1) https://leetcode.com/problems/two-sum/

```
#O(N) Time, O(N) Space
class Solution(object):
    def twoSum(self, nums, target):
        dictionary = {}
        for key, value in enumerate(nums):
            complement = target-value
            if complement in dictionary:
                return [dictionary[complement],key]
            else:
                dictionary[value] = key
```

We will want to use a HashMap(dictionary) for this solution. We will also want to use the enumerate function to iterate through the values in nums so that we can keep the value of the counter. The solution we are looking for is "x+y=target". If we want to solve for y, we will rearrange this to "y = target-x", y being the complement, target being the target number we're trying to sum up to, and value being the current value for our iteration. During our iteration if we find the complement in our dictionary, we are going to return the index of complement and our current index. Otherwise, we set the index of our value to our current key.

Traverse array, if complement is in dictionary, return both its index and our current index, if not then set the index of our current value to our current index.

2) https://leetcode.com/problems/reorder-data-in-log-files/

```
#O(NLogN) Time, O(N) Space
class Solution(object):
    def reorderLogFiles(self, logs):
        letter_list=[]
        digit_list=[]
        for log in logs:
            if log[-1].isdigit():
                digit_list.append(log)
            else:
                letter_list.append(log)
        letter_list = sorted(letter_list, key=lambda letter: (letter.split()[1:],letter.split()[0]))
        return letter_list+digit_list
```

We can have two arrays, one for letters and one for digits. We iterate through each log and check the last index to see if it is a digit or not. If it is, we will add it to the digit array and if it is not, we will add it to the letter array. We will then use a lambda sort function that first checks the suffix and then checks the identifier if there is a tie in suffixes. After the letter array is sorted, we will merge the letter array and digit array (letter array first).

Check if digit or letter log, sort letter log by suffixes then identifiers, merge lists.

3) https://leetcode.com/problems/second-highest-salary/

```
SELECT
    IFNULL(
      (SELECT DISTINCT Salary
       FROM Employee
       ORDER BY Salary DESC
        LIMIT 1 OFFSET 1), NULL)
        AS SecondHighestSalary
```

Use SELECT and test IFNULL, SELECT DISTINCT will return only different values from the Employee table, ORDER BY DESC so you can get the highest salaries first and use LIMIT 1 which takes an OFFSET argument of 1 (one from the top of the list) if there are no results from the test then return NULL, set as the alias SecondHighestSalary.

4) https://leetcode.com/problems/valid-parentheses/

```
#O(N) Time, O(N) Space
class Solution:
    def isValid(self, s: str) -> bool:
        stack = []
        dict = {"]":"[", "}":"{", ")":"("}
        for bracket in s:
            if bracket in dict.values():
                stack.append(bracket)
            elif bracket in dict.keys():
                if stack == []:
                    return False
                elif dict[bracket] != stack.pop():
                    return False
            else:
                return False
        return stack == []
```

We can use a stack and a HashMap for this solution. We iterate through the brackets string and check whether the bracket is an opening or closing bracket. If it is an opening bracket, we push it onto the stack. If it is a closing bracket, we ensure the stack is not empty and then pop the stack to see if this value matches the value of bracket in our dictionary. At the end of the function we should have an empty stack, so we return whether this is true or not.

Use a stack and a dictionary of bracket values, if opening bracket push onto stack, if closing bracket check for empty string and if the value in dictionary matches this popped stack value, return if stack is empty.

5) https://leetcode.com/problems/maximum-subarray/

```
#O(N) Time, O(1) Space
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        for i in range (1, len(nums)):
            if nums[i-1] > 0:
                nums[i] += nums[i-1]
        return max(nums)
```

We can use Kadane's algorithm for this solution. The question is asking for the maximum contiguous subarray, so we will check for positive subarrays. We iterate through the array starting from the second index and we check if the previous element is positive. If it is, we add it to our current element and continue. Once we are out of the for loop we return the max element, this would be the maximum sum out of all the contiguous subarrays.

Iterate through the array starting from the second index, if the previous element is positive then add it to our current element, return max element in array.

6) https://leetcode.com/problems/valid-palindrome-ii/

```
#O(N) Time, O(N) Space
class Solution:
    def validPalindrome(self, s: str) -> bool:
        left = 0
        right = len(s) -1
        while left < right:
            if s[left] == s[right]:
                left += 1
                right -= 1
            else:
                substring_one = s[left:right]
                substring_two = s[left+1:right+1]
                return substring_one==substring_one[::-1] or substring_two==substring_two[::-1]
        return True
```

You start with a typical while loop that checks if the left index matches the right and move inwards, if you have mismatched elements you try to test two substrings, one without the left index included and one without the right index included.

