Leetcode Notes and Practice

Sebastian Pucher

June 22, 2025

Contents

1 Hashing

1.1 Valid Anagram

Question: Given two strings s and t, return true if the two strings are anagrams of each other, otherwise return false

- Create two separate dictionaries
- Loop through one of the input strings, add key letter or letter freq.
- If dict are the same, return true

1.2 Twosum

Question: Given an array of integers nums and an integer target, return the indices i and j such that nums[i] + nums[j] == target and i != j

- Use a hashmap to store the index of each number in the array as the *value*
- On each iteration, check first to see if the difference between the target val and the current num is already stored in the hashmap
- If it is, then return the value at that key (the index), as well as the current index i
- If it's not, then add the current number and index to the hashmap
- **Key Idea:** Always check the existence between the target and the current number as a key in the hashmap first!

1.3 Group Anagram

Question: Given an array of strings strs, group all anagrams together into sublists. You may return the output in any order.

An anagram is a string that contains the exact same characters as another string, but the order of the characters can be different.

- The main idea of this question is to use a hashmap, and use arrays as keys. In this case, the array would have 26 elements, set to 0 to start.
- Each of the 26 elements represents a letter a-z. The idea is that you count each letter when traversing through the string.
- Once you have counted all the letters, use tupple() to change the mutable array into an immutable key that can be used to store the word.
- When adding it to the hash, first check to see if the key already exists. If so, add it to the array of other strs that have that mapping (add it to the values array at that key)
- If it's not a key, add it as a new key, and add the string as an element of an array to the value at that position.
- Return the values of the hashmap.

2 Two Pointers

2.1 Valid Palindrome

Question: Given a string s, return true if it is a palindrome, otherwise return false.

A palindrome is a string that reads the same forward and backward. It is also case-insensitive and ignores all non-alphanumeric characters.

- First change the string to all lowercase with .lower() function, this ensures case sensitive args are taken care of
- Init left and right pointers to first and last characters
- Do a check using **.isalnum()**. If it is *not* alpha numeric, then increment or decrement the pointer and *continue* through the loop
- While left is less than or equal two right, compare the letters, if not the same, return false
- increment / decrement left and right at bottom of loop

2.2 Two Sum II (Two Integer Sum)

Question: Given an array of integers numbers that is sorted in non-decreasing order.

Return the indices (1-indexed) of two numbers, [index1, index2], such that they add up to a given target number target and index1 ¡ index2. Note that index1 and index2 cannot be equal, therefore you may not use the same element twice.

There will always be exactly one valid solution.

Your solution must use O(1) additional space.

- **Main idea:** Use two pointers, one at the beginning of the array, and one at the end
- If the sum number at the two pointers added together is greater than the target, then this means that the current sum is two big, and we must *decrement the right pointer*
- Following this same logic, if the current sum between the two pointers is less than the target, then we must *increment the left pointer*.
- Do this until sum == target, then return.

3 Sliding Window

3.1 Best Time to Buy and Sell Stocks

Question: You are given an integer array prices where prices[i] is the price of NeetCoin on the ith day.

You may choose a single day to buy one NeetCoin and choose a different day in the future to sell it.

Return the maximum profit you can achieve. You may choose to not make any transactions, in which case the profit would be 0.

- **Buy low... sell high**: To achieve this, init two pointers, left wanting to find the low value in the arr, and right wanting to find the high value. Init left to first index, and right to second index in array
- Begin looping through, so long as the right pointer is less than the length of the array
- If the right value is ever less than the left value, then you have found a new low to sell at. *So assign left equal to right, and increment right by one: continue the while loop*
- Upon each iteration check if the difference between right and left values is greater than global profit (which is init to 0). If so, update profit

3.2 Longest Substring Without Repeating Characters

Question: Given a string s, find the length of the longest substring without duplicate characters.

A substring is a contiguous sequence of characters within a string.

