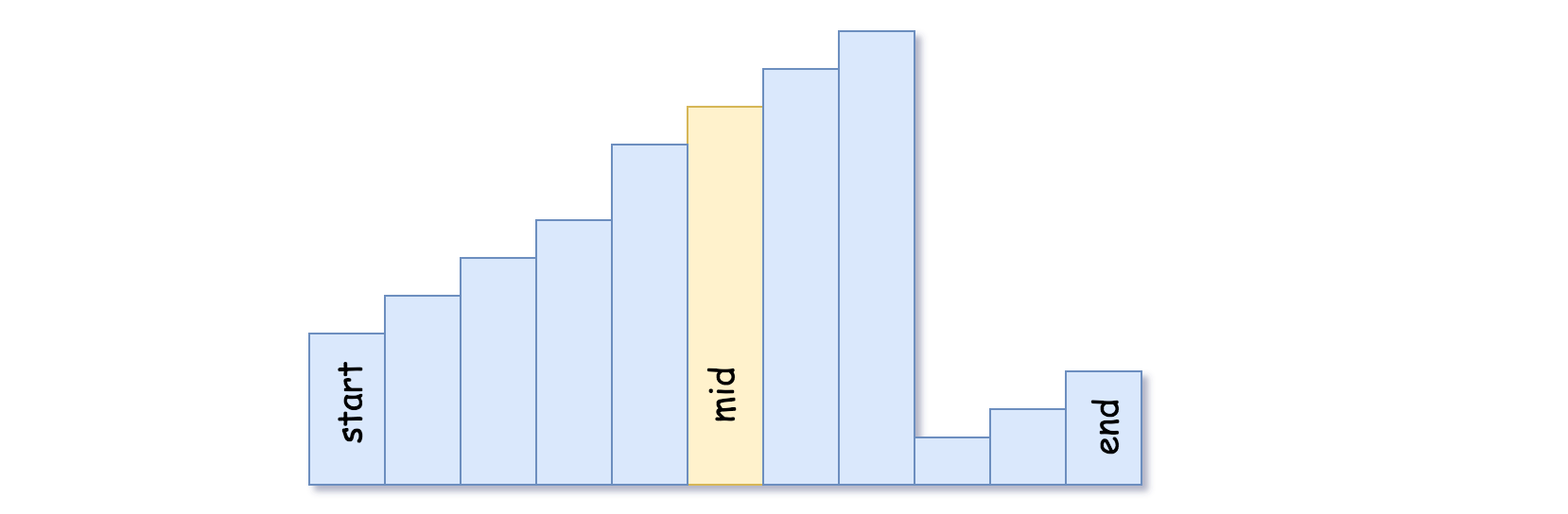
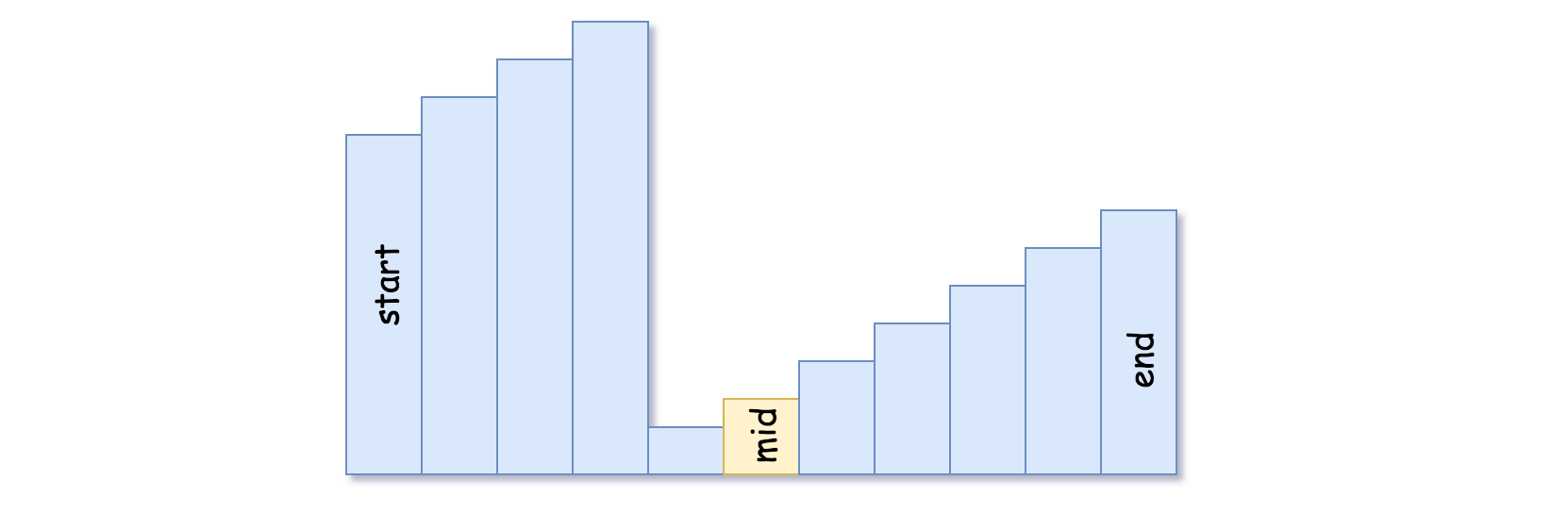
Leetcode compendium

Arrays!

1. Search in a rotated sorted array





class Solution:

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

start = nums[0]

low, high = 0, len(nums) - 1

while low <= high:

mid = (low + high) // 2

if nums[mid] == target:

return mid

elif nums[mid] >= nums[low]:

if target >= nums[low] and target <= nums[mid]:

high = mid - 1

else:

low = mid + 1

else:

if target >= nums[mid] and target <= nums[high]:

low = mid + 1

else:

high = mid - 1

return -1

1. Binary search template 2

Find the first bad version

Trick is left < right strictly and

If left == right and isBadVersion(left):

Return left

This will converge Try on just 2 elements (F,T) and (T,F)

class Solution:

def firstBadVersion(self, n: int) -> int:

low, high = 1, n

while low < high:

mid = (low + high) // 2

if isBadVersion(mid):

high = mid

else:

low = mid + 1

if low == high and isBadVersion(low):

return low

return -1

1. Peak Finder

class Solution:

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

low, high = 0, len(nums) - 1

while low < high:

mid = (low + high)//2

if nums[mid] > nums[mid+1]:

high = mid

else:

low = mid + 1

return low

Binary template 2

High = mid and not mid – 1

1. Find minimum in a rotated array

Trick is to compare to start element!

class Solution:

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

low, high = 0, len(nums) - 1

while low < high:

mid = (low + high) // 2

if nums[mid] > nums[mid+1]:

return nums[mid + 1]

else:

if nums[mid] < nums[low]:

high = mid

else:

low = mid + 1

return nums[0]

1. Find minimum in duplicated rotated sorted array

Only compare to high!!

class Solution:

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

N = len(nums)

start,end = 0,N-1

while start < end:

mid = (start + end) // 2

if nums[mid] > nums[end]:

start = mid + 1

elif nums[mid] < nums[end]:

end = mid

else:

end -= 1

return nums[start]

1. Two Sum

One pass hashtable Try the case(3,3) and target = 6, no need to special case the equal element, the next element would be seen for the VERY FIRST time so you can just return [I and dictionary[element])

Do NOT CREATE A LIST AS THIS IS YOUR FIRST INSTINCT

class Solution:

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

d = defaultdict(int)

for i,n in enumerate(nums):

element = target - n

if element in d:

return [i,d[element]]

else:

d[n] = i

return [-1,-1]

1. Valid parenthesis

Check if there are equal parans and then use a stack to match them

Simple!!

class Solution:

def isValid(self, s: str) -> bool:

stack = list()

match = {'{':'}', '(': ')', '[' : ']'}

c = Counter(s)

if not c['{'] == c['}'] or not c['('] == c[')'] or not c['['] == c[']']:

return False

for bracket in s:

if bracket in ['{','(','[']:

stack.append(bracket)

elif stack:

popped = stack.pop()

if match[popped] != bracket:

return False

return len(stack) == 0

1. Merge two sorted lists

Beautiful and easy solution!!

def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:

dummy = ListNode()

head = dummy

while list1 and list2:

if list1.val < list2.val:

dummy.next = list1

list1 = list1.next

else:

dummy.next = list2

list2 = list2.next

dummy = dummy.next

dummy.next = list1 or list2

return head.next

1. Valid Palindrome

Use simple regex, findall search

In case of findall a list is returned and search returns a match object so use match.span() or match.start() or match.end()

Match.group() returns the actual object

class Solution:

def isPalindrome(self, s: str) -> bool:

#REGEX OVERVIEW IN PYTHON

#Use re.search or re.findall

cleaned = ''.join(re.findall(r'[a-z0-9]',s.lower()))

return cleaned == cleaned[::-1]

1. Invert binary tree!

Mirroring is as simple as swapping left and right nodes

class Solution:

def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:

def mirror(root):

if not root:

return None

if root.left or root.right:

root.left, root.right = root.right, root.left

mirror(root.left)

mirror(root.right)

return root

return mirror(root)

1. Lowest Common Ancestor

Beautiful question!! Use the property of the binary search tree to find which subtree to look at and the FIRST SPLIT is the lowest common ancestor

class Solution:

def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':

if not root:

return None

if p.val < root.val and q.val < root.val:

#left subtree

return self.lowestCommonAncestor(root.left, p, q)

elif p.val > root.val and q.val > root.val:

return self.lowestCommonAncestor(root.right, p, q)

else:

return root

1. Merge overlapping intervals!

Simple solution is –

Check if the next interval starts before the previous one ends and then there is OBVIOUSLY an overlap!!

Sort your intervals and update the end to max of the previous and current intervals!

class Solution:

def merge(self, intervals: List[List[int]]) -> List[List[int]]:

if len(intervals) <= 1:

return intervals

intervals.sort(key=lambda a:a[0])

output = []

for i in range(len(intervals)):

if not output or intervals[i][0] > output[-1][1]:

output.append(intervals[i])

else:

output[-1][1] = max(intervals[i][1], output[-1][1])

return output

1. Balanced Binary Tree

Simply calculate the height and the height diff in the height function itself

class Solution:

def isBalanced(self, root: Optional[TreeNode]) -> bool:

if not root:

return True

result = True

def height(root):

if not root:

return 0

nonlocal result

LR = height(root.left)

RR = height(root.right)

result &= (abs(LR - RR) <= 1)

return max(LR,RR) + 1

height(root)

return result

1. **K Closest Points to Origin**

Use a heap!! Very interesting to use a max heap which will have the kth farthest element on the top and then you can match subsequent elements that are closer by comparing with the maximum elements on the heap

class Solution:

def kClosest(self, points: List[List[int]], k: int) -> List[List[int]]:

#points.sort(key = lambda x : sqrt(x[0]\*\*2 + x[1]\*\*2))

#return points[:k]

dist = [(-sqrt(points[i][0]\*\*2 + points[i][1]\*\*2),i) for i in range(k)]

heapq.heapify(dist)

print(dist)

for i in range(k,len(points)):

d = -sqrt(points[i][0]\*\*2 + points[i][1]\*\*2)

print(d)

if d > dist[0][0]:

heapq.heappushpop(dist, (d,i))

print(dist)

return [points[i] for (\_,i) in dist]

1. **3 Sum**

**Sort and use two pointers!! Its O(N2 + NLOGN) the two sum2 is linear so we call two sum2 n times**

class Solution:

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

nums.sort()

res = []

for i in range(len(nums)):

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

continue

target = nums[i]

left, right = i+1, len(nums) - 1

while left < right:

if nums[left] + nums[right] == -target:

res.append([target, nums[left], nums[right]])

left += 1

right -= 1

while left < right and nums[left] == nums[left-1]:

left += 1

elif nums[left] + nums[right] > -target:

right -= 1

else:

left += 1

return res

1. Graph DFS

Make adjacency list (dictionary first)!!!

Valid path from source to destination for a bidirectional graph

Make a visited() set and you can also say that if the destination is in visited

There is a valid path

class Solution:

def validPath(self, n: int, edges: List[List[int]], source: int, destination: int) -> bool:

result = False

visited = [False]\*n

if not edges:

return True

adjList = defaultdict(list)

for v1,v2 in edges:

adjList[v1].append(v2)

adjList[v2].append(v1)

def dfs(vertex):

nonlocal result

if vertex == destination:

return True

for v in adjList[vertex]:

if visited[v]:

continue

visited[v] = True

result |= dfs(v)

return result

return dfs(source)

Time complexity is O(2^N\*V)

