
 return "IPv4"  
 elif is\_ipv6(IP):  
 return "IPv6"  
 return "Neither"

#

class Solution:  
  
 def v4(self, IP):  
 nums = IP.split('.')  
 for x in nums:  
 if len(x) == 0 or len(x) > 3:  
 return "Neither"  
 if x[0] == '0' and len(x) > 1 or not x.isdigit() or int(x) > 255:  
 return "Neither"  
 return "IPv4"  
  
 def v6(self, IP):  
 nums = IP.split(":")  
 hexdigits = '0123456789abcdefABCDEF'  
 for x in nums:  
 if len(x) == 0 or len(x) > 4 or not all(c in hexdigits for c in x):  
 return "Neither"  
 return "IPv6"  
  
 def validIPAddress(self, IP: str) -> str:  
 # 9:13, 6/03/20  
 if IP.count('.') == 3:  
 return self.v4(IP)  
 elif IP.count(':') == 7:  
 return self.v6(IP)  
 else:  
 return "Neither"

##

class Solution:  
 def validate\_IPv4(self, IP: str) -> str:  
 nums = IP.split('.')  
 for x in nums:  
 # Validate integer in range (0, 255):  
 # 1. length of chunk is between 1 and 3  
 if len(x) == 0 or len(x) > 3:  
 return "Neither"  
 # 2. no extra leading zeros  
 # 3. only digits are allowed  
 # 4. less than 255  
 if x[0] == '0' and len(x) != 1 or not x.isdigit() or int(x) > 255:  
 return "Neither"  
 return "IPv4"  
  
 def validate\_IPv6(self, IP: str) -> str:  
 nums = IP.split(':')  
 hexdigits = '0123456789abcdefABCDEF'  
 for x in nums:  
 # Validate hexadecimal in range (0, 2\*\*16):  
 # 1. at least one and not more than 4 hexdigits in one chunk  
 # 2. only hexdigits are allowed: 0-9, a-f, A-F  
 if len(x) == 0 or len(x) > 4 or not all(c in hexdigits for c in x):  
 return "Neither"  
 return "IPv6"  
  
 def validIPAddress(self, IP: str) -> str:  
 if IP.count('.') == 3:  
 return self.validate\_IPv4(IP)  
 elif IP.count(':') == 7:  
 return self.validate\_IPv6(IP)  
 else:  
 return "Neither"

