NeetCode Blind 75

Questions answered in Python

NOTE: This document is incomplete, but is getting updated weekly. It also has some personal notes at the end I used to prepare for my Amazon interview.

Resources

- Coding interviews are harder than the software engineering job… AlgoMonster's structured curriculum helps you master the key patterns fast so you can land a six-figure dev job with confidence.
- Let Massive's AI auto-apply to 100+ dev jobs every day so you can focus on interview prep:
- Zero To Mastery courses teach the exact skills recruiters want and get you job-ready fast:
- TLDR sends a free, bite-sized tech roundup to your inbox each morning so you stay ahead in minutes

= NeetCode code explanation required
= Hint required, or neetcode theory explanation required

Arrays & Hashing

1. Contains Duplicate:
    a. Sort a list in python with .sort()
    b. Use set() for unique values, and .add to add new values -> exampleList = set()
    c. Solution: Use set to keep track of seen numbers, if a num in nums is in seen, return True since duplicate. Else add that num to seen numbers, return false if all nums are unique.
2. Valid Anagram:
    a. Create a dictionary in python using {} -> dic = {}
    b. Get from dictionary using -> example = dic.get(item, 0), where 0 is a default value if item not in dic
    c. Set a value in dictionary by doing -> dic[example] = 1
    d. Check if two dics are equal using == -> dic1 == dic2
    e. Solution: Create two dictionaries to store letters and count of letters in respective words, and check if the two dictionaries are equal. If they are equal, return True, else return False.
3. Two Sum:
    a. Get index of an element in a list using .index() -> fruits.index("apple")
    b. Slicing list in python: new_list = original_list[start:end:step]
        i. start: The index at which the slicing should begin (inclusive).
        ii. end: The index at which the slicing should end (exclusive).
        iii. step: The increment value for selecting elements (optional).

    c. Solution: For i in range len(nums), calculate total - current number. If that remainder is in the rest of the list (nums[i+1:], return i and the index of the second number (nums[i+1].index(remainder) + i + 1)
4. Group Anagrams:
    a. List in python = [] -> list = []
    b. Add to list using .append(item) -> list.append(2)
    c. Solution: Initialize result dictionary, and list to store all possible dictionaries. Outer loop through all all words in strs. Initialize dic to store current words dictionary. Get dictionary of word. If dictionary not already in list of dictionaries, append dic to dictionaries list. Get index of dic in dictionaries, and get current array of words in result dictionary at that index. Append word to that array, and set this new array back to dictionary value at the index. Return dictionary values after looping through all words.
5. Top K Frequent Elements
    a. To get max value from a list use max() -> maxVal = max(exampleList)
    b. To remove a value from a list use .remove(value) -> exampleList.remove(1)
    c. Solution: Create a dic to store count of all values, and populate this dictionary. Then store the keys and values into two lists respectively. For i in range 0->k, get the max element from the values dictionary (max count) and the index of this item in the list. This is where the ith most frequent item will show up in the keys list, so get that item and append it to the result list. Remove both the max count from values and the item from keys, return result at end of loop.
6. String Encode and Decode
    a. To convert to str use str(item)
    b. To convert string to int use int(item)
    c. To get length of string use len(string)
    d. Solution encode: Initialize empty result string, for each word in array: add length of word + "#" + word, return result
    e. Solution decode: Initialize empty list for result, and i = 0. While i < len(s): set temp = i, increment temp until s[temp] == "#". Then get the length by slicing s[i:temp] and convert to int. Then append to list: s[temp + 1: temp + 1 + length]. Set i = temp + 1 + length. Return result list after loop.
7. Product of Array Except Self
    a. Hint: Keep track of prefix and suffix
    b. Initialize an empty array of a fixed size -> prefix = [0] * len(nums)
    c. To reverse a list or string use [::-1]
    d. Solution: Create 3 arrays of length of nums, called answer, prefix and suffix. Create a list reverse to store the reversed contents of num. Loop through i in range length of nums, and populate prefix, and suffix. If i = 0, prefix[i], suffix[i] = 1. Else prefix[i] = prefix[i-1] * nums[i-1], suffix[i] = suffix[i-1] * reversedNums[i-1]. Reverse the suffix list, then multiply prefix and suffix and store the result in the answers list.
8. Longest Consecutive List
    a. Solution: If length of nums is 0, return 0. Sort nums, initialize count and consecutive to 1. Loop through i in range length nums - 1. If nums[i] == nums[i+1] - 1, increment count, and consecutive is max of count and consecutive. Else if nums[i] == nums[i+1] continue.

Else consecutive is max of count and consecutive, then reset count to 1. Return consecutive.

Two Pointers

1. Valid Palindrome (Could not figure out two pointer method, although it is not necessary)
    a. To determine if a char is a letter use .isalpha() -> s.isalpha()
    b. Convert a string to lowercase using .lower() -> s = s.lower()
    c. NOTE: Alphanumeric characters include letters and numbers.
    d. Solution: Convert s to lower case, and create a variable to store a string. Loop through all characters in s, if it is a letter or digit, append to this new string. Return newString == newString[::-1]
2. (150) Two Sum II - Input Array Is Sorted
    a. Hint: Initialize left pointer at 0, right pointer at length of nums - 1
    b. Solution: Initialize left pointer at 0, right pointer at length of nums -1, and result array. While left < right, calculate numbers[left] + numbers[right]. If that value is = to the target, append left+1 and right+1 to the result and return. Else if result is > target, subtract one from right pointer. Else add one to the left pointer. Return result.
3. 3Sum
    a. NOTE: Solve 2 Sum first, and 2 Sum II - Input Array is Sorted
    b. Hint: If you fix x, the sum of y and z is a 2 sum problem
    c. Solution: Initialize a result list, and sort the nums. For i in range length of nums, create a list, rest, and make it everything else in nums from i onwards. Set the left pointer to 0 and the right pointer to the length of the rest list - 1. Perform the 2 sum approach, however this time if value is = to 0, create a list called temp, append values x, y, and z to it and append this list to the result list. Decrement right by 1. At the end of the loop, return result.
4. Container With Most Water
    a. Solution: Initialize a result value to 0, a length value to the length of height array, a left pointer to 0, and a right pointer to length - 1. While left < right, find the minimum height from either left or right in the height list. Calculate area with height and length, and if it is larger than result, change result to area. If height[left] < height[right, increase left by 1 and decrease length by 1. Else decrease right and length by 1. After loop return result.

Sliding Window

1. Best Time to Buy and Sell Stock
    a. Runtime issues
    b. Solution: Initiate a value to keep track of the profit at 0, a left pointer at 0, and a right pointer at 1. While left and right less than length of nums, calculate current profit. If current profit larger than profit, update profit. If numbers[left] > numbers [right], negative profit so update left pointer to be equal to right, and right pointer += 1. Regardless, update right pointer by 1. After loop, return profit.