1. Find Itinerary!

Base case is when the route is the len(tickets) + 1

Backtrack when a path fails

class Solution:

def findItinerary(self, tickets: List[List[str]]) -> List[str]:

adjList = defaultdict(list)

for t in tickets:

adjList[t[0]].append([t[1],False])

for k in adjList:

adjList[k] = sorted(adjList[k], key=lambda x:x[0])

itinerary = ["JFK"]

def dfs(v):

nonlocal itinerary

if len(itinerary) == len(tickets) + 1:

return True

d = adjList[v]

for t in d:

if not t[1]:

itinerary.append(t[0])

t[1] = True

if not dfs(t[0]):

itinerary.pop() #backtrack, went on the wrong path

t[1] = False #may be its valid later

else:

return True

return False

dfs("JFK")

return itinerary

1. Graph is a valid tree

Just do recursive depth first search and check for trivial cycles (bidirectional graph) using the check if the neigbour == parent

class Solution:

def validTree(self, n: int, edges: List[List[int]]) -> bool:

visited = [False]\*n

adj = defaultdict(list)

for e in edges:

adj[e[0]].append(e[1])

adj[e[1]].append(e[0])

root = 0

def dfs(vertex, parent):

if len(adj[vertex]) > 3 and vertex != root:

return False

for v in adj[vertex]:

if v == parent:

continue

if not visited[v]:

visited[v] = True

ret = dfs(v,vertex)

if not ret:

return False

else:

return False

return True

result = dfs(root,None)

for i in range(1,n):

if not visited[i]:

return False

return result

edges should be N-1

you can use set for visited and then check if len(visited) == n

graph should be fully connected with no cycles to be a tree!

1. Course Schedule (Graph) – Attempt again

Great question!! Use another array for checked and see time complexity drop to O(V+E). Remember this is postorder DFS so we backtrack dfs by visiting the children first and the parent later. The logic is that if the subgraph has no cycles, then adding a new node aka parent wont add a cycle. Brilliant!

class Solution:

def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:

n = numCourses

adj = defaultdict(list)

if not prerequisites:

return True

for p in prerequisites:

adj[p[1]].append(p[0])

visited = [False]\*n

checked = [False]\*n

def backtrackDFS(vertex):

if checked[vertex]:

return True

if visited[vertex]:

return False

visited[vertex] =True

for v in adj[vertex]:

ret = backtrackDFS(v)

if not ret:

return False

visited[vertex] = False

checked[vertex] = True

return True

for i in range(n):

if not backtrackDFS(i):

return False

return True

1. Longest Palindrome!

Greedy simple logic – add x //2 \* 2 and then if result is even, just add 1 to it if the value you are adding is odd.

class Solution:

def longestPalindrome(self, s: str) -> int:

ctr = Counter(s)

result = 0

for v in ctr.values():

result += (v//2) \* 2

if result % 2 == 0 and v % 2 == 1:

result += 1

return result

1. Product of array except self

YOU ARE A BOSS! You remember the left and right product approach.

Use this clever trick for O(N) space

class Solution:

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

n = len(nums)

p1 = [0] \* n

p2 = [0] \* n

p1[0] = 1

for i in range(1,n):

p1[i] = p1[i-1]\*nums[i-1]

Update prev later, multiply first so the i+1th is available for ith iteration

prev = 1

for i in reversed(range(n)):

p1[i] \*= prev

prev \*= nums[i]

return p1

1. All Paths from source to target

Use the queue to append paths itself!!

Then just use popped path[-1] as the node to test upon!

class Solution:

def allPathsSourceTarget(self, graph: List[List[int]]) -> List[List[int]]:

paths = []

n = len(graph)

dst = n-1

def bfs(src,dst):

queue = deque([[src]])

while queue:

path = queue.popleft()

node = path[-1]

if node == dst:

paths.append(path[:])

for v in graph[node]:

copy = path[:]

copy.append(v)

queue.append(copy)

bfs(0,dst)

return paths

Space Complexity = Time Complexity = O(2^V \* V)

2\*V Paths From Src To Dst And Then O(V) For Each Path

1. Majority Element

Beautiful solution! Voting algorithm

Example

5757575755

choose a suffix such that for that window that suffix is the majority element

count +1 for those and -1 for others

eventually everything will cancel out and majority element will remain

class Solution:

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

count = 0

suffix = -1

for n in nums:

if count == 0:

suffix = n

if n == suffix:

count += 1

else:

count -= 1

return suffix

1. Diameter of a binary tree!

Just use the height function and keep track of the sum of heights of different subtrees

class Solution:

def diameterOfBinaryTree(self, root: Optional[TreeNode]) -> int:

if not root:

return 0

maxHeight = -1

def height(root):

nonlocal maxHeight

if not root:

return 0

left = height(root.left)

right = height(root.right)

maxHeight = max(left + right, maxHeight)

return max(left,right) + 1

height(root)

return maxHeight

1. Min Stack!

Smart AF question, use two stacks to keep track of minimum value

Push <= min(value) on the min stack and pop it off when the top of the stack == top of min element. Alternatively we can use a counter to keep track of duplicate minimum values

class MinStack:

def \_\_init\_\_(self):

self.minElements = []

self.stack = []

def push(self, val: int) -> None:

if not self.minElements:

self.minElements.append(val)

elif val <= self.minElements[-1]:

self.minElements.append(val)

self.stack.append(val)

def pop(self) -> None:

element = self.stack.pop()

if self.minElements and element == self.minElements[-1]:

self.minElements.pop()

def top(self) -> int:

if self.stack:

return self.stack[-1]

return -1

def getMin(self) -> int:

if self.minElements:

return self.minElements[-1]

return -1

1. Daily temperatures

Genius solution!

Use a stack of indices, pop when the temperature > top of the stack else append it

Update the result as you pop

class Solution:

def dailyTemperatures(self, temperatures: List[int]) -> List[int]:

n = len(temperatures)

result = [0]\*n

stack = list()

stack.append(0)

for i in range(1,n):

while stack and temperatures[i] > temperatures[stack[-1]]:

top = stack.pop()

result[top] = i - top

stack.append(i)

return result

Time complexity: O(N)*O*(*N*)

At first glance, it may look like the time complexity of this algorithm should be O(N^2)*O*(*N*2), because there is a nested while loop inside the for loop. However, each element can only be added to the stack once, which means the stack is limited to N*N* pops. Every iteration of the while loop uses 1 pop, which means the while loop will not iterate more than N*N* times in total, across all iterations of the for loop.

1. Valid Binary Search Tree

Simple solution keep limits for left and right subtree that update with each traversal of DFS

class Solution:

def isValidBST(self, root: Optional[TreeNode]) -> bool:

def dfs(root, start, end):

if not root:

return True

value = root.val

if not start < value < end:

return False

L,R = True, True

if root.left:

L = dfs(root.left, start, value)

if root.right:

R = dfs(root.right, value, end)

return L and R

return dfs(root, float('-inf'), float('inf'))

1. Combination Sum!

Very simple, use whatever you do to backtrack but PASS THE START INDEX i.e give the current index another chance. Do not go over all the candidates in each pass to avoid duplicated. If you sort the output, it will lead to loss of time.

