12/11/20

12/29/20

**2021**

Jan: 9, 25

Feb: 17,

March: 2, 15,

5/5/21

* Solve new problems to master coding skills and habits and learn new problem-solving routines
* Review old problems to focus on memorizing problem-solving routines so don’t have to start solving problems with instinct.

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

1. **Longest Well-Performing Interval**

****

class Solution:  
 def longestWPI(self, hours: List[int]) -> int:  
 # longest Well-Performing Interval  
 # 4/14/21  
  
 res = total = 0  
 dic = {}  
  
 for i, v in enumerate(hours):  
 total += 1 if v > 8 else -1  
 if total > 0:  
 res = i + 1  
 else:  
 # major point is only need to check if " total-1" in the dic  
 #  
 if total - 1 in dic:  
 res = max(res, i - dic[total - 1])  
 if total not in dic:  
 dic[total] = i  
 return res

1. **556. Next Greater Element III**

Medium

Given a positive integer n, find *the smallest integer which has exactly the same digits existing in the integer* n *and is greater in value than* n. If no such positive integer exists, return -1.

**Note** that the returned integer should fit in **32-bit integer**, if there is a valid answer but it does not fit in **32-bit integer**, return -1.

**Example 1:**

**Input:** n = 12

**Output:** 21

**Example 2:**

**Input:** n = 21

**Output:** -1

**Constraints:**

* 1 <= n <= 231 - 1

**# 1.**

class Solution:  
 def nextGreaterElement(self, n: int) -> int:  
 # 1:10 --> 1:30 --> 4/15/21  
 s = list(str(n))  
 size = len(s)  
 for i in range(size-1,-1,-1):  
 if i == 0:  
 return -1  
 if s[i] > s[i-1]:  
 sort\_list = sorted(s[i:])  
 j = 0  
 while j < len(sort\_list):  
 if sort\_list[j] > s[i-1]:  
 s[i-1], sort\_list[j] = sort\_list[j], s[i-1]  
 break  
 j += 1  
 s[i:] = sort\_list  
 res = int("".join(s))  
 if res > 2\*\*31-1 or res < 2\*\*-31:  
 return -1

# if n < 1 >> 31 or n > (1 << 31) - 1:  
 # return -1  
 return res  
 return -1

**# 2**

class Solution:  
 def nextGreaterElement(self, n: int) -> int:  
 nums = list(str(n))  
  
 m = len(nums)  
 i = m - 1  
 while i > 0:  
 if nums[i] > nums[i - 1]:  
 j = i  
 while j < m:  
 # if nums[j] > nums[i]:  
 if nums[j] > nums[i - 1]:  
 j += 1  
 else:  
 break  
 # nums[i], nums[j-1] = nums[j-1], nums[i]  
 nums[i - 1], nums[j - 1] = nums[j - 1], nums[i - 1]  
 nums[i:] = sorted(nums[i:])  
 break  
 i -= 1  
  
 if i == 0:  
 return -1  
 n = int("".join(nums))  
 if n < 1 >> 31 or n > (1 << 31) - 1:  
 return -1  
 else:  
 return n

1. **418. Sentence Screen Fitting**

class Solution:  
 def wordsTyping(self, sentence: List[str], rows: int, cols: int) -> int:  
 # 2:29 4/15/21  
 sentence = " ".join(sentence) + " "  
 start = 0  
 n = len(sentence)  
  
 for i in range(rows):  
 start += cols  
  
 if sentence[start % n].isspace():  
 start += 1  
 else:  
 while start > 0 and not sentence[(start - 1) % n].isspace():  
 start -= 1  
  
 return start // n

1. **31. Next Permutation**

class Solution:  
 def nextPermutation(self, nums: List[int]) -> None:  
 # 8:57 3/12/20  
 # 10:44 7/1/2020  
 # 1 2 4 3 1 -> 1 4 2 3 1 --> 1 3 1 2 4  
  
 n = len(nums)  
 for i in range(n - 1, 0, -1):  
 if nums[i] > nums[i - 1]:  
 j = i  
 while j < n:  
 if nums[j] <= nums[i - 1]:  
 break  
 j += 1  
 nums[i - 1], nums[j - 1] = nums[j - 1], nums[i - 1]  
 nums[i:] = sorted(nums[i:])  
 return nums  
  
 nums.sort()  
 return nums

class Solution:  
 def nextPermutation(self, nums: List[int]) -> None:  
 # 8:57 3/12/20  
 # 10:44 7/1/2020  
 # 1 2 4 3 1 -> 1 4 2 3 1 --> 1 3 1 2 4  
  
 n = len(nums)  
 for i in range(n - 1, 0, -1):  
 if nums[i] > nums[i - 1]:  
 j = i  
 while j < n:  
 if nums[j] <= nums[i - 1]:  
 break  
 j += 1  
 nums[i - 1], nums[j - 1] = nums[j - 1], nums[i - 1]  
 nums[i:] = sorted(nums[i:])  
 return nums  
  
 nums.sort()  
 return nums

1. **522. Longest Uncommon Subsequence II**

class Solution:  
 def findLUSlength(self, strs: List[str]) -> int:  
  
 # find all the sub-sequence of s  
 # save them in list res  
 def sub\_seq(s, i, tmp, res):  
 if i == len(s):  
 return  
 for j in range(i, len(s)):  
 tmp\_res = tmp + s[j]  
 if tmp\_res not in res:  
 res.append(tmp + s[j])  
 sub\_seq(s, j + 1, tmp\_res, res)  
  
 res = []  
 subseq\_map = collections.Counter()  
 for s in strs:  
 sub\_seq(s, 0, "", res)  
 # save sub-sequences in a dic  
 subseq\_map.update(res)  
 res.clear()  
  
 max\_len = -1  
 for s in subseq\_map:  
 if subseq\_map[s] == 1:  
 max\_len = max(max\_len, len(s))  
 return max\_len

class Solution:  
 def findLUSlength(self, strs: List[str]) -> int:  
  
 # find all the seb-sequence of s  
 # save them in list res  
 def sub\_seq(s, i, tmp, res):  
 if i == len(s):  
 return  
 for j in range(i, len(s)):  
 tmp\_res = tmp + s[j]  
 if tmp\_res not in res:  
 res.append(tmp + s[j])  
 sub\_seq(s, j + 1, tmp\_res, res)  
  
 res = []  
 subseq\_map = collections.Counter()  
 for s in strs:  
 sub\_seq(s, 0, "", res)  
 # save sub-sequences in a dic  
 subseq\_map.update(res)  
 res.clear()  
  
 max\_len = -1  
 res = subseq\_map.most\_common()  
 for i in range(len(res) - 1, -1, -1):  
 s, size = res[i][0], res[i][1]  
 if size == 1:  
 max\_len = max(max\_len, len(s))  
 else:  
 break  
 return max\_len

class Solution:  
 def findLUSlength(self, strs: List[str]) -> int:  
 # brute force  
 # ["a","b","c","d","e","f","a","b","c","d","e","f"]  
 def is\_common(sub, t):  
 i = 0  
 for ch in t:  
 if ch == sub[i]:  
 i += 1  
 if i == len(sub):  
 break  
 # continue  
 return i == len(sub)  
  
 mx\_len = -1

strs = sorted(strs, key=len, reverse=True)  
 for i in range(len(strs)):  
 flag = True  
 for j in range(len(strs)):  
 if i == j:  
 continue  
 if is\_common(strs[i], strs[j]):  
 flag = False  
 break  
 if flag:

#if j == len(strs) - 1 and flag:   
 mx\_len = max(mx\_len, len(strs[i]))  
 return mx\_len

1. **918. Maximum Sum Circular Subarray**

最长子数组的范围可以有两种情况，一种是正常的，数组中的某一段子数组，另一种是分为两段的，即首尾相连，对于第二种情况，需要转换一下思路，除去两段的部分，中间剩的那段子数组其实是和最小的子数组，只要用之前的方法求出子数组的最小和，用数组总数字和一减，同样可以得到最大和。两种情况的最大和都要计算出来，取二者之间的较大值才是真正的和最大的子数组.

但是这里有个 corner case 需要注意一下，假如数组中全是负数，那么和最小的子数组就是原数组本身，则求出的差值是0，而第一种情况求出的和最大的子数组也应该是负数，那么二者一比较，返回0就不对了，所以这种特殊情况需要单独处理一下

class Solution:  
 def maxSubarraySumCircular(self, A: List[int]) -> int:  
 # Maximum Sum Circular Subarray  
 # 8/22/20 3:41  
  
 total = curMax = curMin = 0  
 maxSum = minSum = A[0]  
 for a in A:  
 curMax = max(curMax + a, a)  
 maxSum = max(maxSum, curMax)  
 curMin = min(curMin + a, a)  
 minSum = min(minSum, curMin)  
 total += a  
 if total == minSum:  
 return maxSum   
 # max\_sum < 0 , total - minSum = 0;  
 # so can't use max(maxSum, total - minSum)  
 else:  
 return max(maxSum, total - minSum)

1. **53. Maximum Subarray**

Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

class Solution:  
 def maxSubArray(self, nums: List[int]) -> int:  
 cur\_max = 0  
 max\_val = nums[0]  
  
 for v in nums:  
 cur\_max = max(cur\_max + v, v)  
 max\_val = max(max\_val, cur\_max)  
  
 return max\_val

1. **1073. Adding Two Negabinary Numbers**

class Solution:  
  
 def addBinary(self, A, B):  
 res = []  
 carry = 0  
 while A or B or carry:  
 carry += (A or [0]).pop() + (B or [0]).pop()  
 res.append(carry & 1)  
 carry = carry >> 1  
 return res[::-1]  
  
 def addNegabinary(self, arr1: List[int], arr2: List[int]) -> List[int]:  
 # Adding Two Negabinary Numbers  
 # 11:43 8/24/2020  
  
 res = []  
 carry = 0  
 while arr1 or arr2 or carry:  
 carry += (arr1 or [0]).pop() + (arr2 or [0]).pop()  
 res.append(carry & 1)  
 # 这里由于是负二进制，所以右移1位之后再取负  
 carry = -(carry >> 1)  
 while len(res) > 1 and res[-1] == 0:  
 res.pop()  
 return res[::-1]

1. **1477. Find Two Non-overlapping Sub-arrays Each With Target Sum**

class Solution:  
 def minSumOfLengths(self, arr: List[int], target: int) -> int:  
 # Find two Non-overlapping sub-arrays each with target sum  
 # 12:21 4/22/21  
 # Sliding window  
  
 def get\_sub\_arrays(arr, target):  
 dic = collections.defaultdict(int)  
 cur\_sum = 0  
 dp = [float('inf')] \* len(arr)  
  
 for i, v in enumerate(arr):  
 cur\_sum += v  
 if cur\_sum == target:  
 dp[i] = i + 1  
 elif cur\_sum - target in dic:  
 dp[i] = i - dic[cur\_sum - target]  
 dic[cur\_sum] = i  
  
 if i > 0:  
 dp[i] = min(dp[i - 1], dp[i])  
  
 return dp  
  
 dp\_left = get\_sub\_arrays(arr, target)  
 dp\_right = get\_sub\_arrays(arr[::-1], target)[::-1]  
  
 res = float('inf')  
 for i in range(1, len(arr)):  
 left, right = dp\_left[i - 1], dp\_right[i]  
 if left < float('inf') and right < float('inf'):  
 res = min(res, left + right)  
 return res if res != float('inf') else -1

class Solution:  
 def minSumOfLengths(self, arr: List[int], target: int) -> int:  
 # Find two Non-overlapping sub-arrays each with target sum  
 # 9:26 8/11/20  
 # prefix sum  
 s, lsize, res = 0, float('inf'), float('inf')  
 prefixSum = {0: -1}  
  
 for i, val in enumerate(arr):  
 s += val  
 prefixSum[s] = i  
  
 s = 0  
 for i, val in enumerate(arr):  
 s += val  
  
 if s - target in prefixSum:  
 lsize = min(i - prefixSum[s - target], lsize)  
  
 if s + target in prefixSum and lsize != float('inf'):  
 rsize = prefixSum[s + target] - i  
 res = min(res, rsize + lsize)  
  
 return res if res != float('inf') else -1

1. **954. Array of Doubled Pairs**

class Solution:  
 def canReorderDoubled(self, A: List[int]) -> bool:  
 # Array of Doubled Pairs 9/24 8/31/20  
 c = collections.Counter(A)  
 for x in sorted(c, key=abs):  
 if c[x] > c[x \* 2]:  
 return False  
 c[x \* 2] -= c[x]  
  
 return True

1. **1386. Cinema Seat Allocation**

class Solution:  
 def maxNumberOfFamilies(self, n: int, reservedSeats: List[List[int]]) -> int:  
  