7) https://leetcode.com/problems/best-time-to-buy-and-sell-stock/

```
#O(N) Time, O(1) Space
class Solution:
    def maxProfit(self, prices):
        max_profit = 0
        min_price = float('inf')
        for price in prices:
            min_price = min(min_price, price)
            max_profit = max(max_profit, price - min_price)
        return max_profit
```

We initialize our max_profit to 0, as this would be the base case. We set our min_price to infinity so we can override it with any price within our input. We then iterate through our prices and set the minimum price to the lowest number between our current min_price and the current price in our iteration. We also set the max profit to the highest number between our current max_profit and the current price in our iteration minus our min_price. We do this so that we only keep the highest max_profit, and to check if we are still at the base case of no profit. Return the max_profit.

Iterate through prices to find minimum price and subtract it from the current price and see if this value is higher than the current maximum profit, if it is then save this value.

8) https://leetcode.com/problems/merge-two-sorted-lists/

```
#O(N) Time, O(N) Space
class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        dummy = ListNode(0)
        prev = dummy
        while l1 is not None and l2 is not None:
            if l1.val <= l2.val:
                prev.next = l1
                l1 = l1.next
            else:
                prev.next = l2
                l2 = l2.next
            prev = prev.next
        prev.next = l1 if l1 is not None else l2
        return dummy.next
```

We first set a dummy head as a placeholder. We do this because the current node is 0, and the head we will be returning will be a node from one of the two lists and that actual head will be set after this dummy one. Afterwards, we will set the previous node to the dummy so that we can take a step forward in the merged list. We will then iterate through both lists while neither one of them is null and see if the current node value in the first list is less than or equal to the current node value in the second list. If it is, we set the previous value's next value to the current node in the first list and step forward in list one. If it is not, we set the previous value's next value to the current node in the second list and step forward in list two. After the previous value's next value has been set from one the two conditions set previously, we set that value as the previous value to step forward in the merged list. Once we are out of the while loop, it means one of the lists are null. So, we glue the non-null list onto the end of our merged list since it is already sorted. Once we are done, we return all the values after our dummy head.

Make a dummy head and while neither input lists are null, set the lesser or equal node as the next value in our merged list, once we've reached the end of an input list we will attach the other list to the end of our merged list and return everything after the dummy head.

https://leetcode.com/problems/monotonic-array/

```
#O(N) Time, O(1) Space
class Solution:
    def isMonotonic(self, A: List[int]) -> bool:
        increasing=decreasing=True
        for i in range(1, len(A)):
            if A[i] > A[i-1]:
                decreasing = False
            elif A[i] < A[i-1]:
                increasing = False
        return increasing or decreasing
```

Iterate through the array and compare each index with the index before while checking two conditions, either it is increasing or decreasing return if one of the conditions is true.

https://leetcode.com/problems/monotonic-array/

```
#O(N) Time, O(1) Space
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        pointer_one = None
        pointer_two = head
        while pointer_two != None:
            pointer_three = pointer_two.next
            pointer_two.next = pointer_one
            pointer_one = pointer_two
            pointer_two = pointer_three
        return pointer_one
```

We can solve this question by iterating through the linked list and setting the next node as the previous node, thus reversing the list. We will do this by utilizing three pointers. We will want these pointers to keep track of the current node, the previous node, and the next node. We will initialize pointer one as None, as initially the head will have no previous node, and the second pointer as the head node. While the current node in our iteration (initially the head) is not none, we will set a third pointer to the next node of the current node. We want to do this because once we override pointer_two.next and set it to the previous node, we will lose our bridge to get to the next node. The third pointer is just a placeholder. Once we set the third pointer to the next node, we will override the reference to our next node by setting it to the previous node, pointer_one. We then move both our current node pointer (pointer two) and our previous node pointer (pointer one) forward. We will return pointer one at the end as it will be the new head.

Traverse the linked list and set the next node to the previous node by using three pointers to keep track of the previous, current, and next nodes.

10) https://leetcode.com/problems/add-strings/

```
#O(N) Time, O(N) Space
class Solution:
    def addStrings(self, num1: str, num2: str) -> str:
        result = ''
        carry, index_one, index_two = 0, len(num1) - 1, len(num2) - 1
        while index_one >= 0 or index_two >= 0:
            current_sum = carry
            if index_one >= 0:
                current_sum += int(num1[index_one])
                index_one -=1
            if index_two >= 0:
                current_sum += int(num2[index_two])
                index_two -=1
            result += str(current_sum%10)
            carry = current_sum // 10
        if carry > 0:
            result += str(carry)
        return result[::-1]
```

We can solve this question by starting from the ones position of each string and adding that number to our sum. Similarly, we will move to the tens, hundreds, and thousands position after. We will mod our current sum by 10 in each iteration so that we know if we need to carry over any values, we can keep track of this. We will find the carry value at the end of each iteration by dividing our current sum by 10 and initializing our current sum to the carry value at the beginning of the next iteration. If there are any carry values left over once we have broken out of our while loop, we will append the value to the end of our string, then return the reversed string since it will be backwards.

Traverse strings backwards adding the values in each position while still including carry-overs and return reversed string.

