12/11/20

12/29/20

**2021**

Jan: 9, 25

Feb: 17,

March: 2, 15,

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"

1. **1191. K-Concatenation Maximum Sum**

* if k==1, which is the same as find the maximum subarray;
* if k>1:
  1. if the sum of the array is less than or equal to 0, we can each find max subarray in the first array or concatenation of two arraies, for example
     1. [1,2,-4]+[1,2,-4] the subarray max is 1+2 = 3, which is in the first array;
     2. [1,-4,1]+[1,-4,1] the subarray max is 1+1 = 2, which is in the first two array;
  2. if the sum of the array is greater than 0, we have to add the(k-1)\*sum(array) and maximum subarray of array,

1. **prefix+[maximum subarry] + suffix +(k-2)\*sum(array)+ prefix+[maximum subarry] + suffix** =

prefix + **[maximum subarry] +**

**(k-2) \*sum(array)+ suffix +**

**prefix+[maximum subarry] + suffix =**

prefix + **[maximum subarry] +**

**(k-2) \*sum(array)+ (suffix +**

prefix**+[maximum subarry]) + suffix =**

*prefix* + **[maximum subarry] +**

**(k-1) \*sum(array) +**

*suffix*

1. (prefix+[maximum subarry] + suffix + prefix+[maximum subarry] + suffix) +(k-2)\*sum(array), so we only need to consider the first part, the maximum subarray sum of the first part is [maximum subarry] + suffix + prefix+[maximum subarry] = sum(array)+maximum subarray, since the sum of the array is greater than 0, so it must be the sum(array)+subarrrysum. hope it helpes.
2. def kConcatenationMaxSum(self, arr: List[int], k: int) -> int:  
    # def kConcatenationMaxSum(self, arr, k):  
    *"""* ***:type*** *arr: List[int]* ***:type*** *k: int* ***:rtype****: int  
    """* m = 10 \*\* 9 + 7  
     
    def kadane(arr):  
    cur = 0  
    res = 0  
    for a in arr:  
    cur = max(a, cur + a)  
    res = max(res, cur)  
     
    return res  
     
    if k == 1:  
    return kadane(arr)  
    elif k == 2:  
    return kadane(arr \* 2) % m  
    else:  
    if sum(arr) <= 0:  
    return kadane(arr \* 2)  
    else:  
    return (kadane(arr) + (k - 1) \* sum(arr)) % m
3. **91. Decode Ways**

def numDecodings(self, s: str) -> int:  
 if not s:  
 return 0  
  
 dp = [0 for \_ in range(len(s) + 1)]  
 dp[0] = 1  
 dp[1] = 0 if s[0] == '0' else 1  
  
 for i in range(2, len(dp)):  
  
 # Check if successful single digit decode is possible.  
 if s[i - 1] != '0':  
 dp[i] += dp[i - 1]  
  
 # Check if successful two digit decode is possible.  
 two\_digit = int(s[i - 2: i])  
 if two\_digit >= 10 and two\_digit <= 26:  
 dp[i] += dp[i - 2]  
 return dp[len(s)]

def numDecodings(self, s: str) -> int:  
 # 12/25/2019   
 # 11:15-->11:21-->11:31-->11:39

def numDecodings(self, s: str) -> int:  
 def helper(s, i, i\_dict):  
  
 if i == len(s):  
 return 1  
  
 if s[i] == '0':  
 return 0  
  
 if i == len(s) - 1:  
 return 1  
  
 if i in i\_dict:  
 return i\_dict[i]  
  
 if int(s[i:i + 2]) < 27:  
 ans = helper(s, i + 1, i\_dict) + helper(s, i + 2, i\_dict)  
 else:  
 ans = helper(s, i + 1, i\_dict)  
  
 i\_dict[i] = ans  
  
 return ans  
  
 if not s:  
 return 0  
 i\_dict = {}  
 return helper(s, 0, i\_dict)

def numDecodings(self, s: str) -> int:  
 # 6:45 --> 6:59 3/15/21  
 # 1,2,3,..9   
 # 10  
 # 11, ...19,  
 # 20,  
 # 21...26  
  