 # 12:05 TLE  
 dic = collections.defaultdict(set)  
  
 for seat in reservedSeats:  
 dic[seat[0]].add(seat[1])  
  
 res = 0  
 for row in range(1, n + 1):  
 if row not in dic:  
 res += 2  
 continue  
 temp = 0  
 if (2 not in dic[row]) and (3 not in dic[row]) and 4 not in dic[row] and 5 not in dic[row]:  
 temp += 1  
 if 6 not in dic[row] and 7 not in dic[row] and 8 not in dic[row] and 9 not in dic[row]:  
 temp += 1  
  
 if temp == 0 and (4 not in dic[row] and 5 not in dic[row] and 6 not in dic[row] and 7 not in dic[row]):  
 temp += 1  
  
 res += temp  
  
 return res

class Solution:  
 def maxNumberOfFamilies(self, n: int, reservedSeats: List[List[int]]) -> int:  
 # Cinema Seat Allocation 10:24 8/27/20  
  
 res = 2 \* n  
 rows = collections.defaultdict(set)  
 for r, c in reservedSeats:  
 rows[r].add(c)  
 middle = {2, 3, 4, 5, 6, 7, 8, 9}  
 left = {2, 3, 4, 5}  
 right = {6, 7, 8, 9}  
 small\_middle = {4, 5, 6, 7}  
 for r in rows:  
 reserved = rows[r]  
 # 2 choices  
 if len(reserved & middle) == 0:  
 continue  
 # 1 choice  
 elif len(reserved & left) == 0 or len(reserved & right) == 0 or len(reserved & small\_middle) == 0:  
 res -= 1  
 # 0 choice  
 else:  
 res -= 2  
 return res

1. **33. Search in Rotated Sorted Array**

class Solution:  
 def search(self, nums: List[int], target: int) -> int:  
 # 11:14 4/23/21  
  
 def b\_search(nums, lt, rt, target):  
 while lt <= rt:  
 mid = lt + (rt - lt) // 2  
 if nums[mid] > target:  
 rt = mid - 1  
 elif nums[mid] < target:  
 lt = mid + 1  
 else:  
 return mid  
 return -1  
  
 n = len(nums)  
 lt, rt = 0, n - 1  
 while lt <= rt:  
 mid = lt + (rt - lt) // 2  
 if nums[mid] == target:  
 return mid  
 if nums[mid] < nums[rt]:  
 if nums[mid] < target <= nums[rt]:  
 return b\_search(nums, mid, rt, target)  
 else:  
 rt = mid - 1  
 else:  
 if nums[lt] <= target < nums[mid]:  
 return b\_search(nums, lt, mid - 1, target)  
 else:  
 lt = mid + 1  
 return -1

class Solution:  
 def search(self, nums: List[int], target: int) -> int:  
 # 7/4/2020  
  
 n = len(nums)  
 lt, rt = 0, n - 1  
 while lt <= rt:  
 mid = lt + (rt - lt) // 2  
 if nums[mid] == target:  
 return mid  
 if nums[mid] < nums[rt]:  
 if nums[mid] < target <= nums[rt]:  
 lt = mid + 1  
 else:  
 rt = mid - 1  
 elif nums[mid] > nums[rt]:  
 if nums[lt] <= target < nums[mid]:  
 rt = mid - 1  
 else:  
 lt = mid + 1  
 else:  
 rt = mid - 1  
  
 return -1

class Solution:  
 def search(self, nums: List[int], target: int) -> int:  
 # 7/4/2020  
  
 n = len(nums)  
 lt, rt = 0, n- 1  
 while lt <= rt:  
 mid = lt + (rt - lt) // 2  
 if nums[mid] == target:  
 return mid  
 if nums[mid] < nums[rt]:  
 if nums[mid] < target <= nums[rt]:  
 lt = mid + 1  
 else:  
 rt = mid - 1  
 elif nums[mid] >= nums[rt]:  
 if nums[lt] <= target < nums[mid]:  
 rt = mid - 1  
 else:  
 lt = mid + 1  
  
 return -1

1. **146. LRU Cache**

class LRUCache:  
  
 def \_\_init\_\_(self, capacity: int):  
 self.capacity = capacity  
 self.dic = collections.OrderedDict()  
  
 def get(self, key: int) -> int:  
 if key not in self.dic:  
 return -1  
 val = self.dic[key]  
 self.dic.move\_to\_end(key)  
 return val  
  
 def put(self, key: int, value: int) -> None:  
 self.dic[key] = value  
 self.dic.move\_to\_end(key)  
 if len(self.dic) > self.capacity:  
 self.dic.popitem(last=False)

1. **1146. Snapshot Array**

class SnapshotArray:  
 def \_\_init\_\_(self, length: int):  
 self.cache = []  
 self.d = dict()  
 self.i = -1  
  
 def set(self, index: int, val: int) -> None:  
 self.d[index] = val  
  
 def snap(self) -> int:  
 self.cache.append(dict(self.d))  
 self.i += 1  
 return self.i  
  
 def get(self, index: int, snap\_id: int) -> int:  
 snap = self.cache[snap\_id]  
 return snap[index] if index in snap else 0

1. **842. Split Array into Fibonacci Sequence**

Input:**"0123"**

Output:**[01,2,3]**

Expected:**[]**

class Solution:  
 def splitIntoFibonacci(self, S: str) -> List[int]:  
 # 5:33 4/22  
  
 def backtrack(ans, s):  
 if not s and len(ans) > 2:  
 ans[:] = [int(v) for v in ans]  
 # ans = [int(v) for v in ans]  
 return True  
  
 len\_min = 1  
 if len(ans) >= 2:  
 next\_num = int(ans[-1]) + int(ans[-2])  
 len\_min = len(str(next\_num))  
  
 for i in range(len\_min, len(s) + 1):  
 tmp = int(s[:i])  
 if tmp > (1 << 31) - 1:  
 break  
 if len(ans) < 2 or tmp == next\_num:  
 if i > 1 and s[0] == '0':  
 break  
 ans.append(s[:i])  
 if backtrack(ans, s[i:]):  
 return True  
 ans.pop()  
 return False  
  
 ans = []  
 backtrack(ans, S)  
 return ans

class Solution:  
 def splitIntoFibonacci(self, S: str) -> List[int]:  
 # 5:33 4/22  
  
 def backtrack(ans, s):  
 if not s and len(ans) > 2:  
 ans[:] = [int(v) for v in ans]  
 # ans = [int(v) for v in ans]  
 return True  
  
 len\_min = 1  
 if len(ans) >= 2:  
 next\_num = int(ans[-1]) + int(ans[-2])  
 len\_min = len(str(next\_num))  
  
 for i in range(len\_min, len(s) + 1):  
 tmp = int(s[:i])  
 if tmp > (1 << 31) - 1:  
 break  
 if len(ans) < 2 or tmp == next\_num:  
 if i > 1 and s[0] == '0':  
 break  
 ans.append(s[:i])  
 if backtrack(ans, s[i:]):  
 return True  
 ans.pop()  
 if len(ans) >= 2:  
 break  
 return False  
  
 ans = []  
 backtrack(ans, S)  
 return ans

1. **307. Range Sum Query – Mutable**

分块累加，各不干预。就是将原数组分为若干块，怎么分呢，这里就让每个 block 有 sqrt(n) 个数字就可以了，这个基本是让 block 的个数跟每个 blcok 中数字的个数尽可能相同的分割方法。然后我们就需要一个大小跟 block 个数相同的数组，来保存每个 block 的数字之和。在需要更新的时候，我们就先确定要更新的位置在哪个 block 里，然后只更新该 block 的和。而对于求区域和操作，我们还是要分别确定i和j分别属于哪个 block，若属于同一个 block，那么直接遍历累加即可，若属于不同的，则先从i累加到该 blcok 的末尾，然后中间横跨的那些 block 可以直接将和累加，对于j所在的 blcok，则从该 block 的开头遍历累加到j即可

class NumArray:  
 def \_\_init\_\_(self, nums: List[int]):  
 self.arr = nums  
 self.n = math.ceil(len(self.arr) / (len(self.arr)) \*\* 0.5)  
 self.block = [0] \* self.n  
  
 for i in range(len(self.arr)):  
 self.block[i // self.n] += self.arr[i]  
  
 def update(self, i: int, val: int) -> None:  
 idx = i // self.n  
 self.block[idx] += val - self.arr[i]  
 self.arr[i] = val  
  
 def sumRange(self, i: int, j: int) -> int:  
  
 sums = 0  
 lt\_block, rt\_block = i // self.n, j // self.n  
 if lt\_block == rt\_block:  
 for k in range(i, j + 1):  
 sums += self.arr[k]  
 else:  
 for k in range(i, (lt\_block + 1) \* self.n):  
 sums += self.arr[k]  
 for m\_block in range(lt\_block + 1, rt\_block):  
 sums += self.block[m\_block]  
  
 for k in range((rt\_block) \* self.n, j + 1):  
 sums += self.arr[k]  
  
 return sums

1. **79. Word Search**

class Solution:  
 def exist(self, board: List[List[str]], word: str) -> bool:  
 # 11:06 4/26/21  
  
 def helper(board, i, j, word, m, visited):  
 if m == len(word):  
 return True  
  
 if i < 0 or i == len(board) or j < 0 or j == len(board[0]):  
 return False  
 if (i, j) in visited or board[i][j] != word[m]:  
 return False  
  
 visited.add((i, j))  
  
 # dirs =[[0,-1], [0,1], [-1,0], [1, 0]]  
 dirs = {(0, -1), (0, 1), (-1, 0), (1, 0)}  
 for r, c in dirs:  
 if helper(board, i + r, j + c, word, m + 1, visited):  
 return True  
 visited.remove((i, j))  
 return False  
  
 for i in range(len(board)):  
 for j in range(len(board[0])):  
 if helper(board, i, j, word, 0, set()):  
 return True  
 return False

class Solution:  
 def exist(self, board: List[List[str]], word: str) -> bool:  
 *"""* ***:type*** *board: List[List[str]]* ***:type*** *word: str* ***:rtype****: bool  
  
 """* def helper(board, i, j, word, k, visited): # (board, 0, 0, word, 0, set())  
 if k == len(word):  
 return True  
  
 if (i, j) in visited:  
 return False  
  
 if 0 <= i < len(board) and 0 <= j < len(board[0]):  
 if board[i][j] == word[k]:  
 visited.add((i, j))  
 for dir in [(0, 1), (0, -1), (-1, 0), (1, 0)]:  
 if helper(board, i + dir[0], j + dir[1], word, k + 1, visited):  
 return True  
 visited.remove((i, j))  
  
 return False  
  
 visited = set()  
  
 for i in range(len(board)):  
 for j in range(len(board[0])):  
 if board[i][j] == word[0]:  
 if helper(board, i, j, word, 0, visited):  
 return True  
  
 return False

**4/27/2021**

1. **34. Find First and Last Position of Element in Sorted Array**

**34. Find First and Last Position of Element in Sorted Array**

Medium

5394209Add to ListShare

Given an array of integers nums sorted in ascending order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