11) https://leetcode.com/problems/move-zeroes/

```
#O(N) Time, O(1) Space
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        pointer_one = 0
        for pointer_two in range(0, len(nums)):
            if nums[pointer_two] != 0:
                nums[pointer_one], nums[pointer_two] = nums[pointer_two], nums[pointer_one]
                pointer_one += 1
        return nums
```

We can solve this problem using two pointers. If the order of the non-zero values did not matter, we could have a left pointer and a right pointer then move them inwards swapping right zeros with left non-zeros. Since the order does matter however, we will use two pointers initialized at zero. If the value at the second pointer is a non-zero value, we will swap it with the value at the first pointer and move both pointers forward. If it is a zero, we will only move the second pointer forward until we find a non-zero again to swap with the zero at the first pointer.

Traverse the list using two pointers, one lagging on zeros and one finding non-zeros, if the second pointer finds a non-zero then swap the value with the first pointer and move the first pointer forward.

12) https://leetcode.com/problems/happy-number/

```
#O(log N) Time, O(log N) Space
class Solution:
    def isHappy(self, n: int) -> bool:
        seen = set()
        square_sum = 0
        while square_sum != 1:
            square_sum = 0
            while n > 0:
                square_sum += (n % 10) ** 2
                n = n // 10
            if square_sum in seen:
                return False
            else:
                n = square_sum
                seen.add(n)
        return True
```

We can solve this question by using a set to keep track of the square sums. We will initialize the square sum to be 0 as we have no sum to process initially. We will then make two while loops, the outer will run until we have a square sum equal to 1, the inner will run while our n (our initial input or the previous iteration's square sum) has digits left to process. Before the inner loop, we will set the square sum back to zero so that we are not adding to the previous iteration's square sum. The inner loop will process each digit of n, square it, add it to the square sum, and remove it from n by utilizing the mod and division operators. Once we have our square sum, we are going to see if it is in the set. If it is, we return false and break out of the loop. If it is not, we will add it to the set and set n equal to it so it can be processed in the next iteration. If we have reached the end of the outer loop it is because we have a square sum that equals 1, so we return true.

Use a set to keep track of previous square sums we have seen, process new square sums by squaring each digit and adding them all together.

https://leetcode.com/problems/middle-of-the-linked-list/

```
#O(N) Time, O(1) Space
class Solution:
    def middleNode(self, head: ListNode) -> ListNode:
        slow = fast = head
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
        return slow
```

Use a slow and a fast pointer starting at the head, increment the slow pointer by one and the fast pointer by two, the slow pointer will be halfway when the fast pointer reaches the end of the list so return it.

13) https://leetcode.com/problems/reverse-integer/

```
#O(N) Time, O(1) Space
class Solution:
    def reverse(self, x: int) -> int:
        reversed_int = 0
        is_negative = x < 0
        x = abs(x)
        while x != 0:
            reversed_int *= 10
            reversed_int += x%10
            x //= 10
        if reversed_int > 2**31:
            return 0
        if is_negative == True:
            return -reversed_int
        else:
            return reversed_int
```

We will first check if x is negative so we know whether the final int will need to be negative as well. We will then convert x to positive by taking the absolute value so we can process each digit individually. We will make a while loop for chopping off each digit from x and it will run until there are no more digits to chop. We will start by multiplying the output integer "reversed_int" by 10 so that we will have a placeholder for the next digit. Then we will add the last digit to our output by modding x by 10. After that, we will chop off the digit by dividing x by 10, ensuring it does not convert to a float by using "//" instead of "/". If our reversed integer is larger than a 32-bit signed integer (2^31), we return 0. Otherwise, we return our reversed integer, negative if the input was negative.

Check if input is negative and add each individual digit to output while using multiplication by ten as a digit placeholder, modulus by 10 as a digit parser and division by 10 as a digit remover, also check if 32-bit signed and return negative if negative.

14) https://leetcode.com/problems/merge-sorted-array/

```
#O(M) Time, O(1) Space
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        pointer_one, pointer_two = m - 1, n - 1
        pointer_three = len(nums1) - 1
        while pointer_one >= 0 and pointer_two >= 0:
            if nums1[pointer_one] < nums2[pointer_two]:
                nums1[pointer_three] = nums2[pointer_two]
                pointer_two -= 1
            else:
                nums1[pointer_three] =  nums1[pointer_one]
                pointer_one -= 1
            pointer_three -= 1
        nums1[:pointer_two + 1] = nums2[:pointer_two + 1]
```

We can solve this using three pointers and iterating through both arrays from the end to the beginning. We set the first pointer to be m – 1 (the size of non-zero elements in nums1, minus one to avoid index out of bounds errors), we set the second pointer to be n - 1 (also for the reason just mentioned), and the third pointer to be len(nums1) -1 (this will be at the end of nums1 after the non-zero elements and the zeros as well. Once we set our pointers, we will create a while loop that will run while both pointer one and pointer two are greater than zero so that we do not throw an index out of bounds error. It will check for the smaller of the two elements and set the index the third pointer is aimed at to the smaller number and decrement the smaller number pointer, and the third pointer. If both numbers are equal, no swapping or inserting will need to be done so we can continue. Once we have broken out of the while loop it means that one of the pointers has reached the beginning of it's array so we can prepend the remaining elements from nums2 into nums1 (the "+ 1" in "nums2[:pointer_two + 1]" is because the ending index of string slicing is exclusive.