 # dp  
 n = len(s)  
 if not s or s[0] == '0':  
 return 0  
 dp = [0 for \_ in range(n + 1)]  
 dp[0] = dp[1] = 1  
 for i in range(2, n + 1):  
 com\_num = int(s[i - 2:i])  
 if 10 <= com\_num <= 26:  
 dp[i] += dp[i - 2]  
 elif s[i - 1] == '0':  
 return 0  
 if s[i - 1] != '0':  
 dp[i] += dp[i - 1]  
  
 return dp[n]

def numDecodings(self, s: str) -> int:  
  
 # dp  
 n = len(s)  
 if not s or s[0] == '0':  
 return 0  
 dp = [0 for \_ in range(n + 1)]  
 dp[0] = dp[1] = 1  
 for i in range(2, n + 1):  
 com\_num = int(s[i - 2:i])  
 if 10 <= com\_num <= 26:  
 dp[i] += dp[i - 2]  
 # Assume alway has a valid soln  
 # elif s[i-1] == '0':  
 # return 0  
 if s[i - 1] != '0':  
 dp[i] += dp[i - 1]  
  
 return dp[n]

def numDecodings(self, s: str) -> int:  
  
 **def helper(s, i, i\_dict):**  
 if i == len(s):  
 return 1  
 if s[i] == '0':  
 return 0  
 if i == len(s) - 1:  
 return 1  
  
 if i in i\_dict:  
 return i\_dict[i]  
  
 if int(s[i:i + 2]) < 27:  
 ans = helper(s, i + 1, i\_dict) + helper(s, i + 2, i\_dict)  
 else:  
 ans = helper(s, i + 1, i\_dict)  
  
 i\_dict[i] = ans  
 return ans  
  
 if not s:  
 return 0  
 i\_dict = {}  
 return helper(s, 0, i\_dict)

def numDecodings(self, s: str) -> int:  
  
 def helper(s, i, dic):  
 if i == len(s) or (i == len(s) - 1 and s[i] != '0'):  
 return 1  
 elif s[i] == '0':  
 return 0  
  
 if i in dic:  
 return dic[i]  
  
 ans = helper(s, i + 1, dic)  
  
 if int(s[i:i + 2]) <= 26:  
 ans += helper(s, i + 2, dic)  
  
 dic[i] = ans  
  
 return ans  
  
 dic = {}  
 return helper(s, 0, dic)

1. **15. 3Sum**

def threeSum(self, nums: List[int]) -> List[List[int]]:  
 # 12:00, 3-26-21 (1)  
  
 n = len(nums)  
 nums.sort()  
 res = []  
  
 for i in range(n - 2):  
 if nums[i] > 0:  
 break  
 if i > 0 and nums[i] == nums[i - 1]:  
 continue  
  
 head, tail = i + 1, n - 1  
 while head < tail:  
 total = nums[head] + nums[tail]  
 if total == -nums[i]:  
 res += [[nums[i], nums[head], nums[tail]]]  
 head = head + 1  
 while head < tail and nums[head] == nums[head - 1]:  
 head += 1  
 # while head < tail:  
 # if nums[head] == nums[head-1]:  
 # head += 1  
 # else:  
 # break  
  
 elif total < -nums[i]:  
 head += 1  
 else:  
 tail -= 1  
  
 return res

def threeSum(self, nums):  
 *"""* ***:type*** *nums: List[int]* ***:rtype****: List[List[int]]  
 """* rst = []  
 nums.sort()  
 n = len(nums)  
  
 if n < 3:  
 return []  
  
 for i in range(n - 2):  
 if i > 0 and nums[i] == nums[i - 1]:  
 continue  
 if nums[i] > 0:  
 break  
 lt, rt = i + 1, n - 1  
 while lt < rt:  
 total = nums[lt] + nums[rt] + nums[i]  
 if total == 0:  
 rst.append([nums[i], nums[lt], nums[rt]])  
 lt += 1  
 rt -= 1  
 while lt < rt:  
 if nums[lt] == nums[lt - 1]:  
 lt += 1  
 elif nums[rt] == nums[rt + 1]:  
 rt -= 1  
 else:  
 break  
 elif total < 0:  
 lt += 1  
 else:  
 rt -= 1  
 return rst