* Even though we have already the element 4 in the current combination, we are giving the element *another chance* in the next exploration, since the combination can contain **duplicate** numbers.

class Solution:

def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:

combinations = []

def backtrack(current, target, start):

if target < 0:

return

if target == 0:

combinations.append(current)

return

for i in range(start,len(candidates)):

current.append(candidates[i])

backtrack(current[:], target - candidates[i], i)

current.pop()

backtrack([],target,0)

return combinations

Complexity – O(N^T/M)

* The maximal depth of the tree, would be \frac{T}{M}*MT*​, where we keep on adding the smallest element to the combination.

1. Combination Sum 2

Only unique elements and duplicates allowed but the final result should be unique

Approach – Use Counter

class Solution:

def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:

combinations = []

ctr = Counter(candidates)

def backtrack(current, target, start, ctrTup):

if target < 0:

return

if target == 0:

combinations.append(current)

return

for i in range(start,len(ctrTup)):

value, freq = ctrTup[i]

if freq <= 0:

continue

current.append(value)

ctrTup[i] = (value, freq - 1)

backtrack(current[:], target - value, i, ctrTup)

ctrTup[i] = (value, freq)

current.pop()

backtrack([],target,0,[(k,v) for k,v in ctr.items()])

return combinations

Time Complexity – O(2^N) and Space complexity – O(N)

1. Word Search

Simple DFS backtracking

* No visited necessary
* Mark the starting vertex as MARKER
* In get valid positions, only return positions that HAVE the LETTER you want

class Solution:

def exist(self, board: List[List[str]], word: str) -> bool:

R,C = len(board), len(board[0])

MARKER = '#'

def getValidPositions(i,j, letter):

pos = [(i,j-1), (i,j+1), (i-1,j), (i+1,j)]

for p in pos:

x,y = p

if x < 0 or x > R - 1 or y < 0 or y > C - 1:

continue

if board[x][y] != letter:

continue

yield p

def backtrack(i,j, curr):

if "".join(curr) == word:

return True

letter = word[len(curr)]

pos = getValidPositions(i,j, letter)

for p in pos:

x,y = p

board[x][y] = MARKER

curr.append(letter)

if backtrack(x,y,curr):

return True

curr.pop()

board[x][y] = letter

return False

for i in range(R):

for j in range(C):

if board[i][j] == word[0]:

board[i][j] = MARKER

if backtrack(i,j,[word[0]]):

return True

board[i][j] = word[0]

return False

Time Complexity:

O(N \* 3^L)

L is the length of the word to be matched

Space Complexity:

Recursion call stack O(L)

1. Max Consecutive Ones (Sliding Window)

In sliding window, we want to expand the right and contract from the left.

class Solution:

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

seenZero = False

maxOnes = -1

windowStart, windowEnd = 0,0

lastPos = -1

while windowEnd < len(nums):

if seenZero and nums[windowEnd] == 0:

windowStart = lastPos + 1

seenZero = False

if nums[windowEnd] == 0:

seenZero = True

lastPos = windowEnd

windowEnd += 1

maxOnes = max(maxOnes, windowEnd - windowStart)

return maxOnes

1. Implement a trie

Basic, add marker, a trienode!

class TrieNode:

def \_\_init\_\_(self, value='#'):

self.value = value

self.children = [None]\*26

self.MARKER = False

def \_\_str\_\_(self):

values = []

for c in self.children:

if c :

values += c.value

return f'{self.value, values}'

class Trie:

def \_\_init\_\_(self):

self.root = TrieNode()

def insert(self, word: str) -> None:

curr = self.root

for c in word:

index = ord(c) - ord('a')

if not curr.children[index]:

curr.children[index] = TrieNode(c)

curr = curr.children[index]

curr.MARKER = True

def search(self, word: str) -> bool:

curr = self.root

for c in word:

index = ord(c) - ord('a')

if not curr.children[index]:

return False

curr = curr.children[index]

if not curr.MARKER:

return False

return True

def startsWith(self, prefix: str) -> bool:

curr = self.root

for c in prefix:

index = ord(c) - ord('a')

if not curr.children[index]:

return False

curr = curr.children[index]

return True

# Your Trie object will be instantiated and called as such:

# obj = Trie()

# obj.insert(word)

# param\_2 = obj.search(word)

# param\_3 = obj.startsWith(prefix)

1. DFS on Trie gives preorder traversal and is sorted!!

def startsWith(self, prefix: str) -> bool:

curr = self.root

result = []

for c in prefix:

index = ord(c) - ord('a')

if not curr.children[index]:

return result

curr = curr.children[index]

#at the node where the collection of children starts

def dfsWithPrefix(node, word):

if len(result) == 3:

return result

if node.MARKER:

result.append(word)

for i in range(ord('a'),ord('z')+1):

index = i - ord('a')

if not node.children[index]:

continue

dfsWithPrefix(node.children[index],word+chr(i))

dfsWithPrefix(curr,prefix)

return result

1. Very Simple Top K Frequent

class Solution:

def topKFrequent(self, words: List[str], k: int) -> List[str]:

encoding = Counter(words)

heap = []

for w in encoding:

heap.append((-encoding[w],w))

heapq.heapify(heap)

result = []

while k>0:

result.append(heapq.heappop(heap)[1])

k-=1

return result

1. Topo Sort! Super simple

Use concept of indegree to form a valid order

class Solution:

def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[int]:

adjlist = collections.defaultdict(list)

indegree = [0]\*numCourses

if not prerequisites:

return list(range(numCourses))

for nextCourse, prevCourse in prerequisites:

adjlist[prevCourse].append(nextCourse)

indegree[nextCourse] += 1

queue = deque()

for course in range(numCourses):

if indegree[course] == 0:

queue.append(course)

result = []

while queue:

node = queue.popleft()

print(node)

result.append(node)

for v in adjlist[node]:

indegree[v] -= 1

if indegree[v] == 0:

queue.append(v)

if any(indegree):

return []

return result

1. Min Height Trees

Scary fucking question, refer to the solution and so is alien dictionary

1. When you have more brain power, understand solutions for K Closest points.
2. Meeting Rooms – 2

Super simple, min heap of end times,

Pop it when you have to reuse a room and if not add it to the heap!!

class Solution:

def minMeetingRooms(self, intervals: List[List[int]]) -> int:

if not intervals:

return 0

intervals.sort()

minHeap = [intervals[0][1]]

heapq.heapify(minHeap)

intervals = intervals[1:]

while intervals:

currStart,currEnd = intervals[0]

if currStart >= minHeap[0]:

#reuse and update ending time

heapq.heappop(minHeap)

heapq.heappush(minHeap,currEnd)

intervals = intervals[1:]

return len(minHeap)

1. Construct Binary tree from inorder and preorder traversals

TRICK: Split the node in inorder by preorder index.