Traverse both arrays from the end to the beginning using three pointers to keep track of the end of both non-zero arrays and also the end of the larger array that will house all the elements, check which element is smaller and set the third pointer's index to it, once the traversal is done then prepend any remaining elements from the smaller array into the larger one.

15) https://leetcode.com/problems/add-binary/

```
#O(N + M) Time, O(max(N,M)) Space from saving answer
class Solution:
    def addBinary(self, a, b) -> str:
        x, y = int(a, 2), int(b, 2)
        while y is not 0:
            answer = x ^ y
            carry = (x & y) << 1
            x, y = answer, carry
        return bin(x)[2:]
```

We can solve this question by using bit manipulation. Similarly, we could also have chosen to solve this question like the adding strings leetcode question. To solve this question using bit manipulation however, we will use XOR and AND logic gates. We'll first convert x and y to binary numbers (base 2). Then we'll use a while loop that will run while y is greater than 0. We'll do our initial addition of the two binary numbers where we use an XOR gate to sum each number in each (base 2) place where the numbers aren't both equal to 1. The reason behind this is if both values aren't equal to 1 then we won't need to carry any values. After we get that output, we'll then go in with an AND gate and get the values we'll need to carry where both (base 2) places are 1s. Since we're carrying it, we'll need to shift the value to the left so that we can add it to the the sum of the next iteration. We'll continue to do this until there are no more values to carry and return the binary sum. If we return bin(x) for example 4, it will return 0b100, so we'll want to return everything from the third index onwards (bin(x)[2:]).

Set x and y equal to binary a and b, iterate through the operations with a loop while there are still values to carry, the answer for each iteration is x XOR y, the carry value for the next iteration is x AND y shifted left by one, return the binary form of x and slice off the first two characters.

16) https://leetcode.com/problems/climbing-stairs/

```
#O(N) Time, O(1) Space
class Solution:
    def climbStairs(self, n: int) -> int:
        if n is 0 or n is 1:
            return 1
        one_behind, two_behind = 1, 2
        for i in range(3, n+1):
            one_behind, two_behind = two_behind, one_behind + two_behind
        return two_behind
```

We can solve this question by calulating a bottom-up approach using dynamic programming. We know that for our base cases of combinations if we want to climb 0 or 1 steps, the answer will be 1 as there will be only 1 way to climb one step, and you don't climb 0 steps. If we want to climb 2 steps, we have 2 combinations of doing that, we can either climb 2 at a time or we can climb 1 step twice. Now let's picture if we need to get up 6 steps. If we choose to do a one-step or a two step, respectively we still have 5 or 4 more steps to go. If we were to calculate how many choices we would have for 5 steps plus how many choices we have for 4 steps, we'd have to keep branching off and calculating 2 choices for each choice. This would be 2^N time complexity. However, if we imagine that we're already at step 3 then we know that we either got there from step 2 or step 1 (one step or two steps were taken). If we imagine we're already at step 4, we know we either got there from step 3 or step 2 (one step or two steps were taken). So basically we could always look at the two steps behind us and add the combinations of how many steps it took to get to those steps to see the combination of ways to get to our current step. We can iterate to n+1 (exclusive in Python) and use the Fibonacci sequence to find this while only saving the last two values. Once we break out of our iteration it is because we have gone to n and we will return the sum of the previous two behind n.

If n is 0 or 1 then return 1, else calculate the sum of the last two place's combinations until we've got to n and return the previous two's combinations summed.

17) https://leetcode.com/problems/contains-duplicate/

```
#O(N) Time, O(N) Space
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        seen = {}
        for num in nums:
            if num in seen:
                return True
            else:
                seen[num] = 'Hi'
        return False
```

Traverse array and check if current number is in the seen dictionary, return True if it is otherwise add it, if the end of the array has been reached without a True being returned, return False.

18) https://leetcode.com/problems/verifying-an-alien-dictionary/

```
#O(N) Time, where N is sum of lengths of all words, O(1) Space
class Solution(object):
    def isAlienSorted(self, words, order):
        order_index = {}
        for index, char in enumerate(order):
            order_index[char] = index
        for i in range(len(words) - 1):
            first_difference = False
            word_one = words[i]
            word_two = words[i+1]
            for char_pointer in range(min(len(word_one), len(word_two))):
                if word_one[char_pointer] != word_two[char_pointer]:
                    first_difference = True
                    if order_index[word_one[char_pointer]] > order_index[word_two[char_pointer]]:
                        return False
                    else:
                        break
            if len(word_one) > len(word_two) and not first_difference:
                    return False
        return True
```