2. Longest Substring Without Repeating Characters
    a. Solution: Initiate result variable to 0, left to 0, and right to 1. While left < length of s, and right <= length of s: Get the substring s[left:right]. If the length of this substring is greater than the result, update the result value to the length. If right is < length of s and s[right] is in the substring, shift the left pointer by adding 1 and the index of s[right] in the substring. Increment right by 1 regardless. After loop, return result.
3. Longest Repeating Character Replacement
    a. * Was really close to a solution but had an issue with specific test cases
        i. ABBB, code would "replace" with A's instead of B's
        ii. Got code to work after hint, but very inefficient so watched video
    b. Solution: Create a hashmap to store the count of each letter in the window. Initiate a result value to 1, and a left pointer to 0. Loop through for right in range length of s: get the maximum value in the dic, and the current window of the string. If the length of the window - max value - k <= 0, update result with the max value of result, or length of the window. Else, decrement the value of s[left] in the dictionary by 1, and shift left over by 1. Return result after the loop.
4. Minimum Window Substring
    a. Struggled on this question
    b. Solution: Initialize a value need to 0. Loop through all letters in t, increment need by 1 for each letter and store the count of letters into a dictionary dicT. Initialize a variable have and left to 0, as well as a dicS and a result to None. For right in range(len(s)): if the current letter at the right pointer is in t, then increment the count of the letter in dicS, and if this count is <= the count of the letter in dicT, increment have by 1. Still inside this if statement, while need == have: get the current substring. If result is still None or the length of current substring is < length of result, update result to current. If the letter at the left pointer in s is in t, do the same thing as previous but decrement the count of the letter in dicS by 1. If the value of the letter in dicS < dicT, then decrement have by 1. Regardless, increment left by 1. After all of this, if result is still None return "", else return result.

Stack

1. Valid Parentheses
    a. To pop last item in list in Python, use .pop() -> list.pop()
    b. Solution: Initialize a list called stack. Loop through each char in s: if the char is),], or } then return False if one of either len(stack) == 0 or stack.pop() != the opening bracket corresponding. Else append the char to the stack. After loop, if the length of the stack is > 0, return False. Else return True.

Binary Search

1. Find Minimum in Rotated Sorted Array
    a. Had to watch code solution

   b. Integer division in python with // -> mid = (left + right) // 2
   c. Solution: Initialize the result variable to nums[0], left to 0, and right to len(nums) - 1. While left < right: If nums[left] < nums[right] you know you are currently in the correct window, so update result to min(result, nums[left]). Still in the while loop, update mid to (left + right) // 2, and result to min(result, nums[min]) to cover the edge case where the mid point happens to be the minimum. Now, if nums[mid] >= nums[left] you know that the correct window is to the right of mid, so update left to mid + 1. Else, you know that the correct window is to the left of mid, so update right to mid - 1. After the loop, return result.
      i. 1,2,3,4,5 -> first loop check in right window, second loop result does not get updated, third loop result does not get updated
      ii. 5,1,2,3,4 -> First loop check in left window, second loop right window, third loop result is mid
      iii. 4,5,1,2,3 -> Result is midpoint in first loop
      iv. 3,4,5,1,2 -> first loop check in right window, second loop nums[left] < nums[right]
      v. 2,3,4,5,1 -> result was midpoint after 3rd loop
2. Search in Rotated Sorted Array
   a. Hint: Use similar binary search method as prev, but also check if target is in window
   b. Solution: Initialize left = 0, right = len(nums) - 1. While left <= right: if nums[left] == target or nums[right] == target return left or right. Calculate mid, if nums[mid] == target return mid. If nums[mid] >= nums[left]: if target > nums[mid] or nums[left] > target, target not in window so shift left to mid + 1, else right = mid- 1. Else: if target < nums[mid] or nums[right] < target, target not in window so shift right to mid - 1, else left = mid + 1. After loop return -1.

Linked List

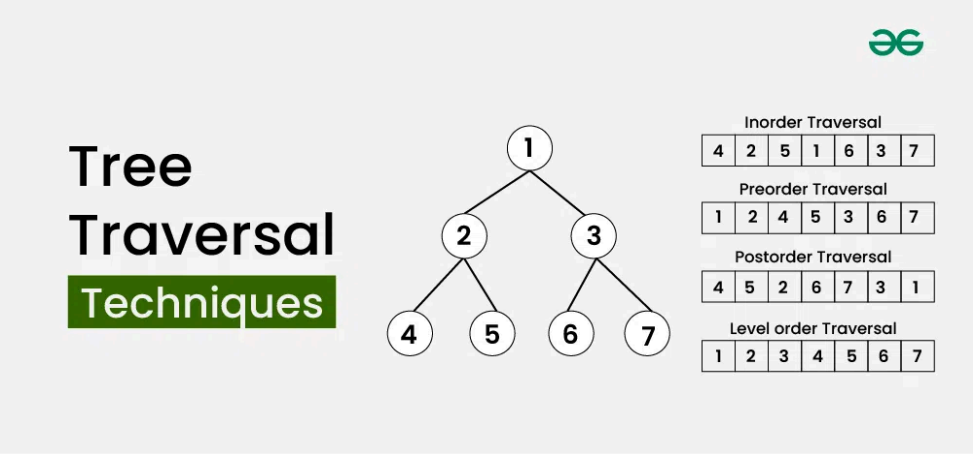
1. Reverse Linked List
   a. Had to briefly glance at code to correct my solution
   b. Hint: Keep track of prev
   c. Solution:
      i. Iterative: Initialize a prev value to None. While head: temp = head.next, head.next = prev, prev = head, head = temp. After loop return prev
      ii. Recursive: if head is none: return none. newHead = head, if head.next: newHead = self.reverseList(head.next), head.next.next = head. Head.next = None, return newHead.
2. Merge Two Sorted Lists
   a. Solution: Initialize result to ListNode(), and current = result. While list1 and list2: if list1.val < list2.val: current.next is a listNode of list 1 value and list1 = list1.next, else the same but for list2. Current = current.next. After while loop return result.next to skip the dummy 0.
3. Reorder List
   a. Hint: Use stack to keep track of values and pop alternatively between first and last value

   b. Solution: initialize an empty stack, and a pointer curr = head. While curr: stack.append(curr.val), curr = curr.next. After loop, initialize a boolean var first to False, recent head.next to None, pop the first item in the stack -> pop(0), and initialize a pointer temp = head. While the stack is not empty, if First is true pop the 0th item in the stack, else pop the last item in the stack, and make temp.next = ListNode(val) and temp = temp.next
4. Remove Nth Node From End of List
   a. Hint: Use two pointer technique, and add a dummy node to the beginning of head.
   b. Solution: Add a dummy node to head. Make right = head, and while n > 0 and right: n -= 1, right = right.next. Then make left = dummy, and while right: left = left.next, right = right.next. Now we have the node before the node we want to delete in left, so do: left.next = left.next.next. Return dummy.next to remove the dummy node.
5. Linked List Cycle
   a. Note: Got solution immediately but in an inefficient way
   b. Inefficient solution: Initialize a list to keep track of all visited nodes. Curr = head, while curr: if curr in visited list, return True, else visited.append(curr), curr = curr.next. Return False after loop.
   c. Proper solution: Initialize a variable slow and a variable head both equal to head. While fast and fast.next: fast = fast.next.next, slow = slow.next, if slow == fast: return True. After loop, return False.
6. Merge K Sorted Lists
   a. Hint: Merge sort where you go through in log(n) time complexity
   b. Solution: Define a function mergeList which takes two lists and merges them like the previous leetcode easy. In the main function, if lists is None or of length 0 return None. While len(lists) > 1: create a list mergedLists = []. For i in range(0, len(lists), 2): list1 = lists[i], list2 = lists[i+1] if not out of bounds, else it is None. Do the mergeList for these two Lists and then append to the mergedLists. After the for loop, lists = mergedList. Return lists[0] after while loop.