- **Main Idea:** This is a sliding window question. The idea is that you will have two pointers start at the beginning of the string, and increment the right pointer, so long as the next character hasn't already occurred in the substring
- More formally: get a local max-len, as well as a global max-len. Begin by incrementing the right pointer and adding the char to some tracked char array, so long if it isn't in the char array already.
- If it is, then we need to *increment the left pointer*, and remove whatever value the left pointer is at from our tracked chars arr or hashmap. When we remove a char, be sure to decrement our local max-len.
- Keep incrementing left pointer and removing from array until we've removed the character that the right pointer was pointing at. Once at this point, we can now continue to increment the right pointer as above.
- When incrementing right pointer, always update the local max, and check if it's greater than the global max
- Once the left pointer has reached the end of the string, stop and return the global max

4 Stack

4.1 Valid Parentheses

Question: You are given a string s consisting of the following characters: '(', ')', '{', '}', '[' and ']'.

The input string s is valid if and only if: – Every open bracket is closed by the same type of close bracket. – Open brackets are closed in the correct order. – Every close bracket has a corresponding open bracket of the same type.

Return true if s is a valid string, and false otherwise.

- **Main idea:** Use a Stack to push open brackets, and pop closed brackets
- Before iterating through each char of the string, create an array of valid open brackets, and an array of valid close brackets (same positioning in each)
- For each char, if the char is neither in each bracket array, return false.
- If the char is an open bracket, push to stack
- If the char is an close bracket, *first check if the stack is non empty* then check to see if the last value on the stack is the corresponding open bracket to the current close bracket
- Continue looping if it is, return false if it isn't
- After the loop completes, ensure that the stack is empty, if it is, return true

4.2 Min Stack

Question: Design a stack class that supports the push, pop, top, and getMin operations.

– MinStack() initializes the stack object. – void push(int val) pushes the element val onto the stack. – void pop() removes the element on the top of the stack. – int top() gets the top element of the stack. – int getMin() retrieves the minimum element in the stack.

Each function should run in O(1) time.

- **Main idea:** This question is very easy except for returning the min in O(1) time. The idea here is to use another internal stack to keep track of all mins, as they are added.
- We can also set up another class var called current min. This is the value that we actually return when getMin() function is called.
- Using internal stack to keep track of min goes as follows: if a new added value is less than the current min, then update the new current min to this value, as well as add this value to the internal min stack

- If we pop the value, and it is the current min, then we need to pop the value from both the regular stack, as well as the min stack, and set the new current min to the top (last) value in the min stack. If the min stack is empty, just set the current min to NONE (current min will be init as NONE).

5 Binary Search

5.1 Binary Search Recursive

Question: You are given an array of distinct integers nums, sorted in ascending order, and an integer target.

Implement a function to search for target within nums. If it exists, then return its index, otherwise, return -1.

Your solution must run in O(logn) time.

- **Main idea:** To solve this problem recursively, you must keep track of left and right indices upon each recursive call
- Initialize left to be index zero, and right to be last index (len(A) -1)
- *Base Case:* If left index equals right index, check if the value is in only position, if it is, return index, else return -1
- If left doesn't equal right, find the mid index, and compare target value to value at middle index
- If it equals the middle index, return
- If it's greater than the middle index, recurse, left = middle index + 1
- If it's less than the middle index, recurse, right = middle index - 1
- **If parameters don't use left and right, you'll have to add them initially as none as defaults, then set them before recursing**

5.2 Binary Search Iterative

Question: You are given an array of distinct integers nums, sorted in ascending order, and an integer target.

Implement a function to search for target within nums. If it exists, then return its index, otherwise, return -1.

Your solution must run in O(logn) time.

- **Main idea:** To solve this problem iteratively, you must keep track of left and right indices and use a while loop
- *While loop Condition:* Loop so long as right is greater than or equal to the left index

- *Base Case:* Within the while loop, calculate the mid index: **mid = left + (diff between right and left / 2)**
- If the target equals the mid index, return mid index
- If left target is greater, then left becomes mid index +1
- If it's less than the middle index right = middle index - 1
- return -1 if not found (outside while loop)

5.3 Search 2D Matrix

Question: You are given an m x n 2-D integer array matrix and an integer target.