We can solve this problem by creating an index of the order for the characters in the "order" string we are given. We will make a HashMap to keep track of this. We will then use a for loop to iterate through each character in the order and map them to their index. We will make another for loop to iterate through each word in the words array (remember, there can be more than two words, so we will compare two words at a time). The two words we will be comparing will be the current index in the iteration, and the word in the index directly after it. We will also need another for loop that iterates through each character within each word until we find a difference. We will set a pointer to the same character index in each word and see if they are different (if they are the same, we move on). If they are different, we check if the order of the first word's character (it is index in the dictionary) is higher than the second word's character. If it is return False, if it is not then we will break out of the loop because the words are sorted and we will compare the next two words. If we have gone through all the characters in the smaller word and have not found a difference between it and the characters in the larger word, we will check if the length of the words. If the length of the first word is larger than the second word, that means the second word has null characters that are smaller than the remaining characters in the first word and thus is false. If we have completed all iterations, we will return True because that would mean we have successfully broken out of all loops or there were words that are the same.

Use a dictionary to keep track of each the order of each character in the order we're given, iterate through the characters in two words and see if they are different and if they're different then see if the order of the first word's character is less than the order of the second word's character, if completed all iterations through smaller word's characters then check if length of first word is smaller than the second word, if it is not then return false, if all iterations are complete return True.

19) https://leetcode.com/problems/roman-to-integer/

```
#O(N) Time, O(1) Space
class Solution:
    def romanToInt(self, s: str) -> int:
        translation = {'M': 1000,'D': 500 ,'C': 100,'L': 50,'X': 10,'V': 5,'I': 1}
        output_sum = 0
        for i in range(0, len(s) - 1):
            if translation[s[i]] < translation[s[i+1]]:
                output_sum -= translation[s[i]]
            else:
                output_sum += translation[s[i]]
        return output_sum + translation[s[-1]]
```

We can solve this problem by traversing the string and comparing each character with the character next to it. If the current character is less than the next character, it means we have a number like 'IX'. If we encounter this, we can subtract the number we are currently at 'I' or 1. This way, during our next iteration when we add 10, our sum will have a net result of 9 added (IX). Our for loop will stop one before the end of the array to avoid an index out of bounds error, so we will want to always add the last character by returning the output sum plus the last character in the string.

Traverse string and compare the current character with the next character, if it is less then subtract it from the sum, if it higher then add it to the output sum, return output sum plus the last character in the string.

20) https://leetcode.com/problems/first-unique-character-in-a-string/

```
#O(N) Time, O(1) Space
class Solution:
    def firstUniqChar(self, s: str) -> int:
        seen = {}
        for char in s:
            if char in seen:
                seen[char] += 1
            else:
                seen[char] = 1
        for i in range(0, len(s)):
            if seen[s[i]] == 1:
                return i
        return -1
```

We can solve this problem by using a HashMap and doing two passes. We will make a for loop that iterates through all the characters in the string and check if they are already in the seen dictionary. If they are, we will increment their count. If they are not, we will set their count to 1. We will then make another for loop for our second pass. We will iterate through the string once more and see if the character at our current index has a count of 1 in our dictionary. If it does, we will return our current index. If nothing gets returned from our traversal, it means that the string is empty or has no unique values, so we return -1.

Use a HashMap and traverse the string twice, once to count the occurrence of each character in the string, another to see if there are any characters with the value of 1 the dictionary, return index if it exists and return -1 if nothing was returned.

21) https://leetcode.com/problems/most-common-word/

```
#O(P+B) Time for the size of paragraph and banned, O(P) Space for count of paragraph words
class Solution:
    def mostCommonWord(self, paragraph: str, banned: List[str]) -> str:
        for char in "?!';,.":
            paragraph = paragraph.replace(char, " ")
        seen = {}
        output_string = ''
        highest_count = 0
        paragraph = paragraph.lower().split()
        for word in paragraph:
            if word in banned:
                continue;
            elif word in seen:
                seen[word] += 1
            else:
                seen[word] = 1
            if seen[word] > highest_count:
                highest_count = seen[word]
                output_string = word
        return output_string
```

We can solve this problem by using a HashMap. We'll start by replacing all of the punctuation marks with the paragraph with an empty space to clean the data. We'll then initialize our seen HashMap and output string to be empty. We'll also initialize our count to 0. Afterwards, we will convert our paragraph to all lowercase characters and split it into an array of strings. We'll then traverse this array of strings and see if the word we're currently at is in banned. If it is, we will continue to the iteration (the next word). We will also check if the word is already in our seen HashMap. If it is, we'll increment it's count value. If it isn't we'll put it there and set it's count to 1, as this is it has had one appearance in the paragraph array so far. After we'll see if the count of the word is higher than our current count. If it is, we'll set the count to the count of appearances the word has appeared in our HashMap and set the output string as our current iteration's word. We'll repeat this until we have reached the end of the paragraph array and we'll have the highest count word and we'll return it as the output_string variable.

Replace punctuation in paragraph with empty spaces, initialize dictionary, output string, and highest count, convert paragraph to lowercase then split it into an array of strings, iterate through array checking if word is in banned string, if it is continue to next word, if it isn't then see if it's already in dictionary, if it is then increment it's value, if it isn't then put it in the dictionary with the value of 1, if the count of the word in the dictionary is greater than the highest count then set the highest count to the word's count and set the word as our output string, return output string at the end.