**#2- Linked List**

**----------------------------------------------------------------------------------------------------------------#**

1. **Linked List**
2. **1474 Delete N nodes after M nodes of a linked list**

**(if there is less than n nodes to remove at the end, remove them as is)**

**def deleteNodes(self, head: ListNode, m: int, n: int) -> ListNode:**

**# 11:48 --> 11:57 11/28/20**

**res = head**

**while True:**

**for \_ in range(m-1):**

**if head:**

**head = head.next**

**else:**

**break**

**if not head:**

**break**

**tail = head**

**for \_ in range(n+1):**

**if tail:**

**tail = tail.next**

**else:**

**break**

**head.next = tail**

**head = tail**

**return res**

1. **206: Reverse a linked list**

**# Recursive Way**

**def reverseList(self, head: ListNode) -> ListNode:**

**# 9:34 11/18/20**

**if not head or not head.next:**

**return head**

**N = self.reverseList(head.next)**

**head.next.next = head**

**head.next = None**

**return N**

**# Iterative way**

**def reverseList(self, head: ListNode) -> ListNode:**

**# 9:34 11/18/20**

**dummy\_head = None**

**while head:**

**node\_next = head.next**

**head.next = dummy\_head**

**dummy\_head = head**

**head = node\_next**

**return dummy\_head**

1. **21 Merge Two Sorted Lists**

**def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:**

**dummy = head = ListNode()**

**while l1 and l2:**

**if l1.val < l2.val:**

**head.next = l1**

**l1 = l1.next**

**else:**

**head.next = l2**

**l2 = l2.next**

**head = head.next**

**if l1:**

**head.next = l1**

**elif l2:**

**head.next = l2**

**return dummy.next**

1. **237. Delete Node in a linked List**

**def deleteNode(self, node):**

**"""**

**:type node: ListNode**

**:rtype: void Do not return anything, modify node in-place instead.**

**"""**

**node.val = node.next.val**

**node.next = node.next.next**

**4.1) 203. Remove Linked List Elements**

**def removeElements(self, head: ListNode, val: int) -> ListNode:**

**# 6:34 11/16/20**

**dummy = ListNode()**

**dummy\_head = dummy**

**dummy.next = head**

**while head:**

**if head.val != val:**

**dummy\_head.next = head**

**dummy\_head = dummy\_head.next**

**head = head.next**

**dummy\_head.next = None**

**return dummy.next**

1. **876. Middle of the Linked List**

**Fast and Slow pointers**

**# if there are Two middle nodes, return the 2nd mid node.**

**def middleNode(self, head: ListNode) -> ListNode:**

**# 11:39 11/18/20**

**slow = fast = head**

**while fast and fast.next:**

**slow = slow.next**

**fast = fast.next.next**

**return slow**

**# if there are Two middle nodes, return the 1st mid node.**

**def middleNode(self, head: ListNode) -> ListNode:**

**# 11:39 11/18/20**

**slow = fast = head**

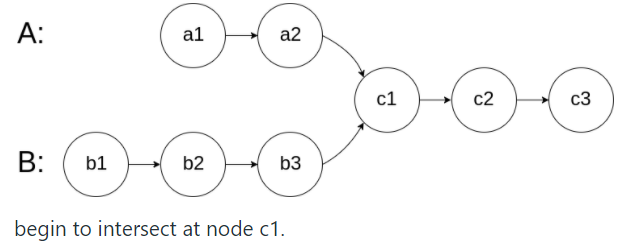
**while fast and fast.next and fast.next.next:**