Each row in matrix is sorted in non-decreasing order. The first integer of every row is greater than the last integer of the previous row. Return true if target exists within matrix or false otherwise.

Can you write a solution that runs in O(log(m * n)) time?

- **Main idea:** Need to preform binary search *twice*.
- First binary search is done on the first column, which approximates where the actual value is
- To preform this first search, set up *iterative* binary search as usual
- **Edge Case:** When checking if the value is greater than the first in the row, must also check if the value is less than or EQUAL TO the last value in that row. This means that the value is *in* that row.
- If this condition is true, then you have found the row, and you can break from the first binary search loop. Store this row as the second array you will search over
- Now preform Binary Search a second time on this row, and return true or false depending on if the value was found or not

5.4 Koko eats Bananas

Question: You are given an integer array piles where piles[i] is the number of bananas in the ith pile. You are also given an integer h, which represents the number of hours you have to eat all the bananas.

You may decide your bananas-per-hour eating rate of k. Each hour, you may choose a pile of bananas and eats k bananas from that pile. If the pile has less than k bananas, you may finish eating the pile but you can not eat from another pile in the same hour.

Return the minimum integer k such that you can eat all the bananas within h hours.

- **Main idea:** The hardest part of this question is knowing what to preform a binary search on, as we need to find the minimum integer k such that all bananas can be eaten within h hours. In this case, the max that k can be is the largest pile.

- *Simplified Solution:* Preform binary search from k = 1 to max pile amount in banana piles: In other words, preform binary search on [1, max(piles)].
- In this case, we start at the half way pile. For this k value, check to see if all the bananas can be completed with this rate.
- If it cannot, this k value is too small, and we need to retry with a new larger k. So preform binary search on the upper half.
- If all the values can be completed in this time, this is a valid k, but it isn't necessarily the *smallest* k value. So we need to preform binary search again anyway on the left side of the array.
- If the smaller value doesn't work, then return the current min. If it does, repeat until the current min is the smallest working one.

6 Linked Lists

6.1 Reverse Linked List

Question: Given the beginning of a singly linked list head, reverse the list, and return the new beginning of the list.

- **Main idea:** Set up a previous and current pointer node that will be used to reassign next pointer values
- First check to see if the head node exists, if so, init prev to None, and current to the head
- While loop condition is while current exists:
- Create a temp value that stores currents next node value
- Set current.next equal to prev
- Set prev equal to current
- Set current equal to temp
- Once this loop has finished, the current value will be None, and we need to return the new head, which is the last node that came before None
- So we return prev

6.2 Cycle Detection Linked List

Question: Given the beginning of a linked list head, return true if there is a cycle in the linked list. Otherwise, return false.

There is a cycle in a linked list if at least one node in the list can be visited again by following the next pointer.

- **Main idea:** Set up a two pointers, one fast pointer and one slow pointer

- The slow pointer will increment one each upon each iteration, the fast pointer will increment by two.
- The idea is that the gap between the fast and slow pointer will gradually grow smaller and smaller until they point to the same node. This can only happen if there is a loop in the linked list
- Set up both pointers to begin at the head node, if the head exists
- Looping condition: While first pointer exists: first = first.next
- Check if second.next and second.next.next exists, if so, second = second.next.next
- Check if first = second. *Note:* Must specifically check that first equals second and not their values. Checking if the node struct stored in memory is the same struct, not just the values!

6.3 Merge Two Sorted Lists

Question: You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list.