**Follow up:** Could you write an algorithm with O(log n) runtime complexity?

**Example 1:**

**Input:** nums = [5,7,7,8,8,10], target = 8

**Output:** [3,4]

**Example 2:**

**Input:** nums = [5,7,7,8,8,10], target = 6

**Output:** [-1,-1]

**Example 3:**

**Input:** nums = [], target = 0

**Output:** [-1,-1]

**Constraints:**

* 0 <= nums.length <= 105
* -109 <= nums[i] <= 109
* nums is a non-decreasing array.
* -109 <= target <= 109

class Solution:  
 def searchRange(self, nums: List[int], target: int) -> List[int]:  
 # 9:56 4/27/21  
  
 def find\_upper(nums, lt, rt, target):  
 while lt <= rt:  
 mid = lt + (rt - lt) // 2  
 if nums[mid] == target:  
 lt = mid + 1  
 else:  
 rt = mid - 1  
 return lt - 1  
  
 def find\_lower(nums, lt, rt, target):  
 while lt < rt:  
 mid = lt + (rt - lt) // 2  
 if nums[mid] == target:  
 rt = mid  
 else:  
 lt = mid + 1  
 return rt  
  
 n = len(nums)  
 if not nums:  
 return [-1, -1]  
 lt, rt = 0, n - 1  
  
 while lt <= rt:  
 mid = lt + (rt - lt) // 2  
 if nums[mid] > target:  
 rt = mid - 1  
 elif nums[mid] < target:  
 lt = mid + 1  
 else:  
 upper = find\_upper(nums, mid + 1, rt, target)  
 lower = find\_lower(nums, lt, mid, target)  
 return [lower, upper]  
 return [-1, -1]

class Solution:  
 def searchRange(self, nums: List[int], target: int) -> List[int]:  
 # 2/27/20, 10:17  
 # 1:30 7/5/2020  
 n = len(nums)  
 lt, rt = 0, n  
 start, end = [-1, -1]  
  
 while lt < rt:  
 mid = lt + (rt - lt) // 2  
 if nums[mid] == target:  
 lt\_tmp = lt  
 rt\_tmp = rt  
 lt = mid + 1  
 while lt < rt:  
 mid = lt + (rt - lt) // 2  
 if nums[mid] == target:  
 lt = mid + 1  
 elif nums[mid] > target:  
 rt = mid  
 end = lt - 1  
  
 lt, rt = lt\_tmp, mid  
 while lt < rt:  
 mid = lt + (rt - lt) // 2  
 if nums[mid] == target:  
 rt = mid  
 elif nums[mid] < target:  
 lt = mid + 1  
 start = rt  
 break  
 elif nums[mid] > target:  
 rt = mid  
 else: # nums[mid] < target  
 lt = mid + 1  
  
 return [start, end]

class Solution:  
 def searchRange(self, nums: List[int], target: int) -> List[int]:  
 # 2/27/20, 10:17  
 n = len(nums)  
  
 def find\_left(nums, target):  
 n = len(nums)  
 lt, rt = 0, n  
 while lt < rt:  
 mid = lt + (rt - lt) // 2  
 if nums[mid] < target:  
 lt = mid + 1  
 else:  
 rt = mid  
 if rt == n or nums[rt] != target:  
 return -1  
 return rt  
  
 def find\_right(nums, target):  
 n = len(nums)  
 lt, rt = 0, n  
 while lt < rt:  
 mid = lt + (rt - lt) // 2  
 if nums[mid] <= target:  
 lt = mid + 1  
 else:  
 rt = mid  
 if rt < n and nums[rt - 1] != target:  
 return -1  
 return rt - 1  
  
 r\_lt = find\_left(nums, target)  
 if r\_lt != -1:  
 r\_rt = find\_right(nums, target)  
 return [r\_lt, r\_rt]  
 return [-1, -1]

1. **198. House Robber**

Medium

6916195Add to ListShare

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array nums representing the amount of money of each house, return *the maximum amount of money you can rob tonight****without alerting the police***.

**Example 1:**

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

**Output:** 4

**Explanation:** Rob house 1 (money = 1) and then rob house 3 (money = 3).

Total amount you can rob = 1 + 3 = 4.

**Example 2:**

**Input:** nums = [2,7,9,3,1]

**Output:** 12

**Explanation:** Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).

Total amount you can rob = 2 + 9 + 1 = 12.

**Constraints:**

* 1 <= nums.length <= 100
* 0 <= nums[i] <= 400

class Solution:  
 def rob(self, nums: List[int]) -> int:  
 n = len(nums)  
 dp = [0 ] \*n  
  
 for i in range(n):  
 if i == 0 or i == 1:  
 dp[i] = nums[i]  
 else:  
 if i == 2:  
 dp[i] = dp[ i -2] + nums[i]  
 else:  
 dp[i] = nums[i] + max(dp[ i -2], dp[ i -3])  
  
 return max(dp[ n -1], dp[ n -2]) if n > 1 else dp[0]

dp[i] stands for largest value can be robbed from houses[:i+1]. It's max(dp[i-1], dp[i-2]+houses[i]) since robber can't rob house[i] if it robbed houses[i-1].  
We only use dp[i-1] and dp[i-2] to update dp[i] , so we can replace dp array by rolling two variables prev and curr.

class Solution:  
 def rob(self, nums: List[int]) -> int:  
  
 prev = curr = 0  
 for x in nums:  
 curr, prev = max(curr, prev+x), curr  
 return curr

class Solution:  
 def rob(self, nums: List[int]) -> int:  
  
 n = len(nums)  
 if n == 1:  
 return nums[0]  
 # elif n == 2:  
 # return max(nums)  
 else:  
 dp = [0] \* n  
 dp[0], dp[1] = nums[0], max(nums[0], nums[1])  
 for i in range(2, n):  
 dp[i] = max(dp[i - 1], dp[i - 2] + nums[i])  
 return dp[n - 1]

**213. House Robber II**

class Solution:  
 def rob(self, nums: List[int]) -> int:  
 # House Robber II , 4:29 --> 9/5/20  
  
 def helper(nums):  
 n = len(nums)  
 if n <= 2: return max(nums)  
 dp = [0] \* n  
 dp[0], dp[1] = nums[0], max(nums[:2])  
 for i in range(2, n):  
 dp[i] = max(dp[i - 1], dp[i - 2] + nums[i])  
 return dp[n - 1]  
  
 n = len(nums)  
 if n < 2:  
 return nums[n - 1]  
 return max(helper(nums[:n - 1]), helper(nums[1:n]))

class Solution:  
 def rob(self, nums: List[int]) -> int:  
 # House Robber II , 4:29 --> 9/5/20  
  
 def helper(nums):  
 n = len(nums)  
 # if n <= 2: return max(nums)  
 dp = [0] \* n  
 dp[0], dp[1] = nums[0], max(nums[:2])  
 for i in range(2, n):  
 dp[i] = max(dp[i - 1], dp[i - 2] + nums[i])  
 return dp[n - 1]  
  
 n = len(nums)  
 if n <= 2:  
 return max(nums)  
 return max(helper(nums[:n - 1]), helper(nums[1:n]))

class Solution:  
 def rob(self, nums: List[int]) -> int:  
 # House Robber II , 4:29 --> 9/5/20  
  
 def basicRob(start, end):  
 rob, nonRob = nums[start], 0  
 for i in range(start + 1, end):  
 preRob, preno = rob, nonRob  
 rob = preno + nums[i]  
 nonRob = max(preRob, preno)  
 return max(rob, nonRob)  
  
 n = len(nums)  
 if n <= 1:  
 return sum(nums)  
 else:  
 return max(basicRob(0, len(nums) - 1), basicRob(1, len(nums)))

1. **322. Coin Change**

class Solution:  
 def coinChange(self, coins: List[int], amount: int) -> int:  
 # 11:31 4/28/21  
  
 n = len(coins)  
 coins.sort()  
  
 dp = [-1] \* (amount + 1)  
 dp[0] = 0  
  
 for i in range(1, amount + 1):  
 for j in range(n):  
 if coins[j] <= i and dp[i - coins[j]] != -1:  
 if dp[i] == -1:  
 dp[i] = dp[i - coins[j]] + 1  
 else:  
 dp[i] = min(dp[i - coins[j]] + 1, dp[i])  
  
 return dp[amount]

class Solution:  
 def coinChange(self, coins: List[int], amount: int) -> int:  
 # 11:31 4/28/21  
  
 n = len(coins)  
 coins.sort()  
  
 dp = [-1] \* (amount + 1)  
 dp[0] = 0  
  
 for i in range(1, amount + 1):  
 for j in range(n):  
 if coins[j] > i:  
 break  
 elif dp[i - coins[j]] == -1:  
 continue  
 else:  
 if dp[i] == -1:  
 dp[i] = dp[i - coins[j]] + 1  
 else:  
 dp[i] = min(dp[i - coins[j]] + 1, dp[i])  
  
 return dp[amount]

class Solution:  
 def coinChange(self, coins: List[int], amount: int) -> int:  
 dp = [-1] \* (amount+1)  
 dp[0] = 0  
 for i in range(1, amount+1):  
 for j in coins:  
 if i - j >= 0 and dp[i-j] != -1:  
 dp[i] = dp[i-j] + 1 if dp[i] == -1 else min(dp[i], dp[i-j]+1)  
 return dp[amount]

class Solution:  
 def coinChange(self, coins: List[int], amount: int) -> int:  
 # 11:31 4/28/21  
 # BFS  
 q = collections.deque([0])  
 step = 0  
 visited = set()  
  
 while q:  
 q\_size = len(q)  
 for \_ in range(q\_size):  
 total = q.popleft()  
 if total == amount:  
 return step  
 elif total > amount or total in visited:  
 continue  
 else:  
 visited.add(total)  
 for v in coins:  
 q.append(v + total)  
 step += 1  
  
 return -1

class Solution:  
  
 def \_\_init\_\_(self):  
 self.mem = {0: 0}  
  
 def coinChange(self, coins: List[int], amount: int) -> int:  
 # 11:31 4/28/21  
 # DFS  
 coins.sort()  
 minCoins = self.getMinCoins(coins, amount)  
  
 if minCoins == float('inf'):  
 return -1  
 return minCoins  
  
 def getMinCoins(self, coins, amount):  
 if amount in self.mem:  
 return self.mem[amount]  
  
 minCoins = float('inf')  
  
 for c in coins:  
 if amount - c < 0: break  
 numCoins = self.getMinCoins(coins, amount - c) + 1  
 minCoins = min(numCoins, minCoins)  
  
 self.mem[amount] = minCoins  
  
 return minCoins

1. **840. Magic Squares In Grid**

class Solution:  
  
 def isMagicSquare(self, grid):  
 *'''  
 Check whether the given grid is a magic square  
 '''* # Check the elements  
 flat = [num for row in grid for num in row]  
 if sorted(flat) != [1, 2, 3, 4, 5, 6, 7, 8, 9]:  
 return False  
  
 # Check the row, column and diagnal sums  
 row\_sums = [sum(row) for row in grid]  
 col\_sums = [sum([row[i] for row in grid]) for i in range(3)]  
 diag\_sums = [sum([grid[i][i] for i in range(3)]), (grid[0][2] + grid[1][1] + grid[2][0])]  
 row\_sums.extend(col\_sums)  
 row\_sums.extend(diag\_sums)  
 return len(set(row\_sums)) == 1  
  
 def numMagicSquaresInside(self, grid: List[List[int]]) -> int:  
 # 9:37 --> 9:52 --> 10:31 9/14/20  
 cnt = 0  
 # Construct the 3x3 square  
 for i in range(len(grid) - 2):  
 for j in range(len(grid) - 2):  
 temp\_grid = [grid[i + k][j:j + 3] for k in range(3)]  
 if self.isMagicSquare(temp\_grid):  
 cnt += 1  
  
 return cnt

**519. Random Flip Matrix**

class Solution:  
  