1. **130. Surrounded Regions**

**# 1.**

def solve(self, board: List[List[str]]) -> None:  
 *"""  
 Do not return anything, modify board in-place instead.  
 """* # 5:00 --> 5:23 3/28/21  
 def helper(board, m, n, r, c):  
 if r == m or c == n or r < 0 or c < 0: #  
 return  
 if board[r][c] == 'O':  
 board[r][c] = 'v'  
 for x, y in [(0, 1), (0, -1), (-1, 0), (1, 0)]:  
 helper(board, m, n, r + x, c + y)  
 return  
  
 m, n = len(board), len(board[0])  
 for i in [0, m - 1]:  
 for j in range(n):  
 if board[i][j] == 'O':  
 helper(board, m, n, i, j)  
  
 for j in [0, n - 1]:  
 for i in range(m):  
 if (i == m - 1 and (j == 0 or j == n - 1)) or (i == 0 and (j == 0 or j == n - 1)):  
 continue  
 if board[i][j] == "O":  
 helper(board, m, n, i, j)  
  
 for i in range(m):  
 for j in range(n):  
 if board[i][j] == 'O':  
 board[i][j] = 'X'  
 if board[i][j] == 'v': #  
 board[i][j] = 'O'

def helper(board, i, j):  
 if i < 0 or i > len(board) - 1 or j < 0 or j > len(board[0]) - 1:  
 return  
 if board[i][j] == 'O':  
 board[i][j] = 'Y'  
 dir = [(0, 1), (0, -1), (1, 0), (-1, 0)]  
 for r, c in dir:  
 helper(board, r + i, c + j)  
  
  
if not board or not board[0]:  
 return  
  
m, n = len(board), len(board[0])  
for i in [0, m - 1]:  
 for j in range(n):  
 helper(board, i, j)  
  
for j in [0, n - 1]:  
 for i in range(m):  
 helper(board, i, j)  
  
for i in range(m):  
 for j in range(n):  
 if board[i][j] == 'O':  
 board[i][j] = 'X'  
 if board[i][j] == 'Y':  
 board[i][j] = 'O'

1. **464. Can I Win**

**Top Down DFS with Memoization: Time: O(N \* 2^N). Space: O(2^N)**

class Solution:  
 def canIWin(self, maxChoosableInteger: int, desiredTotal: int) -> bool:  
  
 # 11:34 3/30/21  
 def helper(nums, total, cur\_sum, dic):  
 # (nums, desiredTotal, 0, {})  
 if not nums:  
 return False  
 elif nums in dic:  
 return dic[nums]  
 else:  
 dic[nums] = False  
 if cur\_sum + max(nums) >= total:  
 dic[nums] = True  
 else:  
 for num in nums:  
 tmp\_nums = tuple(x for x in nums if x != num)  
 if not helper(tmp\_nums, total, cur\_sum + num, dic):  
 dic[nums] = True  
 break  
 return dic[nums]  
  
 nums = tuple(i for i in range(1, 1 + maxChoosableInteger))  
 if sum(nums) < desiredTotal:  
 return False  
 return helper(nums, desiredTotal, 0, {})

1. **1712. Ways to Split Array Into Three Subarrays**

class Solution:  
 def waysToSplit(self, nums: List[int]) -> int:  
 # 3/29/21 10:10  
 # LeftSum ≤ MidSum ≤ (TotalSum−LeftSum) // 2  
  
 def bsearchLeft(presum, i, leftSum):  
 lt, rt = i + 2, len(presum) - 1 # wronng len(presum), lt = i+1  
 while lt < rt:  
 mid = lt + (rt - lt) // 2  
 midSum = presum[mid] - leftSum  
 if midSum < leftSum:  
 lt = mid + 1  
 else:  
 rt = mid  
 return rt  
  
 def bsearchRight(presum, i, target):  
 lt, rt = i + 2, len(presum) - 1 # # wronng len(presum), lt = i + 1  
 while lt < rt:  
 mid = lt + (rt - lt) // 2  
 midsum = presum[mid] - presum[i + 1]  
 if midsum > target:  
 rt = mid  
 else:  
 lt = mid + 1  
 return lt - 1  
  