class Solution:

def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:

preorderIndex = 0

def buildRecursive(left, right):

nonlocal preorderIndex

nonlocal preorder

if left > right:

return None

value = preorder[preorderIndex]

root = TreeNode(value)

preorderIndex += 1

#O(n), can build a hashmap

index = inorder.index(value)

root.left = buildRecursive(left, index - 1)

root.right = buildRecursive(index + 1, right)

return root

return buildRecursive(0, len(preorder) - 1)

Time complexity – O(N)

1. LCA of a Binary Tree

Super simple, find the first split from the top

Return the root if its p or q

Else if you hit one first and didn’t find the other one and we for sure know it exists, it has to be in the subtree of another

class Solution:

def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':

def dfs(root,p,q):

if not root:

return

if root == p or root == q:

return root

r1 = dfs(root.left,p,q)

r2 = dfs(root.right,p,q)

if r1 and r2:

return root

return r1 or r2

return dfs(root,p,q)

1. Merge K Sorted Lists

Use a min heap

Add the node and index of the list

And keep incrementing the head

class Solution:

def mergeKLists(self, lists: List[Optional[ListNode]]) -> Optional[ListNode]:

n = len(lists)

if n == 0:f

return None

if n == 1:

return lists[0]

init = [(lists[i].val, i) for i in range(n) if lists[i]]

if not init:

return None

heapq.heapify(init)

result = ListNode()

head = result

while init:

value, index = heapq.heappop(init)

#index is the index of listnode

head.next = lists[index]

head = head.next

lists[index] = lists[index].next

#may be we push something

if not lists[index]:

continue

newValue = lists[index].val

if newValue < value:

#add this value to result and continue

head.next = lists[index]

lists[index] = lists[index].next

head = head.next

continue

heapq.heappush(init, (newValue,index))

return result.next

Complexity: O(Nlogk)

1. Backspace string compare

Intuition: Iterate in reverse and skip the non back space characters depending on the value of skip

TIP - Use itertools.zip\_longest

def backspaceCompare(self, s: str, t: str) -> bool:

def clean(s):

skip = 0

for c in reversed(s):

if c == '#':

skip += 1

elif skip:

skip -= 1

else:

yield c

return all(x == y for (x,y) in itertools.zip\_longest(clean(s),clean(t)))

1. Longest substring with atmost 2 distinct characters

Brilliant and simple logic

Use a simple hashmap to delete the LEAST RECENT CHARACTER

Such that left becomes that position + 1 for the next window!!!

def lengthOfLongestSubstringTwoDistinct(self, s: str) -> int:

left, right = 0,0

n = len(s) - 1

d = collections.defaultdict(int)

maxLength = float('-inf')

while right <= n:

d[s[right]] = right

right += 1

if len(d) == 3:

#Delete the least recent value!!

idx = min(d.values())

del d[s[idx]]

left = idx + 1

maxLength = max(maxLength, right - left)

return maxLength

1. Character Replacement

<https://leetcode.com/problems/longest-repeating-character-replacement/discuss/91301/Awesome-python-solution>

Too simple, almost embarrassing!

class Solution:

def characterReplacement(self, s: str, k: int) -> int:

left, right = 0,0

n = len(s) - 1

c = collections.Counter()

maxLength = float('-inf')

while right <= n:

c[s[right]] += 1

currWindowSize = right - left + 1

if currWindowSize - c.most\_common(1)[0][1] > k:

c[s[left]] -= 1

left += 1

maxLength = max(maxLength, right - left + 1)

right += 1

return maxLength

1. Count nodes in a binary tree

Use binary search by first converting the num to bin representation

Easy traversal to check if the node exists and then bin search from last level

2\*d to (2\*(d+1) -1)

def countNodes(self, root: Optional[TreeNode]) -> int:

def depth(root):

d = 0

while root.left:

root = root.left

d += 1

return d

def exists(idx,d,root):

for n in bin(idx)[3:]:

if n == "0":

root = root.left

elif n == "1":

root = root.right

if not root:

return False

return True

if not root:

return 0

d = depth(root)

if d == 0:

return 1

left, right = 2\*\*d, 2\*\*(d+1) - 1

if exists(right,d,root):

return right

while left + 1 < right:

pivot = (left + right) // 2

if exists(pivot,d,root):

left = pivot

else:

right = pivot

return left

1. Evaluate Division

Simple graph question

def calcEquation(self, equations: List[List[str]], values: List[float], queries: List[List[str]]) -> List[float]:

adjlist = defaultdict(list)

visited = set()

for eq,v in zip(equations,values):

x,y = eq

adjlist[x].append((y,v))

adjlist[y].append((x,1.0/v))

result = []

def dfs(src, target, curr):

if src == target:

result.append(curr)

return True

for dst,weight in adjlist[src]:

if dst in visited:

continue

visited.add(dst)

curr = curr\*weight

if dfs(dst,target,curr):

return True

curr = curr/weight

return False

for src,dst in queries:

if src not in adjlist or dst not in adjlist:

result.append(-1.0)

continue

visited.add(src)

if not dfs(src,dst,1.0):

result.append(-1.0)

visited.clear()

return result

1. Decode String

Simply use a stack, and regex to match the number

def decodeString(self, s: str) -> str:

stack = []

result = ""

i,n = 0,len(s)

while i < n:

if s[i].isdigit():

match = re.search(r'\d+',s[i:]).group()

i += len(match)

stack.append(match)

continue

if s[i] == ']':

tmp = ""

while stack:

char = stack.pop()

if char.isalpha():

tmp = char + tmp

if char.isdigit():

tmp = int(char)\*tmp

stack.append(tmp)

break

else:

stack.append(s[i])

i+=1

return ''.join

Time Complexity –

0(MaxK^countk x n )

Max k is the maximum value of k and countk is the number of times K is nested and n is the maximum length of encoded string

For, s = 10[ab10[cd]]10[ef], time complexity would be roughly equivalent to 10\*ab\*10\*cd + 10\*ef = 10\*2 x 2

Space Complexity:

The maximum stack size would be equivalent to the sum of all the decoded strings in the form k[nk[n]]

1. Remove stones DFS

Don’t CREATE A O(N^2) adjacency list

Instead create a rows/cols adj list

def removeStones(self, stones: List[List[int]]) -> int:

if len(stones) <= 1:

return 0

rows,cols = defaultdict(list), defaultdict(list)

for (i,j) in stones:

rows[i].append(j)

cols[j].append(i)

def dfs(x,y):

nonlocal visited

nonlocal removed

visited.add((x,y))

for p in rows[x]:

if (x,p) not in visited:

dfs(x,p)

#backtracking

removed.add((x,p))

for q in cols[y]:

if (q,y) not in visited:

dfs(q,y)