 # def \_\_init\_\_(self, n\_rows: int, n\_cols: int):  
  
 # def flip(self) -> List[int]:  
  
 # def reset(self) -> None:  
  
 # Your Solution object will be instantiated and called as such:  
 # obj = Solution(n\_rows, n\_cols)  
 # param\_1 = obj.flip()  
 # obj.reset()  
  
 # 11:13 4/29/21  
  
 def \_\_init\_\_(self, n\_rows: int, n\_cols: int):  
 self.n\_rows = n\_rows  
 self.n\_cols = n\_cols  
 self.start = 0  
 self.end = self.n\_rows \* self.n\_cols  
 self.pos\_dict = collections.defaultdict()  
  
 def flip(self) -> List[int]:  
 # generate index  
 rand\_idx = random.randrange(self.start, self.end)

#rand\_idx = random.randint(self.start, self.end-1)  
  
 # check if we have already put something at this index  
 # The get() method takes maximum of two parameters:   
 # key - key to be searched in the dictionary  
 # value (optional) - Value to be returned if the key is not found. The default value is None.  
 res = self.pos\_dict.get(rand\_idx, rand\_idx)  
  
 # swap - put total at index that we generated  
 self.pos\_dict[rand\_idx] = self.pos\_dict.get(self.start, self.start)  
  
 # decrease total number of values

class Solution:  
 # 9/9/20 9:30  
 def \_\_init\_\_(self, n\_rows: int, n\_cols: int):  
 self.c = n\_cols  
 self.end = n\_rows \* n\_cols - 1  
 self.d = {}  
 self.start = 0  
  
 def flip(self) -> List[int]:  
 rand = random.randint(self.start, self.end)  
 res = self.d.get(rand, rand)  
 self.d[rand] = self.d.get(self.start, self.start)  
 self.start += 1  
 return divmod(res, self.c)  
  
 def reset(self) -> None:  
 self.d.clear()  
 self.start = 0

1. **373. Find K Pairs with Smallest Sums**

class Solution:  
 def kSmallestPairs(self, nums1: List[int], nums2: List[int], k: int) -> List[List[int]]:  
 # Find K pairs with smallest sums 11:48 9/6/20  
  
 # 9:08 429/21  
  
 m, n = len(nums1), len(nums2)  
 res = []  
  
 for i in range(m):  
 for j in range(n):  
 res.append([nums1[i], nums2[j]])  
  
 res.sort(key=lambda x: x[0] + x[1])  
 return res if k > len(res) else res[:k]

#return nums\_new[:k]

**BFS + heapq**

Obviously, **nums1[0]+nums2[0**] is the smallest.  
The next smallest could be **nums1[1]+nums2[0]** or **nums1[0]+nums[2]**, so we add both to the heap.  
Heap pop gives us the current smallest with **nums1[left] and nums2[right]**  
And then heappush **nums1[left+1]+nums2[right]** and **nums1[left]+nums2[right+1]**, if left+1 and right+1 are valid.

Another important observation is that **nums1[left]+nums2[right] may be pushed twice**.  
Because both nums1[left-1]+nums2[right] and nums1[left]+nums2[right-1] will try to heappush nums1[left]+nums2[right]  
So we need a set to marked used left, right pairs.  
The size of heap is increasing, because every heappop tries to heappush two numbers.  
So k heappop will left a heap size of k at most.  
Time: KlogK  
Space: O(K)

class Solution:  
 def kSmallestPairs(self, nums1: List[int], nums2: List[int], k: int) -> List[List[int]]:  
 # Find K pairs with smallest sums 11:48 9/6/20  
  
 # 9:08 429/21  
 import heapq  
 res = []  
 heap = []  
 visited = set()  
 heapq.heappush(heap, (nums1[0] + nums2[0], 0, 0))  
 m, n = len(nums1), len(nums2)  
  
 while k > 0 and heap:  
  
 (total, i, j) = heapq.heappop(heap)  
 if (i, j) in visited:  
 continue  
 visited.add((i, j))  
 res.append([nums1[i], nums2[j]])  
 if i + 1 < m:  
 heapq.heappush(heap, (nums1[i + 1] + nums2[j], i + 1, j))  
 if j + 1 < n:  
 heapq.heappush(heap, (nums1[i] + nums2[j + 1], i, j + 1))  
 k -= 1  
 return res

import heapq  
class Solution:  
 def kSmallestPairs(self, nums1: List[int], nums2: List[int], k: int) -> List[List[int]]:  
 if not nums1 or not nums2:  
 return []  
 res = []  
 len\_nums1, len\_nums2 = len(nums1), len(nums2)  
 my\_hp = [[nums1[0]+nums2[0], 0, 0]] # the index of both nums1 and nums2 are also included in heap item.  
 seen = {(0,0)}  
 while k > 0 and my\_hp:  
 \_, left, right = heappop(my\_hp)  
 res.append([nums1[left], nums2[right]])  
 k -= 1  
 if left < len\_nums1-1 and (left+1, right) not in seen:  
 heappush(my\_hp, [nums1[left+1]+nums2[right], left+1, right])  
 seen.add((left+1, right))  
 if right < len\_nums2-1 and (left, right+1) not in seen:  
 heappush(my\_hp, [nums1[left]+nums2[right+1], left, right+1])  
 seen.add((left, right+1))  
 return res

**4/30/21**

1. **393. UTF-8 Validation**

class Solution:  
 def validUtf8(self, data: List[int]) -> bool:  
 # 9:46 --> 10: 154/30/21  
  
 def int\_to\_binary\_str(num):  
 res = ""  
 while num > 0:  
 res = str(num % 2) + res  
 num //= 2  
  
 if len(res) < 8:  
 to\_add = '0' \* (8 - len(res))  
 res = to\_add + res  
 elif len(res) > 8:  
 res = res[len(res) - 8:]  
 return res  
  
 # def int\_to\_binary\_str(num):  
 # res = ""  
 # while num > 0:  
 # tmp = num % 2  
 # num = num // 2  
 # res = str(tmp) + res  
  
 # n = 8 - len(res)  
 # if n > 0:  
 # res = '0'\*n + res  
  
 # return res  
  
 def is\_multi\_byte(s):  
 return s[0:2] == "10"  
  
 # parse  
 n = len(data)  
 res = []  
 for num in data:  
 res.append(int\_to\_binary\_str(num))  
 i = 0  
 while i < n:  
 code = res[i]  
 if code[0] == '0':  
 i += 1  
  
 elif code[0] == '1':  
 j = 0  
 while j < 8 and code[j] == '1':  
 j += 1  
 if j == 1 or j > 4 or code[j] != '0': # forget j == 1  
 return False  
  
 cnt = j - 1  
 while cnt > 0:  
 i += 1  
 if i >= n: return False # forget i>=n  
 if not is\_multi\_byte(res[i]):  
 return False  
 cnt -= 1  
 i += 1  
  
 return True

class Solution:  
 def validUtf8(self, data: List[int]) -> bool:  
 # 9:46 --> 10: 154/30/21  
  
 count = 0  
 for n in data:  
 if count == 0:  
 if (n >> 7) == 0:  
 count = 0  
 elif (n >> 5) == 0x06:  
 count = 1  
 elif (n >> 4) == 0x0e:  
 count = 2  
 elif (n >> 3) == 0x1e:  
 count = 3  
 else:  
 return False  
 else:  
 if (n >> 6) != 0x02:  
 return False  
 count -= 1  
 return count == 0

1. **332. Reconstruct Itinerary**

class Solution:  
 def findItinerary(self, tickets: List[List[str]]) -> List[str]:  
 # 9:21 5/1/21  
 def backtracking(dept, route, flight\_map, visited, res, n):  
 **if len(route) == n:** for v in route:  
 res.append(v)

# res = route NOT working return True  
  
 for i, next\_stop in enumerate(flight\_map[dept]):  
 if not visited[dept][i]:  
 visited[dept][i] = True  
 if backtracking(next\_stop, route + [next\_stop], flight\_map, visited, res, n):  
 return True  
 visited[dept][i] = False  
  
 return False  
  
 from collections import defaultdict  
  
 flight\_map = defaultdict(list)  
  
 for ticket in tickets:  
 flight\_map[ticket[0]].append(ticket[1])  
  
 visited = {}  
 for dept in flight\_map:  
 flight\_map[dept].sort()  
 visited[dept] = [False] \* len(flight\_map[dept])  
  
 res = []  
 route = ['JFK']  
 n = len(tickets) + 1  
  
 backtracking("JFK", route, flight\_map, visited, res, n, )  
 return res

class Solution:  
  
 def build\_graph(self, tickets):  
 g, ticket\_id = defaultdict(list), 0  
 for start, end in tickets:  
 g[start].append((end, ticket\_id))  
 ticket\_id += 1  
 for k in g:  
 g[k].sort(key=lambda x: x[0])  
  
 return g  
  
 def tour(self, g, itenary, tickets\_usd, N):  
 if len(itenary) == N:  
 return True  
 s = itenary[-1]  
 if s in g:  
 for nbr, ticket\_id in g[s]:  
 if ticket\_id not in tickets\_usd:  
 tickets\_usd.add(ticket\_id)  
 itenary.append(nbr)  
 if self.tour(g, itenary, tickets\_usd, N):  
 return True  
 itenary.pop()  
 tickets\_usd.remove(ticket\_id)  
  
 return False  
  
 def findItinerary(self, tickets: List[List[str]]) -> List[str]:  
 # 6:50--> 7:03 9/7/20 Reconstruct Itinery  
 g = self.build\_graph(tickets)  
 tickets\_usd, itenary = set(), ["JFK"]  
 self.tour(g, itenary, tickets\_usd, len(tickets) + 1)  
  
 return itenary

**5/3/21**

**74. Search a 2D Matrix**

Medium

3201195Add to ListShare

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

* Integers in each row are sorted from left to right.
* The first integer of each row is greater than the last integer of the previous row.

**Example 1:**



**Input:** matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

**Output:** true

**Example 2:**



**Input:** matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13

**Output:** false

**Constraints:**

* m == matrix.length
* n == matrix[i].length
* 1 <= m, n <= 100
* -104 <= matrix[i][j], target <= 104

Accepted

444,758

Submissions

1,160,850

class Solution:  
 def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:  
 # 3:02 5/3/21  
  
 if not matrix or not matrix[0]:  
 return False  
  
 m, n = len(matrix), len(matrix[0])  
  
 top\_row, bottom\_row = 0, m  
  
 while top\_row < bottom\_row:  
 mid = top\_row + (bottom\_row - top\_row) // 2  
 if matrix[mid][-1] == target:  
 return True  
 elif matrix[mid][-1] < target:  
 top\_row = mid + 1  
 else:  
 bottom\_row = mid  
  
 if top\_row == m:  
 return False  
  
 left, end = 0, n  
 while left < end:  
 mid = left + (end - left) // 2  
 if matrix[top\_row][mid] == target:  
 return True  
 elif matrix[top\_row][mid] < target:  
 left = mid + 1  
 else:  
 end = mid  
  
 return False

class Solution:  
 def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:  
 # 3/4/20 7:55pm --> 8:04  
 if not matrix or not matrix[0]:  
 return False  
  
 m, n = len(matrix), len(matrix[0])  
 lt, rt = 0, m \* n  
  
 while lt < rt:  
 mid = lt + (rt - lt) // 2  
 mid\_val = matrix[mid // n][mid % n]  
 if mid\_val == target:  
 return True  
 elif mid\_val < target:  
 lt = mid + 1  
 else:  
 rt = mid  
  
 return False

class Solution:  
 def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:  
 # 3/4/20 7:55pm  
  
 if not matrix or not matrix[0]:  
 return False  
 m, n = len(matrix), len(matrix[0])  
  
 top, btm = 0, m  
 while top < btm:  
 mid = top + (btm - top) // 2  
 if matrix[mid][0] == target or matrix[mid][-1] == target:  
 return True  
 elif matrix[mid][-1] < target:  
 top = mid + 1  
 else:  
 btm = mid  
 if btm == m:  
 return False  
  
 lt, rt = 0, n  
 while lt < rt:  
 mid = lt + (rt - lt) // 2  
 if matrix[top][mid] == target:  
 return True  
 elif matrix[top][mid] < target:  
 lt = mid + 1  
 else:  
 rt = mid  
 return False