 n = len(nums)  
 MOD = 10 \*\* 9 + 7  
 presum = [0] \* (n + 1)  
 for i in range(n):  
 presum[i + 1] = presum[i] + nums[i]  
  
 result = 0  
 for i in range(n - 2):  
 leftSum = presum[i + 1]  
 remain = presum[n] - leftSum  
 if remain < leftSum \* 2:  
 break  
 first = bsearchLeft(presum, i, leftSum)  
 last = bsearchRight(presum, i, remain // 2)  
  
 result += max(last - first + 1, 0)  
  
 return result % MOD

1. **457. Circular Array Loop**

**3/31/21**

A loop is not found if any step returns to the same index or moves in the opposite direction.

**# Use set() to track cycle**

class Solution:  
 def circularArrayLoop(self, nums: List[int]) -> bool:  
 # 6:05, 3/31/21  
  
 n = len(nums)  
 visited = [False] \* n  
  
 for i in range(n):  
 if visited[i]:  
 continue  
 cycle = set([i])  
 cur = i  
 visited[i] = True  
 while True:  
 next\_index = (cur + nums[cur]) % n  
 if next\_index == cur or nums[next\_index] \* nums[cur] < 0:  
 break  
 if next\_index in cycle:  
 return True  
 cycle.add(next\_index)  
 cur = next\_index  
 visited[next\_index] = True  
 return False

**# Use fast/slow pointers for detecting cycle**

1. **1353. Maximum Number of Events That Can Be Attended**

**Sort the events based on the start-time:**

**For the same start-time, add all the end time to a min-heap**

**Remove the event with the smallest end time first, while incrementing current day.**

class Solution:  
 def maxEvents(self, events: List[List[int]]) -> int:  
 # Maximum Number of events that can be attended  
 # 3:41 pm 8/13/20  
 # 2:57 pm, 4/3/2021  
  
 events.sort()  
 end\_day\_heap = []  
 res = d = 0  
 i = 0  
  
 while i < len(events) or end\_day\_heap:  
 if not end\_day\_heap:  
 d = events[i][0]  
 while i < len(events) and events[i][0] == d:  
 heapq.heappush(end\_day\_heap, events[i][1])  
 i += 1  
 heapq.heappop(end\_day\_heap)  
 res += 1  
 d += 1  
 while end\_day\_heap and end\_day\_heap[0] < d:  
 heapq.heappop(end\_day\_heap)  
 # if not end\_day\_heap:  
 # i += 1  
 return res

**# method 2**

class Solution:  
 def maxEvents(self, events: List[List[int]]) -> int:  
 # Maximum Number of events that can be attended  
 # 3:41 pm 8/13/20  
  
 events.sort(reverse=1)  
 h = []  
 res = d = 0  
 while events or h:  
 if not h: d = events[-1][0]  
 while events and events[-1][0] <= d:  
 heapq.heappush(h, events.pop()[1])  
 heapq.heappop(h)  
 res += 1  
 d += 1  
 while h and h[0] < d:  
 heapq.heappop(h)  
 return res

1. **5. Longest Palindromic Substring**

class Solution:  
 def longestPalindrome(self, s: str) -> str:  
 # 8:47 --> 9:03, 4/3/21  
  
 n = len(s)  
 if n < 2: # extra  
 return s # extra  
 dp = [[0] \* n for \_ in range(n)]  
 max\_p = 1  
 max\_str = "" # extra  
  
 for i in range(n):  
 dp[i][i] = 1  
 for j in range(i):  
 # dp[j][i]  
 if s[j] == s[i]:  
 if i - j <= 2:  
 dp[j][i] = 1  
 else:  
 if dp[j + 1][i - 1]:  
 dp[j][i] = 1  
 if dp[j][i]:  
 if i - j + 1 > max\_p:  
 max\_p = i - j + 1  
 max\_str = s[j:i + 1]  
 if max\_p == 1: # extra  
 return s[0] # extra  
 return max\_str

class Solution:  
 def longestPalindrome(self, s):  
 longest = 1  
 lt, rt = 0, 0  
 # return s[lt:rt+1]  
 n = len(s)  
 dp = [[0]\*n for \_ in range(n)]  
  
 for i in range(n):  
 for j in range(i+1):  
 if i == j:  
 dp[i][j] = 1  
 else:  
 if s[i] == s[j]:  
 if i - j <= 2 or dp[j+1][i-1]:  
 dp[j][i] = 1  
 if longest < i - j + 1:  
 longest = i - j + 1  
 lt, rt = j, i  
  
 return s[lt: rt+1]