- **Main idea:** Create a new head node that you will add two each list to, depending on which list has the smaller node when comparing the two
- Set up a current node that will be assigned to the new head node
- While looping condition: If list1 and list2, continue looping and comparing
- At the beginning of each loop, create a new node, and assign current at list1 or list2 to the new node depending on which one is smaller (increment list1 = list1.next or list2 = list2.next, depending on which was smaller)
- Set the current.next to new node, and then increment current: current = current.next
- After while loop is finished, check if list1 or list2 still has any nodes, if either does, assign the remaining to the current.next
- Return new-head.next (because new head is just a dummy node, and doesn't have any value)

7 Trees

7.1 Invert Binary Tree

Question: You are given the root of a binary tree root. Invert the binary tree and return its root.

- **Main idea:** Preform a DFS on the tree, before exploring left and right children, swap the left and right
- In explore function, first check if the current node is None, this is the base case: If it is, simply return
- If not None, create a temp node, and swap the left and right children. Then explore left node, and explore right node
- Explore function can be written as a part of the Solution Class
- In main function, call explore on root, then return root

7.2 Max Depth of Binary Tree

Question: Given the root of a binary tree, return its depth.

The depth of a binary tree is defined as the number of nodes along the longest path from the root node down to the farthest leaf node.

- **Main idea:** Preform a DFS on the tree, at each sub-problem, we want to pass up the max of the left and right at the current node
- In explore function, first check if the current node is None, this is the base case: If it is, simply return
- If not None, recurse on the left and right parts of the tree and store the depths, then return upward the max(left, right)
- Explore function can be written as a part of the Solution Class
- In main function, return the call to explore

7.3 Max Diameter of Binary Tree

Question: The diameter of a binary tree is defined as the length of the longest path between any two nodes within the tree. The path does not necessarily have to pass through the root.

The length of a path between two nodes in a binary tree is the number of edges between the nodes.

Given the root of a binary tree root, return the diameter of the tree.

- **Main idea:** Create a attribute of the Solution class (called max diameter) then preform a DFS on the tree
- The max diameter at each level is the longest number of edges spanning the left and right branches *added together*.
- The base case in this sense is if the current is NONE, it it is, RETURN -1, as there are no edges between the NONE node and the parent Node.

- At each sub-problem, recurse on the left and right sides (adding 1 to the return), and check if right added to left is greater than the member class var of max diameter.
- If it is, set max diameter.
- We want to pass up the longest span at any given node, which is the max between the left and right span
- Call explore function from max diameter, and return the int value that explore returns.

7.4 Balanced Binary Tree

Question: The depth of a binary tree is defined as the number of nodes along the longest path from the root node down to the farthest leaf node.

- **Main idea:** This question is a bit trickier, as you need to return a tuple, and not just a depth (like edge depth or node depth)
- To begin preform a DFS on the tree. Start be defining the base case. If the node is NONE, return (0, True), as a None node is balanced. Because we are defining balance in terms of node height, and not edge height, base case starts at 0 and not -1.
- Next, recurse on the left and right sub trees, and add one to the depth part of the tuple and store the True or False vale.
- Check if the distance between left and right is equal to 1 or 0, if it is, set a current balanced value equal to True.
- If this current balanced is True, and so are the left and right return T/F values are true, then return True upwards to the next subproblem.
- In terms of return depth upwards, we want to return the max depth between the left and right sub nodes.
- So, return (max(left, right), (is-bal and left-bal and right-bal)) upwards
- In main function, return the True or false value from the tuple as the answer

7.5 Same Binary Tree

Question: Given the roots of two binary trees p and q, return true if the trees are equivalent, otherwise return false.

Two binary trees are considered equivalent if they share the exact same structure and the nodes have the same values.

- **Main idea:** Use DFS to search through each tree, checking nodes along the way. If DFS is preformed in the same order for both trees, then they are the same

- Two ways of solving this question: First is less buggy, but also less efficient. Idea is to create one explore call, and a visited array, and add nodes to the array upon being visited. Call explore twice, with two separate visited arrays, one for each tree
- Check the arrays, if they are the same, return true, else return False.
- *NOTE*: Doing it this way means you need to express the NULL Node in the array: In the base case, make sure to add None to the array, when you hit a None node.
- **BETTER WAY:** A better way is to modify the explore function directly and have it accept two nodes.
- Base case in this example becomes if *both* nodes are None, return True upwards. If one node exists, and the other doesn't. Return False.
- Recurse on the left and right subtrees, comparing nodes as you go.