**5/4/21**

1. **473. Matchsticks to Square**

class Solution:  
 def makesquare(self, nums: List[int]) -> bool:  
 # 11:35 9/11/20  
 # 4:28 5/4/21  
  
 def dfs(nums, i, arr):  
 if i == len(nums):  
 return True  
 for j in range(4):  
 if nums[i] <= arr[j]:  
 arr[j] -= nums[i]  
 if dfs(nums, i + 1, arr):  
 return True  
 arr[j] += nums[i]  
 return False  
  
 n = len(nums)  
 len\_total = sum(nums)  
 if len\_total % 4: return False  
 nums.sort(reverse=True)  
 arr = [len\_total // 4] \* 4  
  
 return dfs(nums, 0, arr)

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

class Solution:  
 def findNumberOfLIS(self, nums: List[int]) -> int:  
 # 4:52 5/4/21  
  
 n = len(nums)  
  
 dp = [[1, 1] for \_ in range(n)]  
 max\_pair = [1, 1] # [length, #]  
  
 for i in range(1, n):  
 for j in range(i):  
 if nums[i] > nums[j]:  
 new\_len = dp[j][0] + 1  
 if new\_len == dp[i][0]:  
 dp[i][1] += dp[j][1]  
 elif new\_len > dp[i][0]:  
 dp[i][0] = new\_len  
 dp[i][1] = dp[j][1]  
  
 if max\_pair[0] == dp[i][0]:  
 max\_pair[1] += dp[i][1]  
 elif max\_pair[0] < dp[i][0]:  
 # max\_pair = dp[i] ; WRONG  
 max\_pair[0], max\_pair[1] = dp[i][0], dp[i][1]  
  
 return max\_pair[1]

**5/5/21**

1. **845. Longest Mountain in Array**

class Solution:  
 def longestMountain(self, arr: List[int]) -> int:  
 # 11:44 --> 12:04 --> 12:09 --> 12:18 -->5/5/21  
 # [875,884,239,731,723,685]  
 n = len(arr)  
 if n < 3:  
 return 0  
 stk = []  
 up = True  
 max\_len = 0  
  
 for v in arr:  
 if not stk:  
 stk.append(v)  
 up = True  
 else:  
 if up == True:  
 if v > stk[-1]:  
 stk.append(v)  
 elif v < stk[-1]:  
 if len(stk) == 1:  
 stk[-1] = v  
 else:  
 stk.append(v)  
 up = False  
 else: # v == stk[-1]  
 stk.clear()  
 stk.append(v)  
 else: # up == False  
 if v >= stk[-1]:  
 max\_len = max(max\_len, len(stk))  
 tmp = stk[-1]  
 stk.clear()  
 if v > tmp:  
 stk.append(tmp)  
 stk.append(v)  
 up = True  
 else:  
 stk.append(v)  
  
 if up == False:  
 max\_len = max(max\_len, len(stk))  
 return max\_len

class Solution:  
 def longestMountain(self, arr: List[int]) -> int:  
 # 11:44 --> 12:04 --> 12:09 --> 12:18 -->5/5/21  
 # [875,884,239,731,723,685]  
 n = len(arr)  
 if n < 3:  
 return 0  
 cnt, cur = 0, 0  
 up = True  
 max\_len = 0  
  
 for v in arr:  
 if cnt == 0:  
 cnt, cur = 1, v  
 up = True  
 else:  
 if up == True:  
 if v > cur:  
 cur = v  
 cnt += 1  
 elif v < cur:  
 if cnt == 1:  
 cur = v  
 else:  
 cur = v  
 cnt += 1  
 up = False  
 else: # v == stk[-1]  
 cnt = 1  
 cur = v  
 else: # up == False  
 if v >= cur:  
 max\_len = max(max\_len, cnt)  
 if v > cur:  
 cnt = 2  
 cur = v  
 elif v == cur:  
 cnt = 1  
  
 up = True  
 else:  
 cur = v  
 cnt += 1  
  
 if up == False:  
 max\_len = max(max\_len, cnt)  
 return max\_len

**In this problem, we take one forward pass to count up hill length (to every point).  
We take another backward pass to count down hill length (from every point).  
Finally a pass to find max(up[i] + down[i] + 1) where up[i] and down[i] should be positives**.

class Solution:  
 def longestMountain(self, arr: List[int]) -> int:  
 # 11:44 --> 12:04 --> 12:09 --> 12:18 -->5/5/21  
 # [875,884,239,731,723,685]  
  
 n = len(arr)  
 up, down = [1] \* n, [1] \* n  
  
 for i in range(1, n):  
 if arr[i] > arr[i - 1]:  
 up[i] = up[i - 1] + 1  
 for i in range(n - 2, -1, -1):  
 if arr[i] > arr[i + 1]:  
 down[i] = down[i + 1] + 1  
  
 max\_len = 0  
 # for i in range(n):  
 # if up[i] > 1 and down[i] > 1:  
 # max\_len = max(max\_len, up[i]+down[i]-1)  
  
 for up\_v, down\_v in zip(up, down):  
 if up\_v > 1 and down\_v > 1:  
 max\_len = max(max\_len, up\_v + down\_v - 1)  
  
 return max\_len

1. **801. Minimum Swaps To Make Sequences Increasing**
2. A[i - 1] < A[i] && B[i - 1] < B[i].  
   In this case, if we want to keep A and B increasing before the index i, can only have two choices.  
   -> 1.1 don't swap at (i-1) and don't swap at i, we can get not\_swap[i] = not\_swap[i-1]  
   -> 1.2 swap at (i-1) and swap at i, we can get swap[i] = swap[i-1]+1  
   if swap at (i-1) and do not swap at i, we can not guarantee A and B increasing.
3. A[i-1] < B[i] && B[i-1] < A[i]  
   In this case, if we want to keep A and B increasing before the index i, can only have two choices.  
   -> 2.1 swap at (i-1) and do not swap at i, we can get notswap[i] = Math.min(swap[i-1], notswap[i] )  
   -> 2.2 do not swap at (i-1) and swap at i, we can get swap[i]=Math.min(notswap[i-1]+1, swap[i])

We can define two array to keep track of how many swaps we made at current index i.

To be more specific, wefine swap[i]/keep[i] as the miminum cost to Make Sequences Increasing with/without swap i-th element.

Thus there are two cases we need to consider at current index i :

1. Case 1. If A[i]>A[i-1] and B[i] > B[i-1]:  
   Case 1.1: No need to swap anything,Thus keep[i]=keep[i-1]  
   Case 1.2: We need to swap both (A[i-1],B[i-1]) and (A[i],B[i]) in order to make both sequences increasing. Thus swap[i]=swap[i-1]+1.
2. Case 2. if A[i]>B[i-1] and B[i] > A[i-1]:  
   Case 2.1: Only swap (A[i],B[i]), thus swap[i]=min(swap[i],keep[i-1]+1)  
   Case 2.2: Only swap (A[i-1],B[i-1]), thus keep[i]=min(keep[i],swap[i-1]).

class Solution:  
 def minSwap(self, A: List[int], B: List[int]) -> int:  
  
 n = len(A)  
  
 # In this problem, we have two states that we need to keep track at element i;   
 # what happens if we swap or do not swap at i?  
  
 # min number of swaps to get ascending order if we SWAP at i   
 swap = [0] \* n  
  
 # min number of swaps to get ascending order if we do NOT SWAP at i  
 noswap = [0] \* n  
  
 # base case: swapping 0th element  
 swap[0] = 1  
  
 for i in range(1, n):  
 strictly\_increasing = A[i] > A[i - 1] and B[i] > B[i - 1]  
 strictly\_xincreasing = A[i] > B[i - 1] and B[i] > A[i - 1]  
  
 # we can swap here, but we also do not have too  
 if strictly\_increasing and strictly\_xincreasing:  
  
 # if we decide to swap at i, we can consider the state of either swapping or not swapping i-1, since doing either  
 # will not break out strictly increasing for either array rule  
 swap[i] = min(noswap[i - 1], swap[i - 1]) + 1  
  
 # we chose not to swap at i, so we can consider states if we swap at i - 1 or not swap, since again we will not break the strictly increasing rule  
 noswap[i] = min(noswap[i - 1], swap[i - 1])  
  
  
 # we can not swap at i, since we know that A[i] is not greater than B[i-1], or that B[i] is not greater than A[i-1]  
 elif strictly\_increasing:  
  
 # if we swap at i, then to keep increasing condition, we must also swap at i-1, we are essentially just swapping both i and i-1 and get the same thing   
 swap[i] = swap[i - 1] + 1  
  
 # if we do not swap at i, then we must also chose to not swap at i-1 to keep the problem conditions true  
 noswap[i] = noswap[i - 1]  
  
  
 # we must swap, since A[i] is not greater than A[i-1], or B[i] is not greater than B[i-1]  
 # but we know that A[i] > B[i-1] and that B[i] > A[i-1]  
 elif strictly\_xincreasing:  
  
 # if we swap at i, then we must have not swapped at i-1  
 swap[i] = noswap[i - 1] + 1  
  
 # if we do not swap at i, then we must have swapped at i-1 for the problem conditions to be true  
 noswap[i] = swap[i - 1]  
  
 # take min of both state paths to see who holds the better result  
 return min(noswap[n - 1], swap[n - 1])

Firstly, at each stage , we need to store the value of count if we swap A[i] and B[i] and if we don't swap A[i] and B[i].  
If they are increasing without a swap, then we don't need to do anything when we don't swap. We take the previous value stored for not swapping and store in current index. For swap, that means the swap which occurred previously was a faulty one , so we need to rectify that so swap[i-1] +1  
If they are increasing with a swap, that means B[i-1] < A[i] and A[i-1] < B[i]. This means if we want to swap A[i] and B[i], we can either keep[i-1] + 1 or swap[i-1] +1 (we can swap both or keep one and swap one . Both will work). We take the minimum of these two values. For noswap array, we take the minimum of swap[i-1] and noswap[i].  
In the end, the minimum of both final values (noswap and swap) are taken as the final result.

class Solution:  
 def minSwap(self, A: List[int], B: List[int]) -> int:  
 n = len(A)  
 noswap = [ float('inf')] \* n  
 swap = [float('inf')] \* n  
 noswap[0] = 0  
 swap[0]=1  
 for i in range(1 , n):  
 incr\_without\_swap = A[i] >A[i-1] and B[i] >B[i-1]  
 incr\_with\_swap = A[i]>B[i-1] and B[i] > A[i-1]  
 if incr\_without\_swap:  
 #you can keep both values or swap both values  
 noswap[i]=noswap[i-1]  
 swap[i]= swap[i-1]+1  
 if incr\_with\_swap:  
 swap[i] = min(noswap[i-1] +1,swap[i])  
 noswap[i]= min(swap[i-1],noswap[i])  
 return min(noswap[-1],swap[-1])

many excellent programmers post their solutions which conclude the problem into two condition.  
Here are my thoughts to make it a little easier to understand as take three conditions into account.  
swap[i]: the least swap number in case we swap the i-th  
not\_swap[i]: the least swap number in case we not swap the i-th.