1. **456. 132 Pattern**

class Solution:  
 def find132pattern(self, nums: List[int]) -> bool:  
 # 4:14, 4/4/21  
  
 n = len(nums)  
 min\_v = nums[0]  
  
 for i in range(n):  
 if nums[i] == min\_v:  
 continue  
 elif nums[i] < min\_v:  
 min\_v = nums[i]  
 continue  
 else: # nums[i] > min\_v  
 for j in range(i + 1, n):  
 if min\_v < nums[j] < nums[i]:  
 return True  
 return False

**# Logic**

**# first num = I, If nums[i+1] >=**

# logic

# first num i: if nums[i+1] >= nums[i], i++

# 2nd num: j, j= i+1, if nums[j]<= nums[j+1], j++

# 3rd num: k, k = j+1, if nums[i] < nums[k] < nums[j], -> True

Else: k += 1 until n

If not find k until n: meaning every num after j , is smaller than nums[i], so

Repeat with i = j + 1

class Solution:  
 def find132pattern(self, nums: List[int]) -> bool:  
 # 4:14, 4/4/21  
 n = len(nums)  
 i = j = 0  
 while i < n:  
 while i < n-1 and nums[i] >= nums[i+1]:  
 i += 1  
 j = i + 1  
 while j < n-1 and nums[j] <= nums[j+1]:  
 j += 1  
 k = j+1  
 while k < n:  
 if nums[i] < nums[k] < nums[j]:  
 return True  
 k+= 1  
 i = j + 1  
 return False

1. **179. Largest Number**

class Solution:  
 def largestNumber(self, nums: List[int]) -> str:  
 # 11:00 8/6/20  
 # 9:32 4/7/21  
  
 def sort\_decrease(x, y):  
 if x + y > y + x:  
 return -1  
 elif x + y < y + x:  
 return 1  
 else:  
 return 0  
  
 nums\_str = [str(num) for num in nums]  
 nums\_str.sort(key=functools.cmp\_to\_key(sort\_decrease))  
  
 res = "".join(nums\_str)  
  
 return res if res[0] != '0' else '0'

class Solution:  
 def largestNumber(self, nums: List[int]) -> str:  
 # 11:00 8/6/20  
  
 if not any(nums):  
 return "0"  
 return str(int("".join(sorted(map(str, nums), key= \  
 functools.cmp\_to\_key(lambda x, y: int(y + x) - int(x + y))))))

class Solution:  
 def largestNumber(self, nums: List[int]) -> str:  
 # 11:00 8/6/20  
 # 9:32 4/7/21  
  
 def sort\_decrease(x, y):  
 if x + y > y + x:  
 return -1  
 else:  
 return 1  
 # else:  
 # return 0  
  
 nums\_str = [str(num) for num in nums]  
 nums\_str.sort(key=functools.cmp\_to\_key(sort\_decrease))  
  
 res = "".join(nums\_str)  
  
 return res if res[0] != '0' else '0'