Trees

1. Invert Binary Tree
   a. Understood pseudocode immediately, but did not know how to implement so had to watch Neetcode video
   b. Hint: recursively solve each subtree
   c. Solution: define a base case if root is None, return None. Then swap the left and right by storing root.left in a temp variable, setting root.left = root.right, and root.right = temp. Then recursively call on root.left, then on root.right. Finally return root.
2. Maximum Depth of Binary Tree
   a. Use recursion
   b. Solution: Base case: If root is None return 0. Otherwise return 1 + max(self.maxDepth(root.left), self.maxDepth(root.right))
3. Same Tree
   a. Make sure to use .val to compare the values instead of direct comparisons

    b. Solution: Base cases: If p is None and q is None return True, elif p is None or q is None return False, elif p.val != q.val return False. Otherwise return (self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right))
4. Is Subtree
    a. Used ChatGPT to give me small hint
    b. Hint: Create separate function to check if is same Tree
    c. Hint 2: Recursively solve, return statement is an or with root.left, subRoot and root.right, subRoot
    d. Solution: Create separate function to check if is same tree recursively (see prev.). Base cases: root is None and subRoot is None return True, elif root is None or subRoot is None return False, elif self.sameTree(root, subRoot) return True. Otherwise return (self.isSubtree(root.left, subRoot) or self.isSubtree(root.right, subRoot))
5. Lowest Common Ancestor of a Binary Search Tree
    a. Had to watch Neetcode video explanation but it was incredibly easy to implement after getting the logic behind it
    b. Hint: Check values with p.val and q.val
    c. Solution (recursive): If root is None, return root. Elif root.val > p.val and q.val return self.LCA(root.left, p, q). Elif root.val < p.val and q.val return self.LCA(root.right, p, q). Else return root.
6. Binary Tree Level Order Traversal
    a. Could not understand how to approach this problem
    b. Hint: Use BFS (queue)
    c. Had to watch Neetcode video to break down the code for a BFS solution
    d. Solution: Create a list result to store the lists of values at each level, and a queue to perform the bfs. Append root to the queue. While queue: length = len(queue) (used to make sure only going through one level), values = [], for i in range (length): curr = queue.pop(0), if curr: values.append(curr.val), queue.append(left), queue.append(right). After for loop, if values is not empty, append values to result. After while loop return result.
7. Validate Binary Search Tree
    a. Was failing an edge case where the subtree is valid but not the entire tree
    b. Hint: Create a helper function with left, and right values
    c. Solution: Create a helper function valid(root, left, right): if root is None return True, if root.val <= left or root.val >= right return False, return valid(root.left, left, root.val) and valid(root.right, root.val, right). Return valid(root, float("-inf"), float("inf")).
8. Kth Smallest Element in BST
    a. Understood logic of solution immediately, had to glance at code to understand
    b. Solution: Create a stack to keep track of dfs, a list to store values, and sett a curr node = root. While dfs or curr: while curr: dfs.append(curr), curr = curr.left ; after while loop, curr = dfs.pop(), values.append(curr.val), curr = curr.right ; return values[k-1]
9. Construct Binary Tree from Preorder and Inorder Traversal
    a. Tree traversals:



i.
ii. Preorder: root node first, then left, then right
iii. Inorder: left, then node, then right
iv. Solution:
    1. Base case: if not preorder and not inorder: return None
    2. Root = TreeNode(preorder[0])
    3. Mid = inorder.index(preorder[0]) // index of mid point
    4. Root.left = self.buildTree(preorder[1:mid+1], inorder[:mid]
    5. Root.right = self.buildTree(preorder[mid+1:], inorder[mid + 1:])
    6. Return root

10. Binary Tree Maximum Path Sum
    a. The idea is to go through each node in a DFS approach, and return the maximum value if you do NOT split paths (i.e. only one of left or right). We then also need to update res with the max of res, and current node.val + the left or right path sum
    b. Solution: Define a global variable self.res = 0 to store the result. Define a recursive function dfs(root)
        i. If root is None: return 0
        ii. leftMax = max(0, dfs(root.left))
        iii. rightMax = max(0, dfs(root.right))
        iv. Self.res = max(self.res, root.val + leftMax + rightMax)
        v. Return max(root.val, root.val +leftMax, root.val + rightMax)
    c. Call the dfs: dfs(root)
    d. Return self.res
11. Serialize and Deserialize Binary Tree
    a. String example: 12NN34NN5NN
    b. Serialize:
        i. Got it without help, but had to tweak it to use an array to handle the case where a number had more than one digit
        ii. Solution: Initialize a list res = [], and a stack = [] to keep track of the dfs. Stack.append(root). While stack: curr = stack.pop(). If curr is none, res.append('N'), else stack.append(str(curr.val)). stack.append(curr.right), stack.append(curr.left). After a while loop, return ",".join(res)
    c. Deserialize:

i. Struggled with this
ii. Solution: Data = data.split(“,”), self.i = 0. Def dfs():
    1. If data[self.i] == ‘N’: self.i += 1, return None
    2. Curr = TreeNode(int(data[self.i]))
    3. Self.i += 1
    4. Curr.left = dfs()
    5. Curr.right = dfs()
    6. Return curr
iii. Return dfs()

Heap/Priority Queue

1. Find Median from Data Stream
    a. Got 20/21, had time limit issue so had to watch video
    b. The idea is to use two heaps, one maxHeap called small, and one minHeap called large.
    c. Init: self.small = [], self.large = []
    d. Add: heappush to small. If the max value in small is > the minimum value in large, remove the max from small and add it to large. Also rebalance the size of the two heaps such that they are approximately the same length
        i. heapq.heappush(self.small, -1 * num)
        ii. If self.small and self.large and (self.small[0] * -1) > self.large[0]: val = heapq.heappop(self.small) * -1. heapq.heappush(self.large, val)
        iii. If len(self.small) > len(self.large) + 1: val = heapq.heappop(self.small) * -1. heapq.heappush(self.large, val)
        iv. If len(self.large) > len(self.small) + 1: val = heapq.heappop(self.large) * -1. heapq.heappush(self.small, val)
    e. findMedian:
        i. If len(self.small) > len(self.large): return -1 * self.small[0]
        ii. Elif len(self.large) > len(self.small): return self.large[0]
        iii. Else: return (-1 * self.small[0] + self.large[0]) / 2.0