The first situation:  
A[i-1] < A[i] and B[i-1] < B[i] and A[i-1] < B[i] and B[i-1] < A[i]  
in this case, we could either swap or not swap the i-th, it just OK.  
obviously, the swap[i] = min(swap[i-1],not\_swap[i-1]) + 1  
and the not\_swap[i] = min(swap[i-1],not\_swap[i-1])

Second condition:  
A[i - 1] < A[i] and B[i - 1] < B[i] and (B[i-1]>=A[i] or A[i-1]>=B[i])  
in this case, if we swap the (i-1)-th, we have to swap the i-th, to make A and B sorted, vice versa.  
so that,  
not\_swap[i] = not\_swap[i-1]  
swap[i] = swap[i-1]+1  
Third condition:  
in this case, if we swap the (i-1)-th, we could not swap the i-th, if we not swap the (i-1)-th, we have to swap the i-th  
swap[i] = not\_swap[i-1] + 1  
not\_swap[i] = swap[i-1]

class Solution:  
 def minSwap(self, A, B):  
 N = len(A)  
 not\_swap, swap = [N] \* N, [N] \* N  
 not\_swap[0], swap[0] = 0, 1  
 for i in range(1, N):  
 if A[i - 1] < A[i] and B[i - 1] < B[i] and A[i - 1] < B[i] and B[i - 1] <A[i]:  
 not\_swap[i] = min(swap[i-1],not\_swap[i-1])  
 swap[i] = not\_swap[i] + 1  
 elif A[i - 1] < A[i] and B[i - 1] < B[i] and (B[i-1]>=A[i] or A[i-1]>=B[i]):  
 not\_swap[i] = not\_swap[i-1]  
 swap[i] = swap[i-1]+1  
 else:  
 swap[i] = not\_swap[i-1] + 1  
 not\_swap[i] = swap[i-1]  
 return min(swap[-1], not\_swap[-1])

**5/7/21**

1. **316. Remove Duplicate Letters**

class Solution:  
 def removeDuplicateLetters(self, s: str) -> str:  
 # 4:31 5/7/21  
 n = len(s)  
 sk = []  
 from collections import Counter  
 cnter = Counter(s)  
 i = 0  
 while i < n:  
 if not sk:  
 sk.append(s[i])  
 cnter[s[i]] -= 1  
 **elif s[i] in sk**:  
 cnter[s[i]] -= 1  
 elif cnter[sk[-1]] == 0:  
 sk.append(s[i])  
 cnter[s[i]] -= 1  
 else:  
 if s[i] < sk[-1]:  
 # cnter[sk[-1]] -= 1  
 sk.pop()  
 i -= 1  
 else:  
 sk.append(s[i])  
 cnter[s[i]] -= 1  
 i += 1  
  
 return "".join(sk)

class Solution:  
 def removeDuplicateLetters(self, s: str) -> str:  
  
 # 12/19/2019, 9:33 --> 10:28 --> 11:04 --> 11:18  
 # 12/25/2019  
 letter\_dict = collections.Counter(s)  
 # letter\_set = set()  
 stack = []  
  
 # for letter in s:  
 # if letter in letter\_dict:  
 # letter\_dict[letter] += 1  
 # else:  
 # letter\_dict[letter] = 1  
 i = 0  
 while i < len(s):  
 if not stack:  
 stack.append(s[i]) ## error 1: use append not push  
 letter\_dict[s[i]] -= 1  
 else:  
 top = stack[-1]  
 if s[i] in stack:  
 letter\_dict[s[i]] -= 1  
 else:  
 if s[i] < top:  
 if letter\_dict[top] > 0:  
 # letter\_dict[top] -= 1  
 stack.pop()  
 i -= 1  
 else:  
 stack.append(s[i])  
 letter\_dict[s[i]] -= 1  
 else: # s[i] > top  
 stack.append(s[i])  
 letter\_dict[s[i]] -= 1  
 i += 1  
  
 return "".join(stack)

**5/8/21**

1. **649. Dota2 Senate**

**Key here: once banned, the banned senator is completely out of the game and has voting rights for the rest of all rounds.**

import collections  
def predictPartyVictory(senate):  
 # 5:29 9/28/20  
  
 r, d = collections.deque(), collections.deque()  
 for i, s in enumerate(senate):  
 if s == "R":  
 r.append(i)  
 else:  
 d.append(i)  
  
 while r and d:  
 if r[0] < d[0]:  
 d.popleft()  
 if d:  
 tmp = max(r[-1], d[-1]) + 1  
 r.append(tmp)  
 r.popleft()  
 else:  
 return "Radiant"  
 else:  
 r.popleft()  
 if r:  
 tmp = max(r[-1], d[-1]) + 1  
 d.append(tmp)  
 d.popleft()  
 else:  
 return "Dire"  
  
 return "Radiant" if r else "Dire"

def predictPartyVictory(self, senate: str) -> str:  
 # 5:29 9/28/20  
  
 r, d = collections.deque(), collections.deque()  
 n = len(senate)  
 for i, s in enumerate(senate):  
 if s == "R":  
 r.append(i)  
 else:  
 d.append(i)  
  
 while r and d:  
 i, j = r.popleft(), d.popleft()  
  
 if i < j:  
 r.append(i + n)  
 else:  
 d.append(j + n)  
  
 return "Radiant" if r else "Dire"

**909. Snakes and Ladders**

class Solution:  
 def snakesAndLadders(self, board: List[List[int]]) -> int:  
 # 11:23 9/26/20  
 # 9:46 9/27/20  
  
 # 6:25 5/8/21  
 # 11:23 9/26/20  
 # 9:46 9/27/20  
  
 # 6:25 5/8/21  
 N = len(board)  
  
 def get(s):  
 # Given a square num s, return board coordinates (r, c)  
 quot, rem = divmod(s - 1, N)  
 row = N - 1 - quot  
 col = rem if row % 2 != N % 2 else N - 1 - rem  
 return row, col  
  
 N = len(board)  
 target = N \* N  
 dq = collections.deque([1])  
 visited = set([1])  
 steps = 0  
  
 while dq:  
 q\_len = len(dq)  
 for \_ in range(q\_len):  
 num = dq.popleft()  
 if num == target:  
 return steps  
  
 for i in range(1, 7):  
 next\_num = num + i  
 if next\_num > target:  
 break  
 if next\_num not in visited:  
 visited.add(next\_num)  
 r, c = get(next\_num)  
 tmp = next\_num  
 if board[r][c] != -1:  
 if board[r][c] not in visited:  
 tmp = board[r][c]  
 else:  
 continue  
 # if tmp not in visited: # Wrong  
 # visited.add(tmp)  
 dq.append(tmp)  
 steps += 1  
  
 return -1

5/11/2021

1. **826. Most Profit Assigning Work**

def maxProfitAssignment(self, difficulty: List[int], profit: List[int], worker: List[int]) -> int:  
 # 11:43 --> 11:54 --> 12:03 --> 12:14 5/6/21  
  
 def find\_difficulty(difficulties, diff\_worker):  
  
 lt, rt = 0, len(difficulties)  
 while lt < rt:  
 mid = lt + (rt - lt) // 2  
 if difficulties[mid] < diff\_worker:  
 lt = mid + 1  
 elif difficulties[mid] > diff\_worker:  
 rt = mid  
 else:  
 return mid  
 return rt - 1  
  
 diff\_profit = zip(difficulty, profit)  
 diff\_profit = sorted(diff\_profit, key=lambda x: x[0])  
  
 dic = {}  
 for i, v in enumerate(diff\_profit):  
 diff, prof = v[0], v[1]  
 if i == 0:  
 dic[diff] = prof  
 else:  
 pre\_diff = diff\_profit[i - 1][0]  
 dic[diff] = max(prof, dic[pre\_diff])  
  
 total\_profit = 0  
 difficulties = sorted(dic.keys())  
  
 for diff in worker:  
 diff\_index = find\_difficulty(difficulties, diff)  
 if diff\_index != -1:  
 total\_profit += dic[difficulties[diff\_index]]  
  
 return total\_profit

def maxProfitAssignment(self, difficulty: List[int], profit: List[int], worker: List[int]) -> int:  
 # 8:49 9/26/20  
 # 4:40 5/11/21  
  
 diff\_profit = sorted(zip(difficulty, profit))  
  
 i = total = best = 0  
  
 for diff in sorted(worker):  
 while i < len(diff\_profit) and diff >= diff\_profit[i][0]:  
 best = max(best, diff\_profit[i][1])  
 i += 1  
 total += best  
 return total

1. **1186. Maximum Subarray Sum with One Deletion**

def maximumSum(self, arr: List[int]) -> int:  
 max1 = arr[0] # max subarray sum with at most one deletion  
 max0 = arr[0] # max subarray sum with NO deletion  
 res: int = arr[0] # overall max to be returned  
 for a in arr[1:]:  
 max1 = max(max1 + a, max0, a) # include a, not include a, or start with a  
 max0 = max(max0 + a, a) # update `max0`  
 res = max(res, max1) # update overall max  
 return res

def maximumSum(self, arr: List[int]) -> int:  
 # 3:34 --> 3:56 --> 3:58 --> 9/11/20  
  
 n = len(arr)  
 max\_ending\_here0 = n \* [arr[0]] # no deletion  
 max\_ending\_here1 = n \* [arr[0]] # at most 1 deletion  
 for i in range(1, n):  
 max\_ending\_here0[i] = max(max\_ending\_here0[i - 1] + arr[i], arr[i])  
 max\_ending\_here1[i] = max(max\_ending\_here1[i - 1] + arr[i], arr[i])  
 if i >= 2:  
 max\_ending\_here1[i] = max(max\_ending\_here1[i], max\_ending\_here0[i - 2] + arr[i])  
 return max(max\_ending\_here1)

1. **1129. Shortest Path with Alternating Colors**

def shortestAlternatingPaths(self, n: int, red\_edges: List[List[int]], blue\_edges: List[List[int]]) -> List[int]:  
 # 4:43 10/2/20  
 # 5/14/21 12:48  
 from collections import defaultdict, deque  
 g = defaultdict(lambda: defaultdict(set))  
 red, blue = 0, 1  
 for st, end in red\_edges:  
 g[st][red].add(end)  
 for st, end in blue\_edges:  
 g[st][blue].add(end)  
  
 res = [float("inf")] \* n  
 level = 0  
 dq = deque([(0, red), (0, blue)])  
 while dq:  
 q\_size = len(dq)  
  
 for \_ in range(q\_size):  
 node, color = dq.popleft()  
 opp\_color = color ^ 1  
 res[node] = min(level, res[node])  
 next\_nodes = g[node][opp\_color]  
 for child in list(next\_nodes):  
 g[node][opp\_color].remove(child)  
 dq.append((child, opp\_color))  
  
 level += 1  
  
 return [r if r != math.inf else -1 for r in res]

1. **1524. Number of Sub-arrays With Odd Sum**

def numOfSubarrays(self, arr: List[int]) -> int:  
 # 1:39 --> 2:01 5/12/21  
 n = len(arr)  
  
 # dp = [[0,0]] \* n # BAD  
 dp = [[0, 0] for \_ in range(n)] # even, odd  
 total = 0  
  
 for i, v in enumerate(arr):  
 if i == 0:  
 if v % 2 == 0:  
 dp[i][0] = 1  
 dp[i][1] = 0  
 else:  
 dp[i][0] = 0  
 dp[i][1] = 1  
  
 else:  
 if v % 2 == 0:  
 dp[i][0] = dp[i - 1][0] + 1  
 dp[i][1] = dp[i - 1][1]  
 else:  
 dp[i][0] = dp[i - 1][1]  
 dp[i][1] = dp[i - 1][0] + 1  
  
 total = (dp[i][1] + total) % (10 \*\* 9 + 7)  
  
 return total

def numOfSubarrays(self, arr: List[int]) -> int:  
 # 9:48 9/22/20  
  
 ans = 0  
 even = 0  
 odd = 0  
 for v in arr:  
 if v % 2 == 0:  
 even, odd = even + 1, odd  
 else:  
 even, odd = odd, even + 1  
 ans = (ans + odd) % 1000000007  
 return ans % 1000000007

**5/15/21**

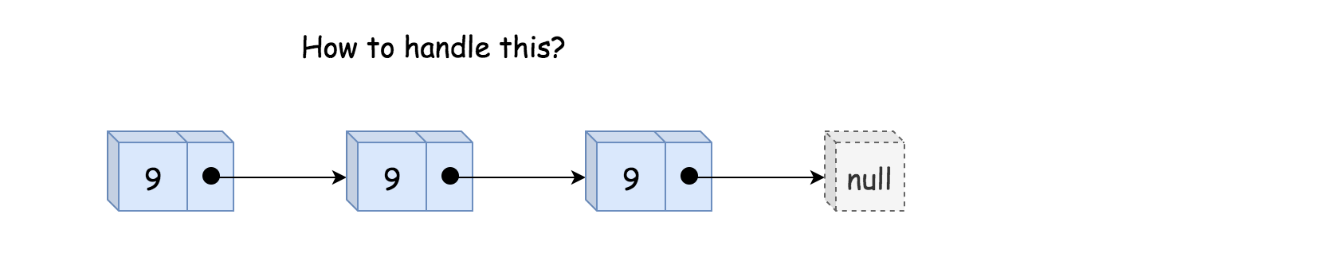
1. **82. Remove Duplicates from Sorted List II**

def deleteDuplicates(self, head: ListNode) -> ListNode:  
 # 10:53--> 11:01 --> 11:09 12/2/2020  
 # 11:06 12/3/20  
  
 dummy = cur = ListNode(0, head)  
  
 while head:  
 if head.next and head.val == head.next.val:  
 while head.next and head.val == head.next.val:  
 head = head.next  
 cur.next = head.next  
 else:  
 cur = cur.next  
  
 head = head.next  
  
 return dummy.next

#### Approach 1: Sentinel Head + Predecessor

**Sentinel Head**

Let's start from the most challenging situation: the list head is to be removed.