1. **1169. Invalid Transactions**

class Solution:  
 def invalidTransactions(self, transactions: List[str]) -> List[str]:  
 # 10:19 --> 10:52 4/8/21  
 dic = collections.defaultdict(list)  
 transactions = [s.split(',') for s in transactions]  
  
 for v in transactions:  
 k = v[0]  
 dic[k].append(v)  
  
 res = []  
 for t\_name, trans in dic.items():  
 trans.sort(key=lambda x: x[3])  
 n = len(trans)  
 invalid\_list = [False] \* n  
 for i, v in enumerate(trans):  
  
 if int(v[2]) > 1000:  
 if not invalid\_list[i]:  
 res.append(",".join(v))  
 invalid\_list[i] = True  
  
 for j in range(i + 1, n):  
 if trans[j][3] == v[3]:  
 continue  
 if -60 <= int(trans[j][1]) - int(v[1]) <= 60:  
 if not invalid\_list[j]:  
 invalid\_list[j] = True  
 res.append(",".join(trans[j]))  
 if not invalid\_list[i]:  
 invalid\_list[i] = True  
 res.append(",".join(trans[i]))  
  
 return res

class Solution:  
 def invalidTransactions(self, transactions: List[str]) -> List[str]:  
 # 11:40, 4/17/20  
  
 # transaction format  
 # [name, time, amount, city]  
  
 # ["bob,689,1910,barcelona","bob,832,1726,barcelona","bob,820,596,bangkok"]  
 res = []  
 records = []  
 for t in transactions:  
 rec = t.split(',')  
 rec[1] = int(rec[1])  
 rec[2] = int(rec[2])  
 records.append(rec)  
 for rec in records:  
 if rec[2] > 1000:  
 rec[1] = str(rec[1])  
 rec[2] = str(rec[2])  
 res.append(','.join(rec))  
 continue  
 for x in records:  
 if rec[0] == x[0] and abs(rec[1] - int(x[1])) <= 60 and rec[3] != x[3]:  
 rec[1] = str(rec[1])  
 rec[2] = str(rec[2])  
 res.append(','.join(rec))  
 break  
 return res

1. **50. Pow(x, n)**

class Solution:  
 def myPow(self, x: float, n: int) -> float:  
 # 3:13 4/8/21  
  
 if x == 0 or n == 1: ##  
 return x  
 elif n == 0: ##  
 return 1  
 elif n < 0:  
 return 1 / self.myPow(x, -n)  
 else:  
 if n % 2 == 0:  
 return self.myPow(x \* x, n / 2)  
 else:  
 return x \* self.myPow(x \* x, (n - 1) / 2)

class Solution:  
 def myPow(self, x, n):  
 *"""* ***:type*** *x: float* ***:type*** *n: int* ***:rtype****: float  
 """* if n < 0:  
 return self.myPow(1/x, -n)  
 if n == 0:  
 return 1  
 if n == 1:  
 return x  
 if n % 2:  
 return x\*self.myPow(x\*x, n // 2)  
 else:  
 return self.myPow(x\*x, n//2)

class Solution:  
 def myPow(self, x, n):  
 *"""* ***:type*** *x: float* ***:type*** *n: int* ***:rtype****: float  
 """* if n == 0 : return 1  
 if n < 0 :   
 x = 1/x  
 n = -n  
 rst, mul = x, 1  
  
 while n > 0:  
 if n == 1:  
 break  
 if n % 2 == 1:  
 mul \*= rst  
 rst \*=rst  
 n //=2  
 return rst \* mul

1. **365. Water and Jug Problem**

only thing we should proof is this:

if x and y are coprime, then we can and only can reach every integer z in [0, x + y]. (1)

then for a GCD g, from gx and gy,  
we can and only can reach every z in {i \* g | i in [0, x + y] }

now, let's see how to proof (1).  
let x be the less one, and y the greater one.  
then fill the two jug to full, we have x and y water each and x + y water in total.  
then we pour out x water each time until we can't.

now we have these different z:

y + x, y, y - x, y - 2x, ... , y % x

finally we have y % x water left, we pour it into the x jug,  
then fill the y jug to full.  
now the two jugs have y % x and y water separately,  
and y + y % x water in total.  
then we pour from y jug into x jug until the x jug is full,  
afterwards do the same thing like before,  
to pour out x water each time until we can't.

finally we get (y + y % x) % x = (y % x + y % x) % x = (2y) % x water left.

now we have these different z:

y + y % x, y + y % x - x, y + y % x - 2x, ... , (2y) % x

do this x times, we get z:

y + (2y) % x, y + (2y) % x - x, y + (2y) % x - 2x, ..., (3y) % x

:

:

:

y + ((x-1)y) % x, y + ((x-1)y) % x - x, y + ((x-1)y) % x - 2x, ... , (xy) % x

then you see (xy) % x = 0, and

set { y % x, (2y) % x, (3y) % x, ... , ((x-1)y) % x } just equals to { 1, 2, 3, 4, 5, ... , x - 1 } . (2)

proof for (2):  
modulo x could get x - 1 different results at most exclusive 0, that's 1,2,3,...,x-1.  
we have x - 1 expressions, suppose there is two same,  
let a != b in [1, x-1] and (ay) % x = (by) % x,  
then we get ((a - b)y) % x = 0,  
then ((a - b) % x) \* (y % x) = 0, (a - b) % x = 0.  
for 1 <= a, b <= x - 1, so we get a = b. it's impossible.

# Greatest Common Denominator  
def gcd(x, y):  
 return x if y == 0 else gcd(y, x % y)  
  
print(gcd(6,9)) # 3   
print(gcd(3,5)) # 1

1. **Longest Substring Without Repeating Characters**

**# sliding window**

class Solution:  
 def lengthOfLongestSubstring(self, s: str) -> int:  
 # 3:27 4/10/21  
 if not s:  
 return 0  
 n = len(s)  
 lt, rt = 0, 1  
 max\_len = 1  
 dic = {s[0]: 0}  
 while rt < n:  
 if s[rt] in dic:  
 repeat\_pos = dic[s[rt]]  
 if repeat\_pos >= lt:  
 max\_len = max(max\_len, rt - lt)  
 lt = repeat\_pos + 1  
 dic[s[rt]] = rt  
 rt += 1  
  
 return max(max\_len, rt - lt)

1. **678. Valid Parenthesis String**

**# Using two Stacks**

class Solution:  
 def checkValidString(self, s: str) -> bool:  
 # 5:35 4/11/21  
 left, star = [], []  
 for i, v in enumerate(s):  
 if v == '(':  
 left.append(i)  
 elif v == ')':  
 if left:  
 left.pop()  
 elif star:  
 star.pop()  
 else:  
 return False  
 else:  
 star.append(i)  
  
 # if len(left) != len(star): # worng  
 if len(left) > len(star):  
 return False  
 while left and star:  
 if star[-1] > left[-1]:  
 star.pop()  
 left.pop()  
 else:  
 return False  
 return True

**# recursive**

class Solution:  
 def checkValidString(self, s: str) -> bool:  
 # 5:35 4/11/21  
  
 def helper(s, i, cnt, dic):  
 if i == len(s):  
 return cnt == 0  
  
 pattern = str(i) + '+' + str(cnt) ##  
 if pattern in dic:  
 return dic[pattern]  
  
 if s[i] == '(':  
 dic[pattern] = helper(s, i + 1, cnt + 1, dic)  
 elif s[i] == ')':  
 if cnt - 1 < 0:  
 dic[pattern] = False  
 else:  
 dic[pattern] = helper(s, i + 1, cnt - 1, dic)  
 else: # '\*'  
 dic[pattern] = helper(s, i + 1, cnt, dic) or helper(s, i + 1, cnt + 1, dic) or \  
 (cnt > 0 and helper(s, i + 1, cnt - 1, dic))  
  
 return dic[pattern]  
  
 return helper(s, 0, 0, {})

**# Scan Left and then Scan Right**

class Solution:  
 def checkValidString(self, s: str) -> bool:  
  
 n = len(s)  
 left\_cnt = 0  
 for i in range(n):  
 if s[i] == '(' or s[i] == '\*':  
 left\_cnt += 1  
 else:  
 left\_cnt -= 1  
 if left\_cnt < 0:  
 return False  
  
 if left\_cnt == 0: return True  
 right\_cnt = 0  
 for i in range(n - 1, -1, -1):  
 if s[i] == ')' or s[i] == "\*":  
 right\_cnt += 1  
 else:  
 right\_cnt -= 1  
 if right\_cnt < 0:  
 return False  
  