Backtracking

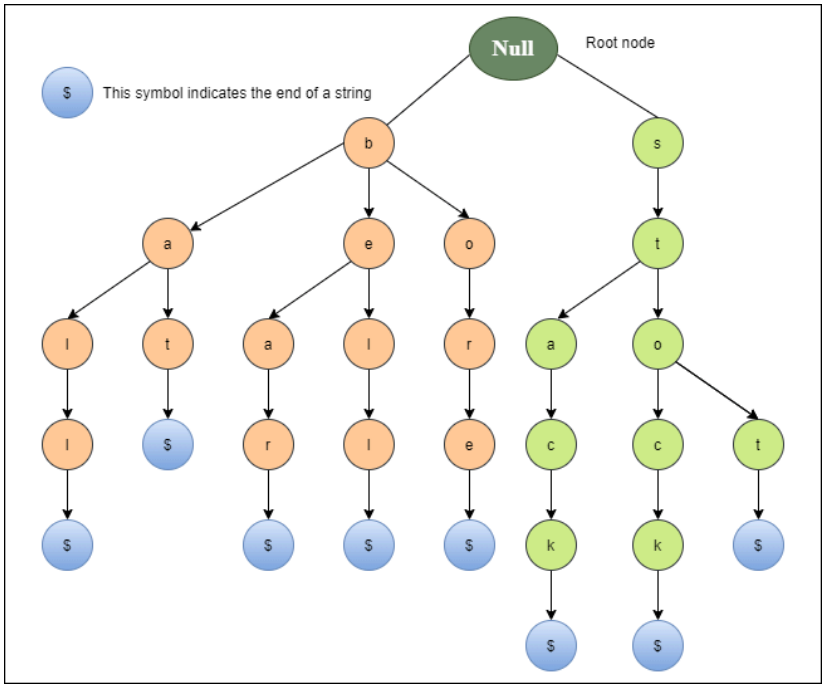
1. Combination Sum
    a. Note: Recall that lists in Python are mutable. This means when you are doing recursive functions, appending shallow copies of a list to a result does not work, as this is just a pointer meaning it will continue to be updated. Instead append a deep copy by using [:]
    b. Solution: Use a recursive decision tree where you either decide to append the curr value and continue, or completely skip over that number and no longer look at use cases with that number
        i. res = []
        ii. def dfs(i, curr, total):
            1. if total == target: res.append(curr[:]), return

2. if i>= len(combinations) or total > target: return
3. curr.append(combinations[i])
4. dfs(i, curr, total + combinations[i])
5. curr.pop()
6. dfs(i + 1, curr, total)

iii. dfs(0, [], 0)
iv. return res

2. Word Search
   a. Solution: Recursively go through each position on the board, check its neighbors, and return True once you complete the word
   b. rows = len(boards)
   c. cols = len(boards[0])
   d. path = set()
   e. def dfs(row, col, i):
      i. if i == len(board): return True
      ii. if i out of bounds: return False
      iii. if board[row][col] != word[i] or (row,col) in path: return False
      iv. path.add((row, col))
      v. res = dfs or for all different possible routes, incrementing i by 1
      vi. path.remove((row, col))
      vii. return res
   f. for r in rows:
      i. for c in cols:
         1. if dfs(r, c, 0): return True
   g. return False

Tries



1. Implement Trie (Prefix Tree)
    a. TrieNode class: Start by creating a separate class TrieNode, and set self.children = {}, and self.isEnd = False
    b. init: self.root = TrieNode()
    c. insert:
        i. initialize curr = self.root, for char in word:
            1. if char not in curr.children: curr.children[char] = TreeNode()
            2. curr = curr.children[char]
        ii. curr.isEnd = True
    d. search:
        i. curr = self.root, for char in word:
            1. if char not in curr.children: return False
            2. curr = curr.children[char]
        ii. return curr.isEnd
    e. startsWith:
        i. curr = self.root, for char in prefix:
            1. if char not in curr.children: return False
            2. curr = curr.children[char]
        ii. Return True
2. Design Add and Search Words Data Structure
    a. TrieNode, same as prev
    b. init: same as prev
    c. addWord: same as prev
    d. search (struggled with backtracking): def dfs(i, curr):

i. for j in range(i, len(word)):
    1. char = word[j]
    2. if char == '.':
        a. for child in curr.children:
            i. if (dfs j + 1, curr.children[child): return True
        b. Return False
    3. else:
        a. if char not in curr.children: return False
        b. curr = curr.children[char]
ii. return curr.isEnd

e. return dfs(0, self.root)

3. Word Search II
    a. Create a TrieNode class, same as prev but also def addWord inside this class
    b. root = TrieNode()
    c. for word in words: root.addWord(word)
    d. Get rows, cols, and also create a set for result and visited
    e. def dfs(row, col, trieNode, currWord):
        i. if row or col out of bounds, or (row, col) in visited, or board[row][col] not in trieNode.children: return
        ii. visited.add((row, col))
        iii. trieNode = trieNode.children[board[row][col]]
        iv. currWord += board[row][col]
        v. if trieNode.isEnd: result.add(currWord)
        vi. dfs(row *, col *, trieNode, currWord) for all 4 possible directions
        vii. visited.remove((row, col))
    f. for r in range(rows):
        i. for c in range(cols):
            1. dfs(r, c, root, "")
    g. return list(result)

Graphs

1. Number of Islands
    a. Hint: Go through each position, and bfs to add neighbour nodes to visited to keep track of current "island"
    b. Solution:
        i. res = 0, visited = set(), rows = len(grid) cols = len(grid[0])
        ii. def bfs(r, c):
            1. queue = [], queue.append((r, c)), visited.add((r, c))
            2. while queue:
                a. row, col = queue.pop(0)

b. *for each possible direction* if in bounds, equal to "1" and not in visited: queue.append((*new r, *new c)), visited.add((*new r, *new c)
c. **note: each possible direction is [1, 0], [-1, 0], [0, 1], [0, -1]

iii. for r in range(rows):
1. for c in range(cols):
a. if grid[r][c] == "1" and (r, c) not in visited:
i. res += 1, bfs(r, c)

iv. return res

2. Clone Graph
a. Hint: Use a hashmap to map old values to new ones, and perform a dfs. At each node, check if a copy is in the hashmap already, if it is return it, if not create a new Node, add it to the hashmap, go through all of the nodes neighbours and append them to the copy's neighbours by recursively calling dfs.
b. Solution:
i. oldToNew = {}
ii. def dfs(node):
1. if node in oldToNew:
a. return oldToNew[node]
2. copy = Node(node.val)
3. oldToNew[node] = copy
4. for n in node.neighbors:
a. copy.neighbors.append(dfs(n))
5. return copy