 Figure 1. The list head is to be removed.

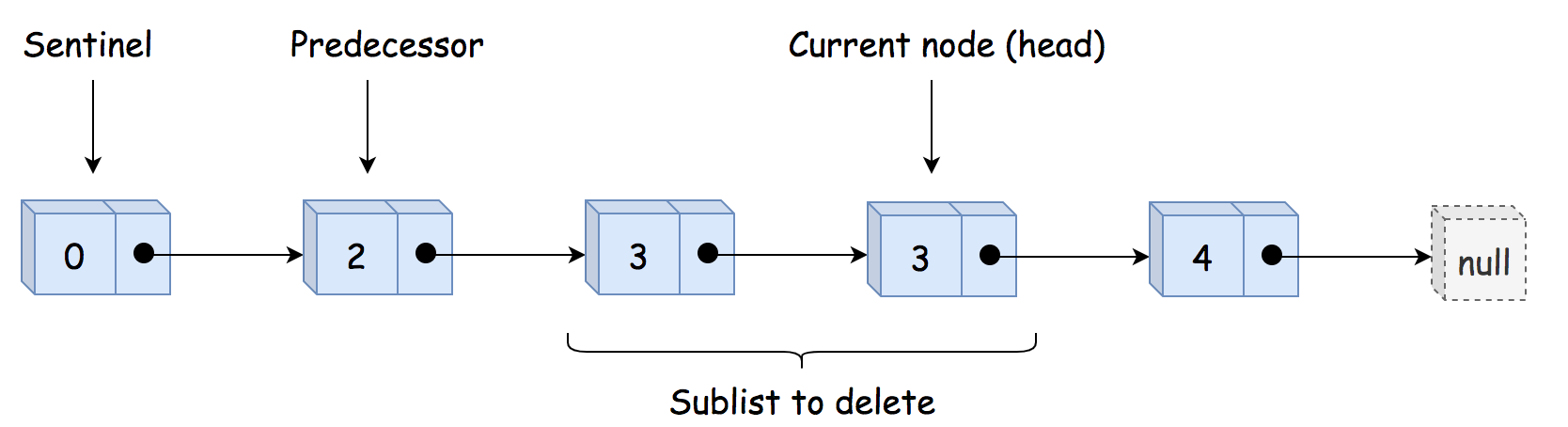
The standard way to handle this use case is to use the so-called [Sentinel Node](https://en.wikipedia.org/wiki/Sentinel_node). Sentinel nodes are widely used for trees and linked lists as pseudo-heads, pseudo-tails, etc. They are purely functional and usually don't hold any data. Their primary purpose is to standardize the situation to avoid edge case handling.

For example, let's use here pseudo-head with zero value to ensure that the situation "delete the list head" could never happen, and all nodes to delete are "inside" the list.

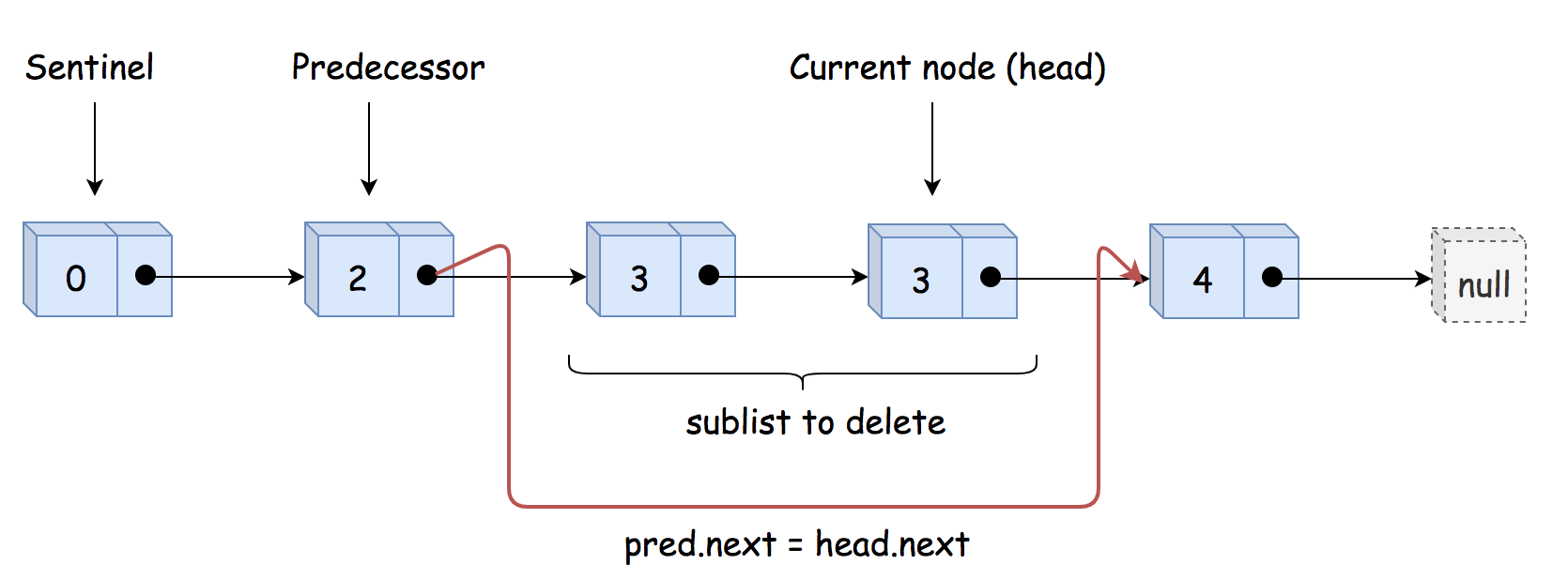
**Delete Internal Nodes**

The input list is sorted, and we can determine if a node is a duplicate by comparing its value to the node after it in the list. Step by step, we could identify the current sublist of duplicates.

Now it's time to delete it using pointer manipulations. Note that the first node in the duplicates sublist should be removed as well. That means that we have to track the predecessor of duplicates sublist, i.e., the last node before the sublist of duplicates.

 Figure 2. The sentinel head, the predecessor, and the sublist of duplicates to delete.

Having predecessor, we skip the entire duplicate sublist and make predecessor to point to the node after the sublist.

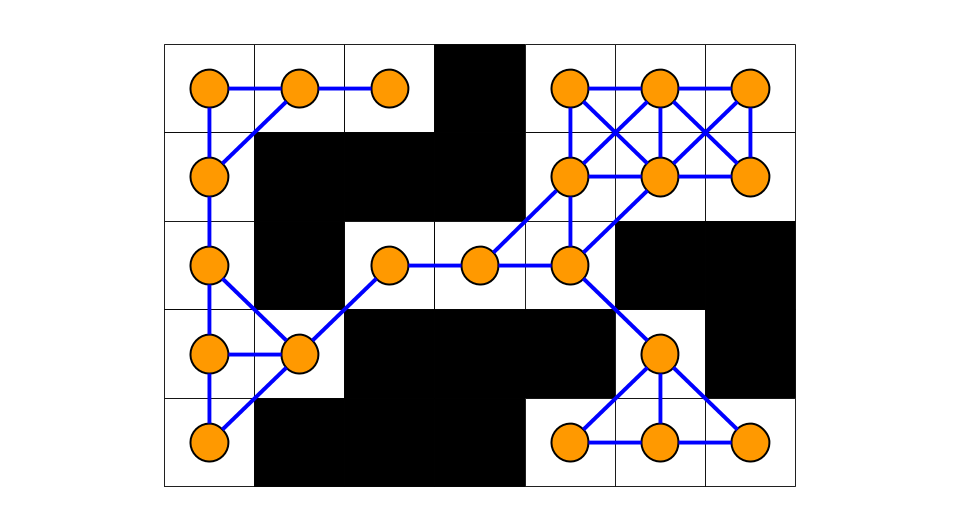
 Figure 2. Delete the sublist of duplicates.

**Implementation**

class Solution:  
 def deleteDuplicates(self, head: ListNode) -> ListNode:  
 # sentinel  
 sentinel = ListNode(0, head)  
  
 # predecessor = the last node   
 # before the sublist of duplicates  
 pred = sentinel  
  
 while head:  
 # if it's a beginning of duplicates sublist   
 # skip all duplicates  
 if head.next and head.val == head.next.val:  
 # move till the end of duplicates sublist  
 while head.next and head.val == head.next.val:  
 head = head.next  
 # skip all duplicates  
 pred.next = head.next  
 # otherwise, move predecessor  
 else:  
 pred = pred.next  
  
 # move forward  
 head = head.next  
  
 return sentinel.next

**5/16/21**

1. **1091. Shortest Path in Binary Matrix**
2. If we do this with every cell, we get the following.

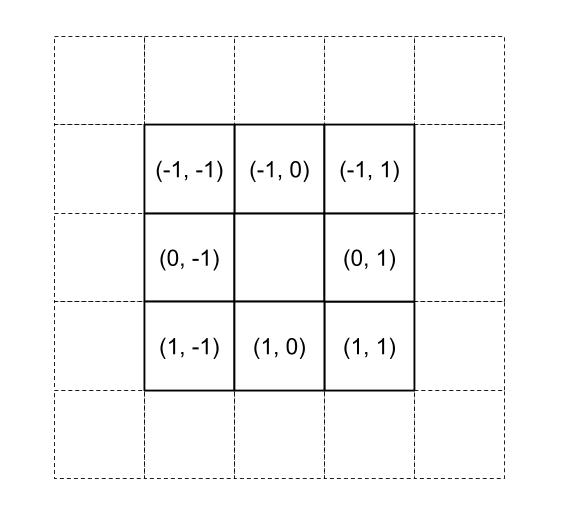


Notice that what we have discovered is that the grid is a **graph**; white cells are nodes, and lines between them are edges. This is a special type of graph we call a [lattice graph](https://en.wikipedia.org/wiki/Lattice_graph). 2D arrays that are representing a graph come up a lot in interview questions. It is essential to be confident with them (don't worry too much about how we will implement this yet; we'll get to that in a bit).

So, we have a graph, and we need to find the length of the shortest path from the top-left to the bottom-right cell. Recall that to find the shortest path in a graph, we should use Breadth-first Search (BFS).