 return True

1. **61. Rotate List**

**# pay attention to : when linked list is**

**Empty, and**

**When k % size = 0**

# Definition for singly-linked list.  
# class ListNode:  
# def \_\_init\_\_(self, val=0, next=None):  
# self.val = val  
# self.next = next  
class Solution:  
 def rotateRight(self, head: ListNode, k: int) -> ListNode:  
 # 1:46 --> 1:57 4/12/21  
 if k == 0 or not head:  
 return head  
 cur, tail = head, None  
 size = 0  
 while cur:  
 size += 1  
 if not cur.next:  
 tail = cur  
 cur = cur.next  
  
 move = k % size  
 if move == 0: # missed  
 return head  
  
 move = size - move - 1  
 cur = head  
 while move > 0:  
 cur = cur.next  
 move -= 1  
 cur.next, new\_head = None, cur.next  
 tail.next = head  
  
 return new\_head

1. **708. Insert into a Sorted Circular Linked List**

****

****

****

*"""  
# Definition for a Node.  
class Node:  
 def \_\_init\_\_(self, val=None, next=None):  
 self.val = val  
 self.next = next  
"""*# Corner case:  
# 1. [] empty circular list  
# 2. [1] , only one element in the circular list  
# 3. [3,3,3], circular list consists elments which are all the same  
  
# Insert into a Sorted circular linked list  
class Solution:  
 def insert(self, head: 'Node', insertVal: int) -> 'Node':  
 # 4:36 8/15/20  
 if not head: # empty list  
 new\_node = Node(insertVal)  
 new\_node.next = new\_node  
 head = new\_node  
 # return head  
 else:  
 cur, nxt = head, head.next  
 while nxt != head:  
 if cur.val <= insertVal <= nxt.val:  
 break  
 elif cur.val > nxt.val:  
 if insertVal >= cur.val or insertVal <= nxt.val:  
 break  
 # else:  
 cur, nxt = nxt, nxt.next  
  
 # deal with list of all the same elements [3,3,3] and [1]  
 new\_node = Node(insertVal)  
 cur.next = new\_node  
 new\_node.next = nxt  
  
 return head

class Solution:  
 def insert(self, head: 'Node', insertVal: int) -> 'Node':  
  
 if head == None:  
 newNode = Node(insertVal, None)  
 newNode.next = newNode  
 return newNode  
  
 prev, curr = head, head.next  
 toInsert = False  
  
 while True:  
  
 if prev.val <= insertVal <= curr.val:  
 # Case #1.  
 toInsert = True  
 elif prev.val > curr.val:  
 # Case #2. where we locate the tail element  
 # 'prev' points to the tail, i.e. the largest element!  
 if insertVal >= prev.val or insertVal <= curr.val:  
 toInsert = True  
  
 if toInsert:  
 prev.next = Node(insertVal, curr)  
 # mission accomplished  
 return head  
  
 prev, curr = curr, curr.next  
 # loop condition  
 if prev == head:  
 break  
 # Case #3.  
 # did not insert the node in the loop  
 prev.next = Node(insertVal, curr)  
 return head

1. **152. Maximum Product Subarray**

**# dp**

class Solution:  
 def maxProduct(self, nums: List[int]) -> int:  
 res, mx, mn = nums[0], nums[0], nums[0]  
 for i in range(1, len(nums)):  
 if nums[i] > 0:  
 mx = max(mx \* nums[i], nums[i])  
 mn = min(mn \* nums[i], nums[i])  
 else:  
 tmp = mx  
 mx = max(mn \* nums[i], nums[i])  
 mn = min(tmp \* nums[i], nums[i])  
 res = max(res, mx)  
  
 return res

**### LAST**

**#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**

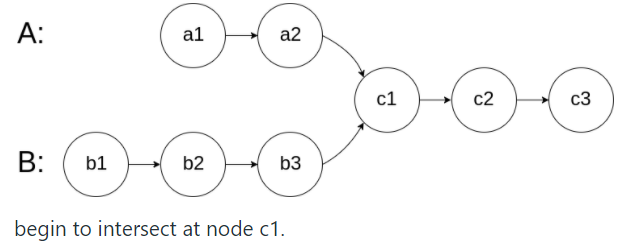
**while fast and fast.next and fast.next.next:**

**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**