**slow = slow.next**

**fast = fast.next.next**

**return slow**

1. **160. Intersection of Two Linked Lists**

****

**# 1. Trim the longer one, so both lists can start at the same length.**

**def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:**

**# 1/15/21**

**lenA, lenB = 0 , 0**

**A, B = headA, headB**

**while A:**

**lenA += 1**

**A = A.next**

**while B:**

**lenB += 1**

**B = B.next**

**while lenA > lenB:**

**lenA -= 1**

**headA = headA.next**

**while lenB > lenA:**

**lenB -= 1**

**headB = headB.next**

**while headA != headB:**

**headA = headA.next**

**headB = headB.next**

**return headA**

**# 2.**

**def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:**

**# 5:38 --> 5:56 --> 11/17/20**

**a, b = headA, headB**

**while a != b:**

**a = a.next if a else headB**

**b = b.next if b else headA**

**return a**

1. **141. Linked List Cycle**

**# (start\_1 + m\_steps % len\_cycle) == (start\_2 + 2\*m\_steps % len\_cycle)**

**#**

**def hasCycle(self, head: ListNode) -> bool:**

**# 5:15 11/17/20**

**slow = fast = head**

**while fast and fast.next:**

**fast = fast.next.next**

**slow = slow.next**

**if fast == slow:**

**return True**

**return False**

**# pre and cur**

**def removeElements(self, head: ListNode, val: int) -> ListNode:**

**# 6:34 11/16/20**

**while head:**

**if head.val == val:**

**head = head.next**

**else:**

**break**

**pre, cur = head, head**

**while cur:**

**if cur.val == val:**

**pre.next = cur.next**

**cur = cur.next**

**else:**

**pre = cur**

**cur = cur.next**

**return head**

1. **Palindrome Linked List**

**def isPalindrome(self, head: ListNode) -> bool:**

**# 8:50 11/16/20**

**# Reverse half and compare**

**#find the head of the second half part**

**fast = slow = head**

**while fast and fast.next:**

**fast = fast.next.next**

**slow = slow.next**

**#slow now is the head of second half**

**#reverse the second half**

**prev = None**

**# Multiple assignment, unpacking**

**while slow:**

**slow.next,slow,prev = prev,slow.next,slow**

**#prev now is the head of reversed second half**

**#compare the first part and the second part**

**while prev:**

**if prev.val != head.val:**

**return False**

**prev, head = prev.next, head.next**

**return True**

**##**

**def isPalindrome(self, head: ListNode) -> bool:**

**# 8:50 11/16/20**

**slow = fast = head**

**rev = None**

**while fast and fast.next:**

**rev, slow.next, slow, fast = slow, rev, slow.next, fast.next.next**

**# rev, rev.next, slow, fast = slow, rev, slow.next, fast.next.next**

**if fast: slow = slow.next**

**while slow and slow.val == rev.val:**

**slow, rev = slow.next, rev.next**

**return not slow**

1. **369. Plus One Linked List**

**# reverse, add 1, and reverse again**

**def plusOne(self, head: ListNode) -> ListNode:**

**# 3:24 --> 3:38 ==> 3:46 1/18/21**

**head\_rev = None**

**while head:**

**head\_rev, head.next, head = head, head\_rev, head.next**

**carry = 1**

**head = head\_rev**

**while head\_rev:**

**total = head\_rev.val + carry**

**head\_rev.val = total % 10**

**carry = total // 10**

**if carry == 0:**

**break**

**# if not head\_rev.next and carry == 1:**

**elif not head\_rev.next:**

**head\_rev.next = ListNode(1)**

**break**

**head\_rev = head\_rev.next**

**head\_rev = None**

**while head:**

**head\_rev, head.next, head = head, head\_rev, head.next**

**return head\_rev**

**# Recursive Approach**

1. **1019. Next Greater Node in Linked List**

**# 1 stack**

**def nextLargerNodes(self, head: ListNode) -> List[int]:**

**# 9:02 1/18/21**

**n = 0**

**tmp = head**

**while tmp:**

**n += 1**

**tmp = tmp.next**

**res = [0] \* n**

**loc = 0**

**sk = []**

**while head:**

**if not sk or sk[-1][1] >= head.val:**

**sk.append([loc, head.val])**

**head = head.next**

**loc += 1**

**else:**

**i, val = sk.pop()**

**res[i] = head.val**

**return res**

**#**

**def nextLargerNodes(self, head: ListNode) -> List[int]:**

**# 9:02 11/20/20**

**res, sk, idx = [], [], 0**

**while head:**

**if not sk or sk[-1][0] >= head.val:**

**sk.append([head.val, idx])**

**res.append(0)**

**head = head.next**

**idx += 1**

**else:**

**val, i = sk.pop()**

**res[i] = head.val**

**return res**

1. **817. Linked List Components**

**#**

**def numComponents(self, head: ListNode, G: List[int]) -> int:**

**# 11:53 --> 12: 07 --> 12: 30 , 11/21/20**

**cnt = 0**

**G = set(G)**

**while head:**

**if head.val not in G:**

**head = head.next**

**else:**

**cnt += 1**

**while head and head.val in G:**

**# G.remove(head.val)**

**head = head.next**

**return cnt**

1. **328. Odd Even Linked List**