22) https://leetcode.com/problems/valid-palindrome/

```
#O(N) Time, O(1) Space
class Solution:
    def isPalindrome(self, s: str) -> bool:
        if len(s) == 0:
            return True
        left = 0
        right = len(s) -1
        letters_numbers = 'abcdefghijklmnopqrstuvwxyz0123456789'
        s = s.lower()
        while left < right:
            if s[left] != s[right]:
                if s[left] not in letters_numbers:
                    left += 1
                elif s[right] not in letters_numbers:
                    right -= 1
                else:
                    return False
            else:
                left +=1
                right -=1
        return True
```

We can solve this question using two pointers. We are only looking for alphanumeric characters (letters and digits), so we will create a reference string that holds all the letters and numbers that would possibly be compared. We will then create a left and right pointer that will compare two characters at a time. Since the check is not case-sensitive, we will convert it to lowercase. If our left and right pointers do not match, we check If they are actually letters or numbers. If they are not, we continue. If they are, we return false as the string is not a valid palindrome.

Use two pointers and a reference string of alphanumeric characters, convert string to lowercase, if there is not a match in characters under the pointers we check if they're alphanumeric, return false if they are but continue if they aren't.

23) https://leetcode.com/problems/single-number/

```
#O(N) Time, O(N) Space
#Another way to solve this is to return 2*sum(set(nums)) - sum(nums)
#This is because c = 2*(a+b+c) - (a+a+b+b+c)
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        seen = {}
        for num in nums:
            if num in seen.keys():
                seen[num] +=1
            else:
                seen[num] = 1
        for i in range(0, len(nums)):
            if seen[nums[i]] == 1:
                return nums[i]
```

We can solve this problem using the same solution as the 'first unique character in a string' problem. We create a HashMap of integers in our array as keys, and a count of their occurrence in the array as their values during our first pass. In our second pass, we see if there is a value in the seen dictionary of 1 and return the key (the num).

Use a HashMap and traverse the array twice, once to count the occurrence of each integer in the array, another to see if there are any integer keys with the value of 1 the dictionary and to return it.

24) https://leetcode.com/problems/missing-number/

```
#O(N) Time, O(1) Space
class Solution:
    def missingNumber(self, nums: List[int]) -> int:
        #N*(N+1)
        #-----
        # 2
        expected_sum = len(nums) * (len(nums) + 1) // 2
        actual_sum = sum(nums)
        return expected_sum - actual_sum
```

We can solve this problem by calculating the sum of the series and subtracting it by the sum of the array. This will give us the missing integer. The sum of the series is N*(N+1)/2 where N is the number of elements in the series (the length of the array in our case). The sum of the array is the sum of all the elements in the array.

Calculate N*(N+1)/2 minus the sum of elements in the array.

25) https://leetcode.com/problems/palindrome-number/

```
#O(N) Time, O(1) Space
class Solution:
    def isPalindrome(self, x: int) -> bool:
        if x < 0:
            return False
        result = 0
        input_int = x
        while x != 0:
            result *= 10
            result += x%10
            x //= 10
        return input_int == result
```

If the integer is negative then return false, otherwise create an output int variable that you will use to reverse the integer by using a while loop and multiplying the output int by ten to create a placeholder then adding the last digit of x by modding x by ten, then removing the last digit from x by dividing x by ten, compare the input int to the output and return true or false.

26) https://leetcode.com/problems/longest-common-prefix/

```
#O(N) Time, O(1) Space
class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        if len(strs) == 0:
            return ""
        shortest_word = min(strs)
        for i in range(0, len(shortest_word)):
            for compare in strs:
                if compare[i] != shortest_word[i]:
                    return shortest_word[:i]
        return shortest_word
```

We can solve this problem by iterating through each string in the array and comparing the characters until we find a difference. We'll first check and see if we received an empty array and return an empty string. We'll then get the shortest string in the array, as it will stop us from getting an index out of bounds error when we do our traversals. We'll then make a for loop that iterates through each character in the shortest word. Within that for loop we'll make another for loop that will iterate through every other string in the array. We'll then make an if/then statement that checks if the character at the ith index matches with the string it's being compared to. If it doesn't, we'll return the string up until that index. This will repeat until we have returned only the common characters and we'll return them.