iii. return dfs(node) if node else None

3. Pacific Atlantic Water Flow
a. Hint: Find all islands that can reach the atlantic, and all that can reach the pacific. Return a set of all of the points on the graph that can reach both. The top row + left column can guaranteed reach the pacific, and the bottom row + right column can guaranteed reach the atlantic. Run a dfs from each position in these parts of the grid checking if the height is reachable (i.e. it is increasing since we are going backwards), and append it to a specific visited set.
b. Solution:
i. rows, cols = len(heights), len(heights[0])
ii. pacific, atlantic = set(), set()
iii. res = []
iv. def dfs(r, c, visited, prevHeight):
1. if r < 0 or r >= rows or c < 0 or c >= cols or (r, c) in visited or grid[r][c] < prevHeight: return
2. visited.add((r, c))
3. dfs(r+1, c, visited, heights[r][c])
4. dfs(r-1, c, visited, heights[r][c])
5. dfs(r, c+1, visited, heights[r][c])
6. dfs(r, c-1, visited, heights[r][c])

v. for r in range(rows):
    1. dfs(r, 0, pacific, heights[r][0])
    2. dfs(r, cols - 1, atlantic, heights[r][cols - 1]
vi. for c in range(cols):
    1. dfs(0, c, pacific, heights[0][c])
    2. dfs(rows - 1, c, atlantic, heights[rows - 1][c])
vii. for i in range(rows):
    1. for j in range(cols):
        a. if (i, j) in atlantic and (i, j) in pacific:
            i. res.append((i, j))
viii. return res

4. Course Schedule
    a. Hint: Created an adjacency list for each node in the graph and their prerequisites. Then create a visited set and perform a dfs on each courses prerequisites.
    b. Solution:
        i. graph = { i:[] for i in range(numCourses) }
        ii. for u, v in prerequisites:
            1. graph[u].append(v)
        iii. visited = set()
        iv. def dfs(course):
            1. if course in visited: return False
            2. if graph[course] == []: return True
            3. visited.add(course)
            4. for prereq in graph[course]:
                a. if not dfs(prereq): return False
            5. visited.remove(course)
            6. graph[course] = []
            7. return True
        v. for course in range(numCourses):
            1. if not dfs(course): return False
        vi. return True
5. NOTE: Skipped last 2 as not included in free leetcode, same with advanced graphs question

1-D Dynamic Programming

1. Climbing Stairs
    a. Solution 1: dp[] * n, set dp[0] = 1, dp[1] = 2 and then loop through all remaining until n setting dp[i] = dp[i-1] + dp[i-2]
    b. Solution 2:
        i. if n <=2: return n
        ii. n1 = 1, n2 = 2
        iii. for i in range(3, n+1)
            1. temp = n1 + n2
            2. n1 = n2

3. n2 = temp

iv. return n2

2. House Robber
    a. Solution:
        i. Dp = [0] * len(nums)
        ii. if len(nums) == 1: return nums[0]
        iii. dp[0] = nums[0]
        iv. dp[1] = max(nums[0], nums[1])
        v. for i in range(2, len(nums)):
            1. dp[i] = max(dp[i-1], dp[i-2] + nums[i])
        vi. return dp[-1]
3. House Robber II
    a. Hint: Do the same dp solution but for both 0 to n-2 and 1 to n-1 then return the max of the two
    b. Solution:
        i. if len(nums) == 1:
            1. return nums[0]
        ii. def helper(houses):
            1. dp = [0] * len(houses)
            2. dp[0] = houses[0]
            3. if len(houses) == 1: return dp[-1]
            4. dp[1] = max(houses[0], houses[1])
            5. for i in range(2, len(houses)):
                a. dp[i] = max(dp[i-1], dp[i-2] + houses[i])
            6. return dp[-1]
        iii. return max(helper(nums[:-1]), helper(nums[1:]))
4. Longest Palindromic Substring
    a. Hint: Check if palindrome by comparing to existing check i.e. if aaa is a palindrome then xaaax is a palindrome. Go through every index and check if expanding around the center is still a palindrome. If it is and that new string is larger than the current string than updated result
    b. Solution:
        i. res = “”
        ii. resLen = 0
        iii. for i in range(len(s)):
            1. l, r = i, i
            2. while l >= 0 and r < len(s) and s[l] == s[r]:
                a. if (r - l + 1) > resLen:
                    i. resLen = (r - l + 1)
                    ii. res = s[l:r+1]
                b. r += 1
                c. l -= 1
            3. l, r = i, i + 1
            4. while l >= 0 and r < len(s) and s[l] == s[r]:

a. if (r - l + 1) > resLen:
    i. resLen = (r - l + 1)
    ii. res = s[l:r+1]
b. r += 1
c. l -= 1

iv. return res

5. Palindromic Substrings
    a. Exact same as above but don't check for resLen, and keep track of total count
6. Decode Ways
    a. Hint: Look over the valid possibilities. If the first digit is 0, nothing else is possible so return 0. For dp[1] if s[0] is 1 or s[0] is 2 and s[1] in "0123456" then dp[1] is 2 else 1.
    b. Solution:
        i. if s[0] == '0': return 0
        ii. if len(s) == 1: return 1
        iii. dp = [0] * len(s)
        iv. dp[0] = 1
        v. if s[1] == '0':
            1. if s[0] not in "12": return 0
            2. else: dp[1] = 1
        vi. if s[1] == '1' or (s[1] == '2' and s[0] in "0123456"): dp[1] = 2
        vii. else: dp[1] = 1
        viii. for i in range(2, len(s)):
            1. if s[i] == '0':
                a. if s[i-1] == '1' or s[i-1] == '2': dp[i] = dp[i-2]
                b. else: return 0
            2. elif s[i-1] == '1': dp[i] = dp[i-1] + dp[i-2]
            3. elif s[i-1] == '2' and s[i] in "0123456": dp[i] = dp[i-1] + dp[i-2]
            4. else: dp[i] = dp[i-1]
        ix. return dp[-1]
7. Coin Change
    a. Hint: Initial dp with a maximum value, and of length amount + 1. Go through bottom up each amount, and for each coin check if adding the coin to dp[i-coin] is a more optimal solution.
    b. Solution:
        i. dp = [float('inf')] * (amount + 1)
        ii. dp[0] = 0
        iii. for i in range(1, amount + 1):
            1. for coin in coins:
                a. dp[i] = min(dp[i-coin] + 1, dp[i])
        iv. return dp[amount] if dp[amount] != float('inf') else -1
8. Maximum Product Subarray
    a. Hint: Keep track of min and max values, no need to use a dp array just store the values
    b. Solution:
        i. res = max(nums)

ii. currMin, currMax = 1, 1
iii. for num in nums:
    1. if num == 0:
        a. currMin, currMax = 1, 1
        b. continue
    2. temp = currMax
    3. currMax = max(num * currMax, num * currMin, num)
    4. currMin = min(num * temp, num * currMin, num)
    5. res = max(res, currMax)
iv. return res