7.6 Subtree of Another Tree

Question: Given the roots of two binary trees root and subRoot, return true if there is a subtree of root with the same structure and node values of subRoot and false otherwise.

A subtree of a binary tree tree is a tree that consists of a node in tree and all of this node's descendants. The tree tree could also be considered as a subtree of itself.

- **Main idea:** Use DFS to search through the main root tree. At each node, check if the current node val equals the target node val. If they do, then we can begin an is same tree check with this node as our new root.
- Checking if a tree is the same can be done in the two ways described in the previous question. We can use a DFS approach, and explore / check as we go, or create arrays that track the order in which the tree was traversed, and compare the arrays
- *SERIALIZE THE TWO TREES*: After banging my head against the wall, the best move it to serialize the lists.
- Once this exploration and checking has been done, if the result is true, we must set a global variable to True, if it's not, we can *keep exploring* the main rooted tree.
- If the main rooted tree gets completely traversed, return false
- Can create the root serization in each stack frame: examine code for best practice in using serialization

8 Heap and Priority Queue

8.1 Kth Largest Element in a Stream

Question: Design a class to find the kth largest integer in a stream of values, including duplicates. E.g. the 2nd largest from [1, 2, 3, 3] is 3. The stream is not necessarily sorted.

Implement the following methods:

- constructor(int k, int[] nums) Initializes the object given an integer k and the stream of integers nums.
- int add(int val) Adds the integer val to the stream and returns the kth largest integer in the stream.

- **Main idea:** Use a heap! By default, python has a built in heap library called heapq that uses min heaps
- In the constructor set the value k to self.k
- Next, heapify the array, this is basically a free operation, as it takes liner time : O(n). *NOTE:* Heapify happens in place (pass by reference), this means that we do not assign the array to a new variable.
- Next set the heapifyed array to another attribute variable (say streams).
- In the addition function, first heappush() the value into the array.
- **heappush() time complexity:** Heappush adds the new values to the last value of the tree, the bubbles up. Bubbling up involves swapping nodes upward towards the root. This swapping can occur a max number of times thats equal to the height of the tree. So worst case is O(log(n)), time.
- **heappop() time complexity:** Heappop is simular to heappush, as it swaps nodes. Only the root is swapped with a leaf node, the leaf node is removed and returned, and sink down swapping function is called on the root. This again can happen in O(log(n)) time.
- In our add function, after we've added the node, we can now heap pop so long as the length of the stream is greater than our kth value. In other words, keep popping until you've popped k times: return the kth element. Time complexity is O(klog(n))

8.2 Last Stone Weight

Question: You are given an array of integers stones where stones[i] represents the weight of the ith stone.

We want to run a simulation on the stones as follows:

At each step we choose the two heaviest stones, with weight x and y and smash them together

- If x == y, both stones are destroyed
- If x ¡ y, the stone of weight x is destroyed, and the stone of weight y has new weight y - x.
- Continue the simulation until there is no more than one stone remaining.

Return the weight of the last remaining stone or return 0 if none remain.

- **Main idea:** Use a heap! This is a max heap problem, so the hardest part about the solution is ensuring we're translating the question correctly while using our min heap. To begin, we can multiply every value by -1 in the array, then call heapify.
- In the simulation, we choose the two heaviest stones, which means the two most negative stones. So heappop twice, and set x and y equal to the return values. Loop and grab the two largest values so long as the length of the heap is greater than 1
- Now check to see if both of the values are the same, if they are, continue to the next iteration (continue inside while loop)
- If x ¡ y, then we must be careful in translating this to logic in the question, this condition will only happen if x is actually greater than y due to the values being negative. If this is the case, then we need to add y back in with weight y-x. In this case, we will add y back in with weight x - y. To see this math work out, I suggest writing out a couple of values to see how the negative works.
- Once the looping expression evals to false, check to see if there are 1 or None values left in the heap
- If there are none, then return 0, else, return -1 * heap[0] - ¿ remember to turn the negative back to positive!