#backtracing

removed.add((q,y))

visited = set()

removed = set()

for (x,y) in stones:

if (x,y) not in visited:

dfs(x,y)

return len(removed)

1. Is Majority element in a sorted array

Do not use the voting algorithm, the array is sorted, use binary search twice to find the beginning and ending of the target element in left and right half of the array. Implement bisect\_left yourself.You can just search for target+1 for bisect\_right.

class Solution:

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

#use bin search in left and in right

def bisect\_left(arr,element):

left, right = 0, len(arr)

while left < right:

mid = (left + right)//2

if arr[mid] < element:

left = mid + 1

else:

right = mid

return left

left, right = 0, len(nums) - 1

mid = (left + right)//2

if nums[mid] != target:

return False

p1 = bisect\_left(nums,target)

p2 = bisect\_left(nums,target+1)

return p2 - p1 > len(nums)//2

Time Complexity:

O(Logn)

Split Array Largest Sum

Binary Search logic

1. What is the min that the ship should carry in a day? = It should carry atleast the max in the array. Meaning, if the max weight is 10, and the ship can carry only 9, that doesnt make sense.
2. Now what is the max the ship can carry? = The sum of all the weights. You can think of this as, if you get an array but the no of days is D =1 , ie, the ship has to carry all the weight in one day, then you dont have an option except to carry all the weight. So the max = sum of items in the array.
3. No logically speaking the min weight the ship can distribute is somewhere in between this min and max. right?
4. Therefore you have a lower bound and an upper bound, and you have to find the ideal weight the ship can carry. This is quite a good clue to lean towards binary search because you can get to the answer in log(n) time.

def shipWithinDays(self, weights: List[int], days: int) -> int:

#same as split array largest sum

l,h = max(weights),sum(weights)

while l < h:

capacity = (l+h)//2

#if I can fulfill this capacity within days

cum = 0

d = 1

for w in weights:

cum += w

if cum > capacity:

cum = w

d += 1

if d > days:

l = capacity + 1

break

else:

h = capacity

return l

1. K Closest Elements

Binary search, key is to initialize left and right window

L= Bisect\_left – 1

R = L + 1

Return L+1:R

class Solution:

def findClosestElements(self, arr: List[int], k: int, x: int) -> List[int]:

def bisect\_left(A,x):

l,h = 0, len(A)

while l < h:

mid = (l+h)//2

if A[mid] < x:

l = mid + 1

else:

h = mid

return l

L = bisect\_left(arr,x) - 1

H = L + 1

while H - L - 1 < k:

if L < 0:

H += 1

continue

if H == len(arr) or abs(arr[L] - x) <= abs(arr[H] - x):

L -= 1

else:

H += 1

return arr[L+1:H]

1. Redundant connection 2

There are two cases for the tree structure to be invalid.

1) A node having two parents;

including corner case: e.g. [[4,2],[1,5],[5,2],[5,3],[2,4]]

2) A circle exists

simple, count incoming edges

if a node has two incoming edges, skip those edges one by one till no cycle remains

<https://leetcode.com/problems/redundant-connection-ii/discuss/108070/Python-O(N)-concise-solution-with-detailed-explanation-passed-updated-testcases>

1. Maximum Size Subarray Sum Equals

Not a sliding window question, it’s a prefix sum question

Calculate running prefix sum, if the diff of two i.e prefix\_sum – k is in hashmap, return the diff of indices

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

prefix\_sum = 0

longest\_subarray = 0

indices = dict()

for i, num in enumerate(nums):

prefix\_sum += num

if prefix\_sum == k:

longest\_subarray = i+1

if prefix\_sum - k in indices:

print(i, indices[prefix\_sum -k ])

longest\_subarray = max(longest\_subarray, i - indices[prefix\_sum - k])

if prefix\_sum not in indices:

indices[prefix\_sum] = i

return longest\_subarray

1. Shortest Distance Color

Make a hashmap of indices for each color and do binary search on the index

Use bisect\_left but remember that sometimes it will give you the (index+1) if there is no exact match so pick the min between the index-1 and index

def shortestDistanceColor(self, colors: List[int], queries: List[List[int]]) -> List[int]:

hashmap = collections.defaultdict(list)

for i,c in enumerate(colors):

hashmap[c].append(i)

result = []

for index,color in queries:

if color not in hashmap:

result.append(-1)

continue

indices = hashmap[color]

if index <= indices[0]:

result.append(indices[0] - index)

elif index > indices[-1]:

result.append(index - indices[-1])

else:

closest\_index = bisect.bisect\_left(indices, index)

result.append(min(abs(index - indices[closest\_index - 1]), abs(index - indices[closest\_index])))

return result

1. Max Sliding Window

Use a deque, if the index is i-k pop left, if the currnt element is greater than the elements before it remove those and if I >= k-1, start appending top of queue to result

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

queue=deque()

result = []

for i in range(len(nums)):

if queue and queue[0] == i-k:

queue.popleft()

while queue and nums[i] >= nums[queue[-1]]:

queue.pop()

queue.append(i)

if i >= k - 1:

result.append(nums[queue[0]])

return result

1. Random pick with weight (Google)

def \_\_init\_\_(self, w: List[int]):

self.prefix\_sum = [0] \* (len(w))

self.prefix\_sum[0] = w[0]

for i in range(1,len(w)):

self.prefix\_sum[i] = self.prefix\_sum[i-1] + w[i]

self.sum = sum(w)

def pickIndex(self) -> int:

target = random.random() \* self.sum

index = bisect.bisect\_left(self.prefix\_sum, target)

return index

Think that if we had an array [1,2,3,4,3]. Normal random pickIndex would pick any index from 0 to 4 with equal probability. But we want that index=1 is picked by 2/13 probability, index=0 with 1/13 probability and so on. (13 is sum of weights). To ensure that one way to think of it if we make a larger array (of size 13) where the values are the indices such that index i is repeated w[i] times then if we do a normal rand on this array then index 0 to 12 will be picked randomly with equal probability. 13 index array -> [0, 1,1, 2,2,2, 3,3,3,3, 4,4,4]. So there is a 3/13 chance of picking 2 as 2 is repeated thrice in the new array.

Now instead of actually constructing this 13 index array, we just know the range of the index of the 13 index array where value = i. Eg:

* for index=0, range is {0,0}
* index =1, range of indices of the new array is {1,2}
* index=2, range={3,5}
* index=3, range ={6,9}
* index = 4, range = {10,12}

In other words,

* index=0, range is <1
* index=1, range is <3
* index=2, range is <6
* index = 3, range is < 10
* index = 4, range is < 13

If you notice the above numbers 1,3,6,10,13 - they are cumulative sum.  
The reason this happens is because for every range: right = left + (w[i] - 1) and left is (prev right+1). So if we substitute 2nd equation into 1st. right = (prev right)+w[i]; i.e. keep adding prev sum to current weight.