9. Word Break
    a. Hint: Go backwards through the word, using a dp to store True or False values depending on if you can break the word at that index.
    b. Hint 2: dp[i] = dp[i + len(sub-word)]
    c. Solution:
        i. dp = [False] * (len(s) + 1)
        ii. dp[-1] = True
        iii. for i in range(len(s) - 1, -1, -1):
            1. for word in wordDict:
                a. if i + len(word) <= len(s) and s[i:i+len(word)] == word:
                    i. dp[i] = dp[i + len(word)]
                b. if dp[i]:
                    i. break
        iv. return dp[0]
10. Longest Increasing Subsequence
    a. Hint: Go backwards, and go through nums[i:] onwards. Update the dp with max(dp[i], dp[j] + 1). Return max(dp) at the end.
    b. Solution:
        i. dp = [1] * len(nums)
        ii. for i in range(len(nums)-1, -1, -1):
            1. for j in range(i+1, len(nums)):
                a. if nums[j] > nums[i]:
                    i. dp[i] = max(dp[i], dp[j] + 1))
        iii. return max(dp)

2-D Dynamic Programming

1. Unique Paths
    a. Hint: Initialise a dp array to [1] * n because it is guaranteed that the top row has 1 path for each cell. Then go through each row after that, initialising a newRow dp. Each element in that newRow = left cell + cell above values. Set dp = newRow after each loop.
    b. Hint 2: Never use the value for the row number to access the dp
    c. Solution:
        i. dp = [1] * n

ii. for i in range(1, m):
    1. newRow = [1] * n
    2. for j in range(1, n):
        a. newRow[j] = newRow[j-1] + dp[j]
    3. dp = newRow
iii. return dp[-1]

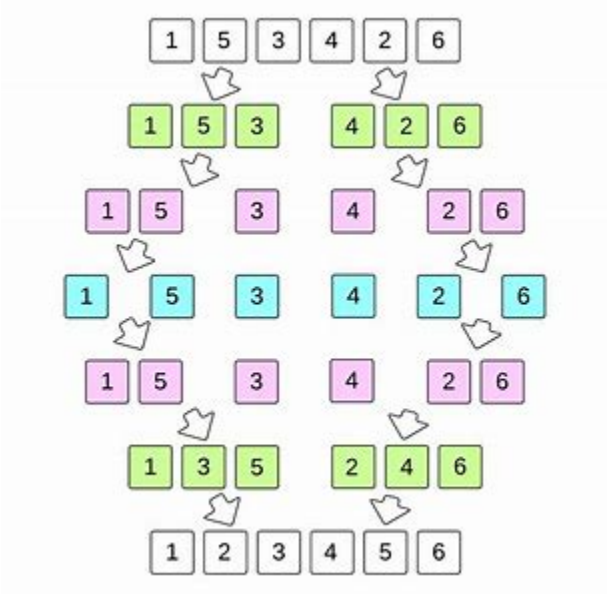
2. Longest Common Subsequence
    a. REMEMBER: 2d dp initialization is dp = [[0 for j in range(len(text2))] for i in range(len(text1))]
    b. Hint: Solving bottom up. Comparing the two strings in a 2d matrix, the position matrix[i][j] is 1 + the diagonal downwards to the right if the characters are equal, else it's the max to the right or above it.
    c. Solution:
        i. dp = [[0 for j in range(len(text2)+1))] for i in range(len(text1) + 1)]
        ii. for i in range(len(text1), -1, -1):
            1. for j in range(len(text2), -1, -1):
                a. if text1[i] == text2[j]:
                    i. dp[i][j] = 1 + dp[i+1][j+1]
                b. else:
                    i. dp[i][j] = max(dp[i+1][j], dp[i][j+1])
        iii. return dp[0][0]

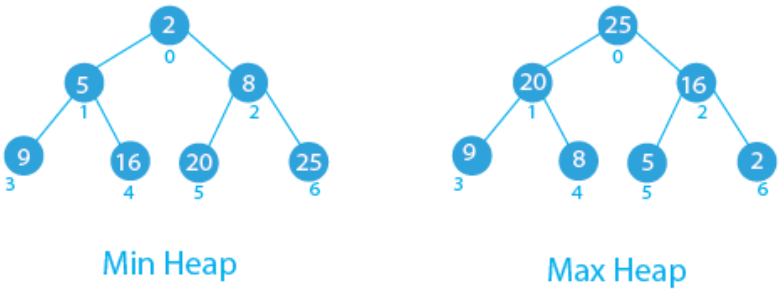
Greedy

1. Maximum Subarray
    a. Suuuuper easy after I got the hint
    b. Hint: Use a “sliding window” approach, if the current sum of the window is negative, disregard it entirely as we know it won’t be a part of the longest sum.
    c. Solution:
        i. res = float(‘-inf”)
        ii. curr = 0
        iii. for num in nums:
            1. curr += num
            2. res = max(res, curr)
            3. if curr < 0:
                a. curr = 0
        iv. return res
2. Jump Game
    a. Hint: DP approach is TLE, but if you are curious: initialize dp array filled with False, but dp[0] = True.
        i. For i in range(1, len(nums)):
            1. For j in range(i):
                a. if dp[j] and nums[j] + j >= i:
                    i. dp[i] = True, break

b. Hint: Set a “goal post” at the last position in the array. Go backwards through the array, and whenever you are able to reach this goal post, update the position of the goal post to the current index you are at. This is because we know if you can reach the goal post from an index i, we now just want to make sure we can reach index i, etc etc.
c. Solution:
    i. goal = len(nums) - 1
    ii. for i in range(len(nums) - 2, -1, -1):
        1. if i + nums[i] >= goal:
            a. goal = 1
    iii. return goal == 0

Notes

- collections.defaultdict(set) used to create a dictionary where each key's value is a set by default
    - Used in valid sudoku to keep track of rows, columns and grids
        - Recall grids is row // 3, column // 3
- BFS = start at source node and go layer by layer through the tree
    - Implement using queue
    - Used in Binary Tree Level Order Traversal
        - Queue to store values. While queue: get length of queue to ensure you are only looping through current level -> for i in range(length): get curr node by popping from queue, then append left and right to queue if node is not None.
- DFS = start at source node, go as far down as you can, and then backtrack until you find an unexplored branch
    - Stack
- Infinite is float("inf")
- Concatenate list:
    - Can use + operator i.e. nums + nums
- Merge Sort
    - Logic: Continuously split the array into halves until you get to sizes of 1. Then start going back up the splits and merge the arrays using a two pointer technique.
        - 
    - O(nlogn)
    - Solution: def two functions, merge(arr, l, m, r) and mergeSort(arr, l, r). Return mergeSort(nums, 0, len(nums) - 1)
        - merge(arr, l, m, r): create a list left = arr[l:m+1] and right = arr[m+1:r+1], as well as the index i = l to go through the arr, j = 0 as the left pointer, and k = 0 as the right pointer. Perform the two pointer analysis by doing
            - while j < len(left) and k < len(right):
                - if left[j] < right[k]: arr[i] = left[j], j += 1