Get the shortest string, iterate through the characters in that string and within each iteration if there is a mismatch between the character index in another string, return the shortest string up until that index and return the shortest string at the end. '

27) https://leetcode.com/problems/string-compression/

```
#O(N) Time, O(1) Space
class Solution:
    def compress(self, chars):
        write_pointer = read_pointer = 0
        while(read_pointer < len(chars)):
            char = chars[read_pointer]
            count = 0
            while(read_pointer < len(chars) and chars[read_pointer] == char):
                read_pointer += 1
                count += 1
            chars[write_pointer] = char
            write_pointer += 1
            if count > 1:
                for digit in str(count):
                    chars[write_pointer] = digit
                    write_pointer += 1
        return write_pointer
```

We can solve this problem by using two pointers. One pointer will read the values in the array and one will lag behind to write the values. We'll initialize both of these pointers to 0 and then create an outer while loop that will run while the read pointer hasn't reached the end of the list. We'll then make a count to keep track of the number of each characters. We'll then make an inner while loop that will run while the read pointer hasn't reached the end of the list and while the read pointer isn't pointing at a different character than what we're counting. We'll use this inner while loop to increment the read pointer and count by one. Once we've broken out of the inner while loop, we know that we've gone through and read the amount of same characters so we'll use the write pointer to write the character, we'll increment it by one more place and check if the count is greater than 1. If it is we'll know that we need to compress the string by writing a counter. We'll iterate through the string converted count (this way if we have a digit greater than 9 we will write it like "1", "0", this is a stupid edge case), we'll write the character and increment the write pointer. Once we're done with our iterations we'll return the write pointer as that will be the new length of the array.

Use two pointers and a counter, one pointer to read the length of the contiguous characters and increment the counter and one to write the character and count.

28) https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/

```
#O(N) Time, O(1) Space
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if len(prices) is 0:
            return 0
        maxprofit = 0
        for i in range(1, len(prices)):
            if prices[i] > prices[i-1]:
                maxprofit += prices[i] - prices[i-1]
        return maxprofit
```

A very simple way to solve this problem is to check if a profit is possible by comparing the current day's price with the previous day's price. If the price is higher on the current day than it was the day before, that is a profit and we'll want that deal. We'll start by checking if the length of our prices array is 0, if it is there is no profit to be had so we'll return 0. Otherwise, we'll initialize our max profit variable to 0. We'll then iterate through the prices array starting from the second day and compare each day with the day before. If the price on the current day is higher then we'll subtract the previous day's price from the current day's price and add the difference to the max profit variable and after all our iterations we'll return the max profit.

If array is empty return 0, otherwise traverse array and compare each index's price with the index before it. If it's lower, subtract previous day's price from current price and add it to the max profit, return max profit at the end.

29) https://leetcode.com/problems/subdomain-visit-count/

```
#O(N) Time, O(N) Space
class Solution:
    def subdomainVisits(self, cpdomains):
        visit = {}
        for count_domain in cpdomains:
            count, domain = count_domain.split(" ")
            subs = domain.split(".")
            subs[0] = domain
            index = domain.find(".")
            subs[1] = domain[index +1:]
            for sub in subs:
                if sub not in visit:
                    visit[sub]  = int(count)
                else:
                    visit[sub] += int(count)
        pairs = []
        for element in visit:
            visit_count = "{} {}".format(visit[element], str(element))
            pairs.append(visit_count)
        return pairs
```

We can solve this problem by splitting the string twice and using a hashmap to increment the count for each subdomain. We'll start by initializing our visit dictionary. We'll then split our input string on the space to separate the count and the entire domain into two different indices. Next we'll create our list of subdomains by splitting the domain string on the dots. Since question wants the output to be in the format "x a.b.c, y b.c, z c", we'll need to override the first and second index of our subs list. We'll make the entire domain as the first element in our subdomain list, and the second element the slice of the domain after the first dot. The third element won't be overridden if there are three subdomains so it will still be "com" or "org" etc. We'll then use a for loop to iterate through the subs list and check if each subdomain is in the visit dictionary. If it is, we'll increment it's current count value by the count once more. If it's not, we'll put the suddomain in the dictionary with it's initial count as the value. We will then initialize our pairs dictionary. Next we'll use a for loop to iterate through the elements in the visit dictionary and make a string of the subdomain and it's visited count. We'll then append each string to the pairs list and return it.

Make a hashmap and iterate through each element of the input list splitting it on the space to get the count and domain, then splitting it on the dots to get each subdomain but overriding the first two elements with the complete domain and the domain with the first portion removed, then put each subdomain in a dictionary with the initial count as its value if not already there, incrementing the count if it is, then initialize an output list where you'll use a for loop to iterate through the dictionary and make a string of the subdomain and count of visits that you'll append to the array and return it.

30) https://leetcode.com/problems/logger-rate-limiter/

```
#O(N) Time, O(N) Space
class Logger:

    def __init__(self):
        self.dictionary = {}

    def shouldPrintMessage(self, timestamp: int, message: str) -> bool:
        for i in list(self.dictionary):
            if timestamp - self.dictionary[i] >= 10:
                del self.dictionary[i]
            else:
                break
        if message not in self.dictionary:
            self.dictionary[message] = timestamp
            return True
        else:
            return False
```

The easiest way to solve this question is for each message see if it's in a HashMap we make. If it isn't we'll add the message to the dictionary as a key with the value of the timestamp. If it is then we'll see if our current timestamp minus the message's timestamp value in the dictionary is greater than or equal to 10. If it is, we'll update the value to our current timestamp. If it isn't then we'll return False. This solution will be constant time, but will take a ton of space since we aren't doing anything with old messages.