1. Find leaves of binary tree using topological sort

Tree is a dag, USE TOPO SORT ON IT for outdegree = 0 that is leaves

class Solution:

def findLeaves(self, root: Optional[TreeNode]) -> List[List[int]]:

result = []

outdegree = collections.defaultdict(list)

def fill\_outdegree(root,parent):

if not outdegree[root]:

outdegree[root] = [0,parent]

if not root.left and not root.right:

return

if root.left:

fill\_outdegree(root.left, root)

outdegree[root][0] += 1

if root.right:

fill\_outdegree(root.right, root)

outdegree[root][0] += 1

fill\_outdegree(root,None)

queue = deque()

for k in outdegree:

if outdegree[k][0] == 0:

queue.append([k,outdegree[k][1]])

while queue:

size = len(queue)

result.append([])

for \_ in range(size):

node,parent = queue.popleft()

result[-1].append(node.val)

if not parent:

break

outdegree[parent][0] -= 1

if not outdegree[parent][0]:

queue.append([parent, outdegree[parent][1]])

return result

<https://leetcode.com/discuss/interview-experience/2016202/Top-LeetCode-posts-that-you-cannot-miss>

1. Find duplicates in the array

Simple logic, negate the array element to mark it as visited

class Solution:

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

result = []

for i in range(len(nums)):

index = abs(nums[i]) - 1

if nums[index]< 0:

result.append(abs(nums[i]))

else:

nums[index] \*= -1

return result

1. Bisect Left and Bisect right

def bisect\_right(a, x, lo=0, hi=None):

"""Return the index where to insert item x in list a, assuming a is sorted.

The return value i is such that all e in a[:i] have e <= x, and all e in

a[i:] have e > x. So if x already appears in the list, a.insert(x) will

insert just after the rightmost x already there.

Optional args lo (default 0) and hi (default len(a)) bound the

slice of a to be searched.

"""

if lo < 0:

raise ValueError('lo must be non-negative')

if hi is None:

hi = len(a)

while lo < hi:

mid = (lo+hi)//2

if x < a[mid]: hi = mid

else: lo = mid+1

return lo

def bisect\_left(a, x, lo=0, hi=None):

"""Return the index where to insert item x in list a, assuming a is sorted.

The return value i is such that all e in a[:i] have e < x, and all e in

a[i:] have e >= x. So if x already appears in the list, a.insert(x) will

insert just before the leftmost x already there.

Optional args lo (default 0) and hi (default len(a)) bound the

slice of a to be searched.

"""

if lo < 0:

raise ValueError('lo must be non-negative')

if hi is None:

hi = len(a)

while lo < hi:

mid = (lo+hi)//2

if a[mid] < x: lo = mid+1

else: hi = mid

return lo

1. Distance k from binary tree

Form a dict with parent node, then do BFS from target

def distanceK(self, root: TreeNode, target: TreeNode, k: int) -> List[int]:

parent\_dict = collections.defaultdict(TreeNode)

def dfs(root, parent):

if not root:

return

parent\_dict[root] = parent

dfs(root.left, root)

dfs(root.right, root)

dfs(root,None)

queue = deque([(target,0)])

seen = set()

seen.add(target)

result = []

while queue:

node,dist = queue.popleft()

if dist == k:

result.append(node.val)

continue

for n in node.left,node.right,parent\_dict[node]:

if n and n not in seen:

seen.add(n)

queue.append((n,dist+1))

return result

1. Vertical order!!

Beautiful BFS solution, store the col for each node and then find the min and max range and return it

def verticalOrder(self, root: Optional[TreeNode]) -> List[List[int]]:

if not root:

return []

row\_col = collections.defaultdict(list)

min\_col,max\_col = math.inf,-math.inf

def fill\_row\_col(root):

nonlocal row\_col,min\_col,max\_col

queue = deque([(root,0)])

while queue:

node, loc = queue.popleft()

row\_col[loc].append(node.val)

max\_col = max(max\_col, loc)

min\_col = min(min\_col,loc)

if node.left:

queue.append((node.left, loc-1))

if node.right:

queue.append((node.right, loc+1))

fill\_row\_col(root)

result = []

for i in range(min\_col, max\_col+1):

result.append(row\_col[i])

return result

1. Word Distance

Simple and sweet, init distance as -1,-1 and keep a track of min distance

def shortestDistance(self, wordsDict: List[str], word1: str, word2: str) -> int:

w1,w2 = -1,-1

minDiff = math.inf

for i,w in enumerate(wordsDict):

if w == word1:

w1 = i

if w == word2:

w2 = i

if w1 != -1 and w2 != -1:

minDiff = min(minDiff, abs(w1 - w2))

return minDiff

1. Group strings

Group by hash, you can either create a tuple

For each string create a tuple.

mp = defaultdict(list)

for s in strings:

key = tuple((ord(c)-ord(s[0]))%26 for c in s)

mp[key].append(s)

return mp.values()

Or do this

def groupStrings(self, strings: List[str]) -> List[List[str]]:

def shift\_string(s):

if s[0] == "a":

return s

shifted = ""

for c in s:

k = (ord(c) - ord(s[0])) % 26

shifted += chr(k + ord('a'))

return shifted

groups = collections.defaultdict(list)

for s in strings:

hash\_key = shift\_string(s)

groups[hash\_key].append(s)

return groups.values()

1. Tree Diameter

Use topological sort!! If you can find a centroid, then you can find radius

Radius \* 2 = diameter

def treeDiameter(self, edges: List[List[int]]) -> int:

adjlist = collections.defaultdict(list)

for e1,e2 in edges:

adjlist[e1].append(e2)

adjlist[e2].append(e1)

n = len(edges)

queue = deque()

for v in adjlist:

if len(adjlist[v]) == 1:

queue.append((v,1))

diam = 0

remaining = n+1

while remaining > 2:

remaining -= len(queue)

next\_round = deque()

while queue:

node,dist = queue.popleft()

diam = max(diam,dist)

for v in adjlist[node]:

adjlist[v].remove(node)

if len(adjlist[v]) == 1:

next\_round.append((v,dist+1))

queue = next\_round

return diam\*2 if remaining == 1 else diam\*2 + 1

1. Subarrays with k different integers

First find atmost(k)

Important is the length which is right – left which will find all subarrays with length atmost k

Intuition:

The intuition behind atMostK(A, K) - atMostK(A, K - 1);:

This problem is a hard ask, until you realize that we've actually solved this problem before in other sliding window problems. Generally, the sliding window problems have some kind of aggregate, atMost k, largest substring, min substring with k etc. They're always "given an array or string, find some computed sub problem" value.