            - Else: arr[i] = right[k], k += 1
        - i += 1
        - While j < len(left):
            - Arr[i] = left[j], j+=1, i += 1
        - While k < len(right):
            - Arr[i] = right[k], k+=1, i+=1
    - mergeSort(arr, l, r):
        - If l == r: return arr
        - m = (l + r) // 2
        - Recursively call mergeSort(arr, l, m) and mergeSort(arr, m+1, r)
        - Then merge by calling merge(arr, l, m, r)
        - Return arr
- Compare strings to sort array of strings:
    - ```
      def compare(a, b):
        if a + b > b + a:
          return -1
        else:
          return 1
      arr = sorted(arr, key=cmp_to_key(compare))
      ```
    - Insert into front of list:
        - list.insert(index, val)
    - Heap:
        - Heaps are binary trees for which every parent node has a value less than or equal to any of its children.
        - Add/pop in logn, get min in O(1)
        - ```
          import heapq
          listForTree = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
          heapq.heapify(listForTree)              # for a min heap
          ```
        - If you then want to pop elements, use:
        - ```
          heapq.heappop(minheap)      # pop from minheap
          ```
        - 

        - Max heap: negate the values
        - To get min or max, just get first index of heap (like a list)
          ```

- Copy list: [:]
- ^ is XOR in python
- For all permutations/combinations, you can import itertools and use itertools.permutations(list)

Big Tech notes:

- Common prefix: get smallest word of a list of words by doing min(array, key = len)
- Valid Parentheses: Can optimize by using dictionary
- First unique character: two loops, one for dictionary, one to go through char in s and see if dic[char] == 1
- Flood Fill: define dfs to go through all elements in the grid - make sure to add the (r, c) to a visited set to avoid max recursion error
- Climbing stairs cost: dp[0] * len(cost), dp[0] is cost[0], dp[1] is cost[1]. Then dp[i] = min(dp[i-1] + cost[i], dp[i-2] + cost[i]). ***IMPORTANT*** After loop return min of the last two dp elements
- Fibonnaci:
    - DP approach: dp = [0] * (n + 1), dp[0] = 0, dp[1] = 1, for 2 in range n + 1, dp[i] = dp[i-1] + dp[i-2]
        - Note: this can be achieved without dp but instead a result variable and two variables to keep track of “dp[i-1]” and “dp[i-2”
    - Recursive: Time complexity O(2^n), just have two base cases, if n == 1 or n == 0, return n. Else return self.fib(n-1) + self.fib(n-2)
- Sqrt(x): use binary search. If x == 0 or x == 1 return x, then l, r = 0, x. If mid * mid == x return mid, elif mid * mid < x l = mid + 1 res = max(res, mid), else r = mid
- Climb stairs: dp[i] = dp[i-1] + dp[i-2]
- Symmetric Tree: write a recursive function that takes a left node, and right node. Check if not left and not right then return True, but if one of them is null and the other isn’t return False. Then check if the values are equal, if they are not then return False. Then recursively check function(left.left, right.right) and function(left.right, right.left).
    - For iterative approach, use a queue and do a bfs, with logic being very similar to the recursive BUT if both are None continue instead of returning True
- IsAnagram: Can sort both strings and return s == t, but this is inefficient at O(nlogn). Instead create a dic for both strings, populate them with frequency of characters, and return dicS == dicT
- Missing Number **: Inefficient solution which is O(n^2) is looping through for i in range(len(nums)) and if i not in nums return i. Can also sort and then go through but that is O(nlogn). OPTIMAL solution is to get the sum of 0-n and the sum of nums, return sum of nums - sum 0-n.

- ** Move Zeroes: Have a pointer to keep track of the index to swap, and then a pointer to go through all elements in the array: l, r = 0, 0. While r and l are < len(nums), if nums[r] != 0: swap nums[r] and nums[l] and increment l. Always increment r.
    - Sorting approach: nums.sort(key=lambda x: x == 0)
- FizzBuzz: Modulus operator is costly, so you can also implement the answer by initializing a result with str(i) in range (1, n+1). And then loop through (2, n, 3) and replace res[i] with “Fizz”, and do likewise with (4, n, 5) - “Buzz” and (14, n, 15) with “FizzBuzz” respectively.
- IsPalindrome: Can just convert x to a str and compare if reversed is the same. If done mathematically, if x < 0 return False. Create a temp = x, and a res = 0. While temp > 0: num = temp % 10, res = res * 10 + num, temp = temp // 10. Return res == x.
- Check if Array is Sorted and Rotted: Check is done by going through each num in nums and making sure there is only ever one case where nums[i] < nums[i-1]. NOTE: Check for wrapping around as well.
- ** Buy And Sell Stock: Buy low sell high. Initialize res = 0 and curr = prices[0]. For i in range(1, len(prices): if prices[i] < curr: curr = prices[i], else: res = max(res, prices[i] - curr). Return res
    - Dp approach: dp = [0] * len(prices), min_price = prices[0]. For i in range(1, len(prices)): dp[i] = max(dp[i-1], prices[i] - min_price), min_price = min(min_price, prices[i]). Return dp[-1].
- **Merge Sorted Array: Create a pointer end, and a pointer one, two = m - 1, n - 1. Go backwards and fill accordingly.

Stack using queues: to add something to the stack, append it to the queue. And then pop all elements to len(i-1) and append to the back so the added element is front of queue.
Queue using stacks: use two stacks one to enqueue and one to dequeue.

In Order:

- Two Sum:
    - Brute force O(n^2), nested for loops
    - Optimal O(n) create a hashmap visited to store the num and their index. Loop through the nums, see if the remainder from total - nums[i] is in the visited hash. If it is return res[i, visited[remainder]]. Else visited[nums[i]] = i
- Two Sum Less Than K: sort nums, two pointer updating res with max of curr and res
- Trapping Rain Water
    - O(n), Memory O(n). Create an array leftMax where leftMax[i] = max(leftMax[i-1], height[i-1]). Do same thing with rightMax[i] = max(rightMax[i+1], height[i+1]. Then go through i in range(len(heights) and res += min(leftMax[i], rightMax[i]) - height[i] if that value is > 0.
    - Two pointer Memory O(1): l, r = 0, len(height)-1. leftMax = height[l], rightMax = height[r]. While l < r: if leftMax < rightMax: l += 1, leftMax = max(leftMax, height[l]), res += leftMax - height[l]. Else same thing but with r and decrementing
- Best Time to Buy and Sell Stock:
    - res = 0, min_price = res[0]. For i in range(1, len(prices)): if prices[i] < min_price: min_price = prices[i]. Else: res = max(res, prices[i] - min_price)
    - dp = [0] * len(prices), min_price = prices[0]. For i in range(1, len(prices)): dp[i] = max(dp[i-1], prices[i] - min_price), min_price = min(min_price, prices[i])
- Longest Substring Without Repeating Characters
    - O(n^2): define a valid function to make sure each char in a substring is unique, and then sliding window across substrings, if valid update res = max(res, r-l).
    - O(n) and mem O(m) where m # unique chars: dic to store chars and last seen in index, l, r = 0, 0, res = 0, while r<len(s): if s[r] in dic: l = max(l, dic[s[r]] + 1). Else: res = max(res, r-l+1), dic[s[r]] = r, r += 1.
- Merge Sorted Array: Pointer at end, one = m - 1, two = n - 1. While both >= 0, if nums1[one] >= nums2[two]: nums1[end] = nums1[one], one -= 1. Else same with 2. End -= 1. After loop, do while two >= 0 and do same thing.
- Number of Islands:
    - Rows, cols, visited = set(). Def dfs: if r or c out of bounds, or (r, c) in visited, or grid[r][c] == ‘0’: return. visited.add((r, c)), dfs all directions. After dfs, for r in rows, for c in cols: if grid[r][c] == ‘1’ and (r, c) not in visited: dfs(r, c), islands+=1
- Add Two Numbers:
    - Create a res list, and head = res. Also a carry = 0. While l1 and l2: curr = l1.val + l2.val+carry, carry=1 if curr >= 10 else 0. Head.next = ListNode(curr%10)
- Valid Parentheses:
    - Use dic for brackets, and stack. For char in s, if char in brackets: [if len(stack) 0 return False. If stack.pop() != brackets[char] return False.] Else stack.append(char). Return len(stack) == 0