Finding the shortest path between two nodes in a graph is almost always done using BFS, and all programmers should know this. BFS is one of the fundamental algorithms that you are expected to be confident coding before a tech interview. So, if you're finding this question challenging, then you're doing the right thing by working on it now.

Applying BFS to this problem, we'll use the queue to keep track of cells that we have numbered but haven't yet numbered the \* neighbors\* of. While usually for BFS, we'd need a "visited" set to avoid infinite looping around cycles, we won't need one for this approach because we're going to overwrite the input, and so only unvisited cells will have a 0 in them.



The most common pattern is to put these "offsets" into a list as follows.

directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]

#### Approach 2: Breadth-first Search (Without Overwriting the Input)

**Intuition**

The first approach is nice in that it's very intuitive—it's directly based on how you might solve the problem on a whiteboard. It also avoids the need for a "visited" set or data structures to keep track of distances, thus saving a constant amount of memory over typical BFS implementations. However, like all in-place algorithms, overwriting the input can cause problems. Here are a couple of possible scenarios you need to consider.

1. That the algorithm is running in a \* multithreaded\* environment, and it does not have exclusive access to the grid. Other threads might need to read the grid too, and might not expect it to be modified.
2. That there is only a single thread or the algorithm has exclusive access to the grid while running, **but** the grid might need to be reused later or by another thread once the lock has been released.

For the second scenario, Approach 1 could be modified to restore the grid—simply loop over it at the end, replacing all numbers that are greater than 1 with a 0, and additionally set the top-left cell to a 0.

For the first scenario, though, this won't work. We would have to come up with an algorithm that doesn't modify the input.

You should **always** discuss the possibility of overwriting the input with your interviewer and clarify what kind of environment your algorithm is expected to run in. Sometimes they won't care, sometimes they'll state it has to run in a multithreaded environment, or sometimes they'll have a particular preference as it impacts what they're trying to see from you.

Anyway, to avoid over-writing the input, we could go for a more traditional BFS algorithm that uses a "visited" set (which can be implemented as either a hash set or a new 2D array). It will also need to keep track of the distances in some other way.

**Algorithm**

Firstly, we can reuse our get\_neighbors(...) function from above. It won't need any modifications as we'll handle the "visited" logic in the main function. This is another big advantage of keeping the get\_neighbors(...) logic separate; even if you end up completely changing your main algorithm, this bit won't need re-writing.

For the actual BFS algorithm, there are several variants that we'll now talk about. All use a visited set. All achieve the same result and have the same time and space complexities; however, all are based on different ways of intuitively viewing the problem. Like always, it's great to understand more than one way of solving a problem.

Distances on queue

The simplest variant is to use a visited set and to put the distances on the queue, alongside the row and column (triplets instead of pairs).

visited = a new set

queue = a new queue

enqueue (0, 0, 1)

add (0, 0) to visited

while the queue is not empty:

row, col, distance = dequeue and unpack a cell

if (row, col) is the bottom right cell:

return distance

for each open neighbour:

if neighbour is in visited:

continue

otherwise, add neighbour to visited

enqueue (neighbour\_row, neighbour\_col, distance + 1)

return -1

This approach is nice in that it's very easy to code and reason about. While the distances going on the queue are repetitive, correct code is generally worth a lot more "points" in an interview than attempted-to-optimize-but-it' s-buggy code is!

Starting a new collection for each distance

BFS works by examining cells in order of distance from the starting point. In other words, all cells at a distance of x are visited before any cells at a distance of x + 1. Additionally, cells at a distance of x can only enqueue other cells that are at a distance of x + 1. Therefore, there are at most two unique distances in the queue at any one time.

This implementation utilizes this property by keeping track of two collections of cells to be explored: the remaining ones at the current distance and the ones at the next distance. There is no need to use queues, as the order that cells of the same distance are explored does not matter. Any data structure that has O(1)*O*(1) insertions and removals will do.

visited = a new set

add (0, 0) to visited

current\_layer = a new list

next\_layer = a new list

add (0, 0) to current layer

while current\_layer is not empty:

row, col = remove and unpack a cell from current\_layer

if (row, col) is the bottom right cell (target):

return distance

for each open neighbor:

if the neighbor is in visited:

continue

add neighbor to visited

add neighbor to next\_layer

if current\_layer is now empty:

current\_distance += 1

current\_layer = next\_layer

next\_layer = a new list

return -1

Keeping track of how many cells at each distance are on the queue

This final implementation uses the same BFS property as the above one, but in a different way.

At the start, there is exactly 1 cell at a distance of 1. Once we have dequeued and processed that cell, we know all cells currently in the queue must be of distance 2. We can check at this point how many of them there are and then dequeue and process that number of cells. Now we know all of the cells in the queue are of distance 3. This argument extends to the entire grid.

visited = a new set

queue = a new queue

enqueue cell (0, 0)

add (0, 0) to visited

current\_distance = 1

while queue is not empty:

nodes\_on\_queue = current queue length

repeat nodes\_on\_queue\_times:

row, col = dequeue and unpack a cell

if (row, col) is the bottom right cell (target):

return distance

for each open neighbour:

if neighbour is in visited:

continue

add neighbour to visited

enqueue (neighbour\_row, neighbour\_col, distance + 1)

current\_distance += 1

return -1

While elegant, this implementation is more complicated and has more room for bugs. If you are confident, though, this is probably the way to go.

def shortestPathBinaryMatrix(self, grid: List[List[int]]) -> int:  
 # 2:20 5/12/21  
  
 from collections import deque  
 n = len(grid)  
 step = 1  
 visited = set([(0, 0)])  
 dq = deque([(0, 0)])  
 target = (n - 1, n - 1)  
  
 if grid[0][0]:  
 return -1  
  
 dirs = [(0, -1), (0, 1), (1, 0), (-1, 0), (-1, -1), (1, 1), (-1, 1), (1, -1)]  
  
 while dq:  
 dq\_size = len(dq)  
 for i in range(dq\_size):  
 top = dq.popleft()  
 # if top == target:  
 if top[0] == target[0] and top[1] == target[1]:  
 return step  
  
 for r, c in dirs:  
 row, col = top[0] + r, top[1] + c  
 if 0 <= row < n and 0 <= col < n:  
 if grid[row][col] == 0 and (row, col) not in visited:  
 visited.add((row, col))  
 dq.append((row, col))  
 step += 1  
  
 return -1

def shortestPathBinaryMatrix(self, grid: List[List[int]]) -> int:  
 # 8:50 9/24/20  
 # BFS and overwriting original grid  
 # NO visited check needed due to overwriting  
 n = len(grid)  
 if not grid or grid[n - 1][n - 1] == 1 or grid[0][0] == 1: return -1  
  
 # visited = [[False] \* n for \_ in range(n) ]  
 dq = collections.deque([(0, 0)])  
 # visited[0][0] = True  
 cnt = 0  
 moves = {(0, -1), (0, 1), (-1, -1), (1, 1), (-1, 0), (1, 0), (1, -1), (-1, 1)}  
  
 while dq:  
 m = len(dq)  
 for i in range(m):  
 x, y = dq.popleft()  
 if x == y == n - 1:  
 return cnt + 1  
 # if visited[x][y]:  
 # continue  
 # visited[x][y] = True  
 for mv in moves:  
 dx, dy = x + mv[0], y + mv[1]  
 if 0 <= dx < n and 0 <= dy < n and grid[dx][dy] == 0:  
 dq.append((dx, dy))  
 grid[dx][dy] = 1  
  
 cnt += 1  
  
 return -1

class Solution:  
 def shortestPathBinaryMatrix(self, grid: List[List[int]]) -> int:  
  
 max\_row = len(grid) - 1  
 max\_col = len(grid[0]) - 1  
 directions = [  
 (-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]  
  
 # Helper function to find the neighbors of a given cell.  
 def get\_neighbours(row, col):  
 for row\_difference, col\_difference in directions:  
 new\_row = row + row\_difference  
 new\_col = col + col\_difference  
 if not (0 <= new\_row <= max\_row and 0 <= new\_col <= max\_col):  
 continue  
 if grid[new\_row][new\_col] != 0:  
 continue  
 yield (new\_row, new\_col)  
  
 # Check that the first and last cells are open.   
 if grid[0][0] != 0 or grid[max\_row][max\_col] != 0:  
 return -1  
  
 # Set up the BFS.  
 queue = deque([(0, 0, 1)])  
 visited = {(0, 0)}  
  
 # Do the BFS.  
 while queue:  
 row, col, distance = queue.popleft()  
 if (row, col) == (max\_row, max\_col):  
 return distance  
 for neighbour in get\_neighbours(row, col):  
 if neighbour in visited:  
 continue  
 visited.add(neighbour)  
 # Note that the \* splits neighbour into its values.  
 queue.append((\*neighbour, distance + 1))  
  
 # There was no path.  
 return -1

class Solution:  
 def shortestPathBinaryMatrix(self, grid: List[List[int]]) -> int:  
  
 max\_row = len(grid) - 1  
 max\_col = len(grid[0]) - 1  
 directions = [  
 (-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]  
  
 # Helper function to find the neighbors of a given cell.  
 def get\_neighbours(row, col):  
 for row\_difference, col\_difference in directions:  
 new\_row = row + row\_difference  
 new\_col = col + col\_difference  
 if not (0 <= new\_row <= max\_row and 0 <= new\_col <= max\_col):  
 continue  
 if grid[new\_row][new\_col] != 0:  
 continue  
 yield (new\_row, new\_col)  
  
 # Check that the first and last cells are open.   
 if grid[0][0] != 0 or grid[max\_row][max\_col] != 0:  
 return -1  
  
 # Set up the BFS.  
 current\_layer = [(0, 0)]  
 next\_layer = []  
 visited = {(0, 0)}  
 current\_distance = 1  
  
 while current\_layer:  
  
 # Process the current layer.  
 for row, col in current\_layer:  
 if (row, col) == (max\_row, max\_col):  
 return current\_distance  
 for neighbour in get\_neighbours(row, col):  
 if neighbour in visited:  
 continue  
 visited.add(neighbour)  
 next\_layer.append(neighbour)  
  
 # Set up for processing the next layer.  
 current\_distance += 1  
 current\_layer = next\_layer  
 next\_layer = []  
  
 # There was no path.  
 return -1

class Solution:  
  
 def shortestPathBinaryMatrix(self, grid: List[List[int]]) -> int:  
  
 max\_row = len(grid) - 1  
 max\_col = len(grid[0]) - 1  
 directions = [  
 (-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]  
  
 # Helper function to find the neighbors of a given cell.  
 def get\_neighbours(row, col):  
 for row\_difference, col\_difference in directions:  
 new\_row = row + row\_difference  
 new\_col = col + col\_difference  
 if not (0 <= new\_row <= max\_row and 0 <= new\_col <= max\_col):  
 continue  
 if grid[new\_row][new\_col] != 0:  
 continue  
 yield (new\_row, new\_col)  
  
 # Check that the first and last cells are open.   
 if grid[0][0] != 0 or grid[max\_row][max\_col] != 0:  
 return -1  
  
 # Set up the BFS.  
 queue = deque([(0, 0)])  
 visited = {(0, 0)}  
 current\_distance = 1  
  
 # Do the BFS.  
 while queue:  
 # Process all nodes at current\_distance from the top-left cell.  
 nodes\_of\_current\_distance = len(queue)  
 for \_ in range(nodes\_of\_current\_distance):  
 row, col = queue.popleft()  
 if (row, col) == (max\_row, max\_col):  
 return current\_distance  
 for neighbour in get\_neighbours(row, col):  
 if neighbour in visited:  
 continue  
 visited.add(neighbour)  
 queue.append(neighbour)  
 # We'll now be processing all nodes at current\_distance + 1  
 current\_distance += 1  
  
 # There was no path.  
 return -1

**### LAST**