**7:06**

**#**

**def oddEvenList(self, head: ListNode) -> ListNode:**

**# 1/20/21**

**e = e\_head = ListNode()**

**o = o\_head = ListNode()**

**cnt = 0**

**while head:**

**if cnt % 2 == 0:**

**e.next = head**

**head = head.next**

**e = e.next**

**e.next = None**

**else:**

**o.next = head**

**head = head.next**

**o = o.next**

**o.next = None**

**cnt += 1**

**e.next = o\_head.next**

**return e\_head.next**

**#**

**def oddEvenList(self, head: ListNode) -> ListNode:**

**# 1/20/21**

**e = e\_head = ListNode()**

**o = o\_head = ListNode()**

**cnt = 0**

**while head:**

**if cnt % 2 == 0:**

**e.next = head**

**head = head.next**

**e = e.next**

**# e.next = None**

**else:**

**o.next = head**

**head = head.next**

**o = o.next**

**# o.next = None**

**if not head:**

**o.next = None**

**cnt += 1**

**e.next = o\_head.next**

**return e\_head.next**

1. **725 Split List in Parts**

**def splitListToParts(self, root: ListNode, k: int) -> List[ListNode]:**

**# 1:12 --> 1:35 --> 11/24/20**

**res = [None] \* k**

**head = root**

**len\_nodes = 0**

**while head:**

**len\_nodes += 1**

**head = head.next**

**avg, ext = divmod(len\_nodes, k)**

**i = 0**

**while i < k and root:**

**res[i] = root**

**n = avg + 1 if i < ext else avg**

**for j in range(1, n):**

**root = root.next**

**tmp = root.next**

**root.next = None**

**root = tmp**

**i += 1**

**return res**

1. **24. Swap Nodes in Pairs**

**def swapPairs(self, head: ListNode) -> ListNode:**

**dummy = dummy\_head = ListNode()**

**dummy.next = head**

**while head and head.next:**

**tmp = head.next.next**

**dummy.next, dummy.next.next = head.next, head**

**dummy = head**

**head = dummy.next = tmp**

**return dummy\_head.next**

1. **148. Sort List**

**def sortList(self, head: ListNode) -> ListNode:**

**# 11/27/20, 8:51**

**def get\_mid(head):**

**# mid\_prev = None**

**# while head and head.next:**

**# mid\_prev = head if not mid\_prev else mid\_prev.next**

**# head = head.next.next**

**# mid = mid\_prev.next**

**# mid\_prev.next = None**

**# return mid**

**mid = head**

**while head and head.next:**

**if mid != head:**

**mid = mid.next**

**head = head.next.next**

**tmp = mid.next**

**mid.next = None**

**return tmp**

**def merge(l1, l2):**

**head = dummy = ListNode()**

**while l1 and l2:**

**if l1.val < l2.val:**

**head.next = l1**

**l1 = l1.next**

**else:**

**head.next = l2**

**l2 = l2.next**

**head = head.next**

**if l1:**

**head.next = l1**

**if l2:**

**head.next = l2**

**return dummy.next**

**def merge1(list1, list2):**

**head = dummy = ListNode()**

**while list1 and list2:**

**if list1.val < list2.val:**

**head.next = list1**

**list1 = list1.next**

**else:**

**head.next = list2**

**list2 = list2.next**

**head = head.next**

**if list1:**

**head.next = list1**

**else:**

**head.next = list2**

**return dummy.next**

**if not head or not head.next:**

**return head**

**mid\_node = get\_mid(head)**

**left = self.sortList(head)**

**right = self.sortList(mid\_node)**

**return merge(left, right)**

1. **147. Insertion Sort List**

**def insertionSortList(self, head: ListNode) -> ListNode:**

**# 6:11, 11/5/20**

**# 11/7/20**

**# 11/28/20**

**# 11/29/20 4:53**

**# 11/30/20, 12:16**

**new\_head = ListNode()**

**cur = new\_head.next = head**

**while cur and cur.next:**

**if cur.val <= cur.next.val:**

**cur = cur.next**

**else:**

**next\_node = cur.next.next**

**cur\_head = new\_head**

**while cur\_head.next.val < cur.next.val:**

**cur\_head = cur\_head.next**

**cur.next.next = cur\_head.next**

**cur\_head.next = cur.next**

**cur.next = next\_node**

**return new\_head.next**

1. **143. Reorder List**

**def reorderList(self, head: ListNode) -> None:**

**"""**

**Do not return anything, modify head in-place instead.**

**"""**

**# 2:48 9/29/20**

**# 1:41 12/2/20**

**# 6:03 -- 1/29/21,**

**if not head or not head.next or not head.next.next:**

**return head**

**# find the middle one**

**pre = slow = fast = head**

**while fast and fast.next:**

**pre = slow**

**slow = slow.next**

**fast = fast.next.next**

**if fast: # odd**

**mid = slow.next**

**slow.next = None**

**else: # even**

**mid = slow**

**pre.next = None**

**# reverse mid**

**tail = None**

**while mid:**

**mid.next, tail, mid = tail, mid, mid.next**

**new\_head = ListNode()**

**while head and tail:**

**new\_head.next = head**

**head = head.next**

**new\_head = new\_head.next**

**new\_head.next, tail, new\_head = tail, tail.next, tail**

**if head: # odd**

**new\_head.next = head**

**return new\_head.next**

**#**

**def reorderList(self, head: ListNode) -> None:**

**"""**

**Do not return anything, modify head in-place instead.**

**"""**

**# 2:48 9/29/20**

**if not head:**

**return head**

**# find mid**

**slow, fast = head, head**

**while fast.next and fast.next.next:**

**slow = slow.next**

**fast = fast.next.next**

**mid = slow.next**

**slow.next = None**

**# reverse mid**

**mid\_rev = None**

**while mid:**

**tmp\_next = mid.next**

**mid.next = mid\_rev**

**mid\_rev = mid**

**mid = tmp\_next**

**# combine**

**while head and mid\_rev:**

**tmp\_next = head.next**

**head.next = mid\_rev**

**mid\_rev = mid\_rev.next**

**head.next.next = tmp\_next**

**head = tmp\_next**