- LRU Cach

```python
class Solution:
    def minArraySum(self, nums: List[int], k: int, op1: int, op2: int) -> i
        n = len(nums)
        @cache
        def dp(i, op1, op2):
            if i >= n:
                return 0
            s = dp(i+1, op1, op2) + nums[i]
            if op1 > 0:
                s = min(s, dp(i+1, op1-1, op2) + (nums[i] + 1) // 2)
            if op2 > 0 and nums[i] >= k:
                s = min(s, dp(i+1, op1, op2-1) + nums[i] - k)
            if op1 > 0 and op2 > 0 and (nums[i] + 1) // 2 >= k:
                s = min(s, dp(i+1, op1-1, op2-1) + (nums[i] + 1) // 2 - k)
            if op1 > 0 and op2 > 0 and nums[i] >= k:
                s = min(s, dp(i+1, op1-1, op2-1) + (nums[i] - k + 1) // 2)
            return s
        ans = dp(0, op1, op2)
        dp.cache_clear()
        return ans
```

```python
import heapq
class Solution(object):
    def leastInterval(self, tasks, n):
        counts = {}
        for task in tasks:
            curr = counts.get(task, 0) + 1
            counts[task] = curr
        maxHeap = [-cnt for cnt in counts.values()]
        heapq.heapify(maxHeap)

        queue = []
        time = 0
        while queue or maxHeap:
            time += 1
            if maxHeap:
                cnt = heapq.heappop(maxHeap)
                cnt += 1
                if cnt != 0:
                    queue.append([cnt, time + n])
            if queue:
                curr = queue[0][1]
                if curr == time:
                    curr = queue.pop(0)[0]
                    heapq.heappush(maxHeap, curr)
        return time
```

```python
s Solution(object):
def getLargestOutlier(self, nums):
    n = len(nums)
    total_sum = sum(nums)
    freq = {}
    max_val = float('-inf')
    for num in nums:
        freq[num] = freq.get(num, 0) + 1
    for num in nums:
        sum_left = total_sum - num
        if sum_left % 2 != 0:
            continue
        target = sum_left // 2
        if target in freq:
            if target != num:
                max_val = max(max_val, num)
            elif target == num and freq[num] >
                max_val = max(max_val, num)
    return max_val
```

```python
class Solution(object):
    def reorganizeString(self, s):
        if not s:
            return ""
        d = {}
        for c in s:
            if c in d:
                d[c] += 1
            else:
                d[c] = 1
        # push (-ve frq, char) pairs into heap
        h = []
        from heapq import heappush, heappop
        for k in d:
            heappush(h, (-d[k], k))

        res = ""
        # pop and examine frq and append to res
        while len(h) > 1:
            f1, c1 = heappop(h)
            f2, c2 = heappop(h)
            res += c1
            res += c2
            if abs(f1) > 1: # if char repeats
                heappush(h, (f1+1, c1)) # push back with decrement frq
            if abs(f2) > 1:
                heappush(h, (f2+1, c2)) # push back with decrement frq
        if h:
            f, c = h[0]
            if abs(f) > 1:
                return "" # this means we have something like h = [(2, "a")] which mea
escape from repeating same char in text
            else:
                res += c
```

```python
def groupAnagrams(self, strs):

    anagram_dict = {}
    count_empty = 0

    for word in strs:
        if word == "":
            count_empty +=1
            continue
        sorted_word = "".join(sorted(word))
        if sorted_word in anagram_dict:
            anagram_dict[sorted_word].append(word)
        else:
            anagram_dict[sorted_word] = [word]
    if count_empty > 0:
        anagram_dict[""] = [""] * count_empty
    return anagram_dict.values()
```

```python
ass Solution(object):
    def canFinish(self, numCourses, prerequisites):
        graph = { i:[] for i in range(numCourses)}
        for u, v in prerequisites:
            graph[u].append(v)
        visited = set()
        def dfs(course):
            if course in visited:
                return False
            if graph[course] == []:
                return True
            visited.add(course)
            for prereq in graph[course]:
                if not dfs(prereq): return False
            visited.remove(course)
            graph[course] = []
            return True
        for course in range(numCourses):
            if not dfs(course): return False
        return True
```

```python
class LRUCache(object):
    def __init__(self, capacity):
        self.capacity = capacity  # Maximum capacity of the cache
        self.cache = OrderedDict()  # Use OrderedDict

def get(self, key):
    if key in self.cache:
        # Move the accessed key to the end (mark as recently used)
        self.cache.move_to_end(key)
        return self.cache[key]
    return -1

def put(self, key, value):
    if key in self.cache:
        # Move the key to the end to mark it as recently used
        self.cache.move_to_end(key)
    self.cache[key] = value
    if len(self.cache) > self.capacity:
        # Remove the least recently used item (first item in the OrderedDict)
        self.cache.popitem(last=False)
```

```python
class Solution(object):
    def minSwaps(self, nums):
        n = len(nums)
        count_ones = sum(nums)  # Total number of 1's in the array
        if count_ones == 0 or count_ones == n:
            return 0  # If no 1's or all are 1's, no swaps needed
        # Extend the array to simulate a circular array
        nums_extended = nums + nums
        # Sliding window to find the max number of 1's in any window of size count_ones
        max_ones_in_window = 0
        current_ones = 0
        for i in range(len(nums_extended)):
            if i < count_ones:
                current_ones += nums_extended[i]
            else:
                # Slide the window: add the next element and remove the previous
                current_ones += nums_extended[i] - nums_extended[i - count_ones]
            max_ones_in_window = max(max_ones_in_window, current_ones)
        # Minimum swaps needed
        return count_ones - max_ones_in_window

```