So how do we use this our advantage? Well, the ask: different integers in that subarray is exactly K is exactly this. We can rewrite the problem to something like this:

subArrayExactlyK = subArrayAtMostK - subArrayAtMostK - 1. This is basically saying, give me the amount of subarrays we can form with *at least* 3, and give me the amount of subarrays we can form with *at least* 2, and the diff between the two will be *only* subarrays at 3 (since we have eliminated everything 2 and under).

Example:  
Input: A = [1,2,1,2,3], K = 2  
Output: 7

subArrayAtMostK = 12  
if k = 2, there are 12 possible subarrays that are at least 2 values.  
This is the array possibilities we create, where the count is the end - start (see below code example), since a sub problem will contribute to the overall amount of possibilities. You see below for the AtMostK problems if this concept is confusing. The other trick here is that in other atMostK problems, they ask for length, but length can also be a substitute for amount of sub problems, since the length of any given range, say 1212, also constitutes 4 different subarray possibilities.  
[1, 12, 121, 1212, 23]

1 + 2 + 3 + 4 + 2 = 12

subArrayAtMostK - 1 = 5  
since k = 1, every subarray is a single element, so the length of the array. There are by definition, only 5 different subarrays that can be formed.  
[1, 2, 1, 2, 3]  
1 + 1 + 1 + 1 + 1 = 5

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

def atmost(k,nums):

left = 0

visited = Counter()

length = 0

for right in range(len(nums)):

visited[nums[right]] += 1

while len(visited) > k:

if visited[nums[left]]:

visited[nums[left]] -= 1

if not visited[nums[left]]:

del visited[nums[left]]

left += 1

length += right - left

return length

return atmost(k,nums) - atmost(k-1,nums)

1. Minimum Window Substring

Super easy,use the logic count = 0 when the count of every number becomes 0 in the target string. Shrink the window to get the minimum valid substring and then when count becomes 1, go looking ahead for the next candidate

class Solution:

def minWindow(self, s: str, t: str) -> str:

src = Counter(t)

dst = src.copy()

found = len(src)

l,r = 0,0

sol\_l, sol\_r = 0,0

ans = math.inf

while r < len(s):

if s[r] in src:

dst[s[r]] -= 1

if dst[s[r]] == 0:

found -= 1

r += 1

if found == 0:

while found == 0:

if r - l < ans:

ans = r - l

sol\_l,sol\_r = l,r

if s[l] in dst:

dst[s[l]] += 1

if dst[s[l]] == 1:

found += 1

l += 1

return s[sol\_l:sol\_r]

1. Max Profit assigning work

My solution is simple

Make a mapping, then keep a max and then use binary search on the difficulty array

class Solution:

def maxProfitAssignment(self, difficulty: List[int], profit: List[int], worker: List[int]) -> int:

mapping = collections.defaultdict(int)

for d,p in zip(difficulty,profit):

mapping[d] = max(mapping[d],p)

difficulty.sort()

np = [mapping[d] for d in difficulty]

for i in range(1,len(np)):

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

max\_profit = 0

for ability in worker:

index = bisect.bisect\_right(difficulty, ability)

if not index:

continue

max\_profit += np[index-1]

return max\_profit

1. Find visible points

Basic logic is simple, calculate all angles and add replicas of 360 + x

Maintain a circular sliding window starting from smallest element

So slide the window from smallest to right, till its less than angle

And slide if the distance becomes > angle

SORT the angles!!

class Solution:

def visiblePoints(self, points: List[List[int]], target: int, location: List[int]) -> int:

angles = []

max\_points = 0

same\_loc = None

count = 0

lx,ly = location

for i in range(len(points)):

x,y = points[i]

if lx == x and ly == y:

count += 1

continue

angle = math.degrees(math.atan2(y-ly,x-lx))

angles.append(angle)

angles.sort()

angles += [(360 + x) for x in angles]

l = 0

max\_len = 0

for r in range(len(angles)):

while l < len(angles) and (angles[r] - angles[l]) > target:

l += 1

max\_len=max(max\_len,r - l + 1)

return max\_len + count

1. Step-By-Step-Directions-From-A-Binary-Tree-Node-To-Another

Either do it via BFS with VISITED! Or find LCA and then just do lca to start and lca to end and convert all lca\_to\_start to “U”

For LCA:

Return the node p or q whichever is found.

If both are found, the immediate root is LCA else whichever is found is the LCA

Remember that!

# Definition for a binary tree node.

# class TreeNode:

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

# self.val = val

# self.left = left

# self.right = right

class Solution:

def getDirections(self, root: Optional[TreeNode], startValue: int, endValue: int) -> str:

def dfs(root,target,path=[]):

if not root:

return ""

if root.val == target:

return ''.join(path[:])

ans = ""

if root.left:

path.append("L")

ans += dfs(root.left,target,path)

path.pop()

if root.right:

path.append("R")

ans += dfs(root.right,target,path)

path.pop()

return ans

def lca(root,start,end):

if not root:

return None

if root.val == start or root.val == end:

return root

r1 = lca(root.left,start,end)

r2 = lca(root.right,start,end)

if r1 and r2:

return root

return r1 or r2

lca\_node = lca(root,startValue,endValue)

lca\_start = dfs(lca\_node,startValue)

lca\_end = dfs(lca\_node,endValue)

return ''.join(["U"]\* len(lca\_start)) + lca\_end

1. Maximum Number of Points with Cost

One of the hardest questions ever, involves doing minimum path falling sum and best sightseeing pair. Uses concept of left and right dp

Its basically a maximization problem in left and right dps

import numpy as np

class Solution:

def maxPoints(self, points: List[List[int]]) -> int:

R,C = len(points), len(points[0])

t = [[0 for \_ in range(C)] for \_ in range(R)]

for j in range(0,C):

t[0][j] = points[0][j]

left = [-math.inf for \_ in range(C)]

right = [-math.inf for \_ in range(C)]

for i in range(1,R):

#case 1: j > k : t(i-1)(k) + k + points(i,j) - j

left[0] = t[i-1][0]

for j in range(1,C):

left[j] = max(left[j-1], t[i-1][j] + j)

#case 2:j < k : t(i-1)(k) - k + points(i,j) + j

right[C-1] = t[i-1][C-1] - (C-1)

for j in range(C-2,-1,-1):

right[j] = max(right[j+1], t[i-1][j] - j)

for j in range(C):

t[i][j] = max(left[j] - j, right[j] + j) + points[i][j]

return np.max(t[R-1])

1. Campus Bikes 2

The state is the bikes assigned in bitmask!!!