9 Graphs

9.1 Number of Islands

Question: Given a 2D grid grid where '1' represents land and '0' represents water, count and return the number of islands.

An island is formed by connecting adjacent lands horizontally or vertically and is surrounded by water. You may assume water is surrounding the grid (i.e., all the edges are water).

- **Main idea:** Preform a DFS on the graph using a visited nodes list. Explore all neighbor nodes of a starting node, so long as the neighbor has not been visited.
- We will wrap the DFS call in a for loop that loops through every node. Every time the explore call returns, we will += our island counter by the return call of explore (which will return 1 or 0). This is the value that we will eventually return.

- Inside our explore function, we must first check to see if the node had a value 1 or 0, if its 0, simply 0
- If the value is 1, then we can explore all of the nodes neighbors.
- We can fist built our another function that returns all the neighbors of the given node. We can then loop through this list, for every non-visited neighbor.
- After we have finished the recursive calling, we can return 1 to the parent wrapper function.
- Now having finished exploring the graph, we can return the island counter.
- **Note:** I used a list of tuples (x,y) to store visited nodes, but there's probably a better way to do this.

10 One-Dimensional Dynamic Programming

10.1 Climbing Stairs

Question: You are given an integer n representing the number of steps to reach the top of a staircase. You can climb with either 1 or 2 steps at a time.

Return the number of distinct ways to climb to the top of the staircase.

- **Main idea:** We can use **Memoization** to keep track of sub solutions to each part of the problem.
- **Subproblem:** At each step, the number of distinct ways that can be reached is the sum of the number of distinct ways that can be reached at steps n-1 and n-2. memo[n-1] + memo[n-2].
- We can set our current memo value, to the sum of the precious two memo values.
- To begin this problem, (or any DP problem), always start by populating the memo with the base cases of the problem.
- In this case, we populate our memo with memo = 1:1, 2:2
- At the beginning of our recursion, we will check to see if the current value step is in the memo, if it is, then we return that memo value.
- If not, we add a new entry to our memo that equals our sub problem: memo[n] = memo[n-1] + memo[n-2]. To do this, we first must check to make sure that memo[n-1] and memo[n-2] exist. If we're doing the top down approach, they won't, so we make a recursive call: memo[n] = self.memoClimb(n-1, memo) + self.memoClimb(n-2, memo).
- We then return the memo[n] upward
- **Note:** This problem can also be done iteratively, and honestly probably is easier to do that way, I put both my solutions in the python file.

10.2 Min Cost Climbing Stairs

Question: You are given an array of integers cost where cost[i] is the cost of taking a step from the ith floor of a staircase. After paying the cost, you can step to either the (i + 1)th floor or the (i + 2)th floor.

You may choose to start at the index 0 or the index 1 floor.

Return the minimum cost to reach the top of the staircase, i.e. just past the last index in cost.

- **Main idea:** We can use **Memoization** to keep track of sub solutions to each part of the problem.
- **Subproblem:** At each step, the minimal cost would be the cost to get to that current step, plus the min at steps n-1, n-2. : memo[step] = currentCost + min(self.memoStairs(step-1, cost, memo), self.memoStairs(step-2,cost, memo))
- To begin this problem, (or any DP problem), always start by populating the memo with the base cases of the problem.
- In this case, we populate our memo with the costs associated at steps 1 and 2
- At the beginning of our recursion, we will check to see if the current value step is in the memo, if it is, then we return that memo value.
- If not, we add a new entry to our memo that equals our sub problem.
- Because we need to get to the last stair, the stair that doesn't have a value, we need to create a current cost variable that tracks the current values current cost. If the last stair is met, then the current cost will just equal 0.
- After all of the recursion has been done, we will return memo[step]