For a better trade of time for space (the solution shown above), we can iterate through the beginning of our dictionary and check for values that are older than 10 seconds from our current timestamp and delete them. Then we'll just need to check if the message is in our dictionary. If it isn't we'll add it with the timestamp, if it is we'll return False because that will mean it isn't older than ten seconds.

Initialize a dictionary, then traverse the dictionary for message values are that are older than 10 seconds from our current timestamp and delete them, then check if the message we were just given is in the dictionary, if not add it to the dictionary with the timestamp as it's value, if it is then return False because it would have been deleted if it was expired.

31) https://leetcode.com/problems/first-bad-version/

```
#O(log N) Time, O(1) Space
class Solution:
    def firstBadVersion(self, n):
        left, right = 1, n
        while left < right:
            mid = (left + right)//2
            if isBadVersion(mid) is True:
                right = mid
            else:
                left = mid + 1
        return left or right
```

We can solve this problem using a binary search. We'll initialize our left pointer to 1 (our first version) and our right pointer to n (our last version). Then we'll use a while loop that will run until the left and right pointers are right next to eachother. During each iteration we'll initialize our middle pointer as the mean of the left and right pointers (left + right)//2. If the middle pointer is a bad version, we'll move further down the left side of the array by setting our right pointer to the middle pointer. Remember, we want to know the earliest bad version. If middle isn't bad then that means the first bad version came later than where the mid pointer is so we'll set our left pointer one after the middle pointer. Once we break out of our loop it means that our left and right pointer have overlapped so you can return either pointer.

Solve using binary search, set left pointer to 1 and right pointer to n, while the left pointer hasn't reached the right pointer set the middle pointer as the average of them both, check if the middle pointer is bad, if it is then set the right pointer to the mid pointer, if it isn't then set the left pointer one to the right of the mid pointer, return left or right pointer once out of the loop.

32) https://leetcode.com/problems/linked-list-cycle/

```
#O(N) Time, O(1) Space
class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        if head is None or head.next is None:
            return False
        slow = head
        fast = head.next
        while slow != fast:
            if fast is None or fast.next is None:
                return False
            slow = slow.next
            fast = fast.next.next
        return True
```

We can detect a cycle by either storing the values in a dictionary (O(N)) Space, or we can use Floyd's algorithm. We'll start off by ensuring the head and first position aren't none, we'll return False if they are. Next we'll initialize a slow pointer at the head and a fast pointer at head.next. We'll then create a while loop that will run while the fast pointer hasn't reached the slow pointer. If the fast pointer has reached the end of the list we'll return False because we'll know there isn't a cycle, so we'll make a conditional statement that checks that. Otherwise, we'll increment slow by one index and fast by two. If the fast pointer has caught up to the slow pointer we'll break out of our loop and return True.

Check if head or head.next is None and return False if one is, set a slow pointer to head and a fast pointer to head.next, create a while loop that will run as long as slow hasn't reached fast, if fast has reached the end of the list return False otherwise increment slow by one and fast by two, return True if loop gets broken.

33) https://leetcode.com/problems/sum-of-two-integers/

```
#O(1) Time, O(1) Space
class Solution:
    def getSum(self, a: int, b: int) -> int:
        x, y = abs(a), abs(b)
        if x < y:
            return self.getSum(b, a)

        if a > 0:
            sign = 1
        else:
            sign = -1

        if a * b >= 0:
            while y is not 0:
                x, y = x ^ y, (x & y) << 1
        else:
            while y is not 0:
                x, y = x ^ y, (~x & y) << 1
        return x * sign
```

We can solve this problem by using bit manipulation. Inherently there are a lot of use cases when summing both positive and negative numbers because a and b could be lesser or greater than the other, one could be negative, both could be negative, both could be positive, so we want to reduce the number of use cases to two. We'll either be dealing with the sum or difference of two positive numbers. It will either be x plus why where x is greater, or x minus y where x is greater. We'll start by converting the two integers a and b to their absolute values and setting x and y equal to them. Both of the two cases we mentioned previously require x to be the greater, so we'll check if that is the case and if not we'll re-run our function with the inputs switched. We'll then check our initial a value and see if it's negative or positive or not so that we'll know what our result integer should be. Next we'll see if the product of our two input integers are positive. If it is positive then we know we're adding two positive integers. If it's negative then we know we're taking the difference instead. For the addition computation we'll use a loop that will continue while while there are still values to carry. It will be consist of setting x to (x XOR y), the true values where one but not both bits aren't 1, and y to (x AND y), where both bits are 1 shifted left by 1 to add in the next iteration. For the difference computation, the only difference is we'll be using the negation of x due to our second test case. Return x (the answer) multiplied by the sign so it's either positive or negative.

Set x to the absolute value of a, and y to the absolute value of y, if x less than y re-run the function with the inputs switched, then check if a is positive or negative and set a sign variable that will be multiplied to the output to return a positive or negative result, then check if the product of a and b is positive, if so then compute the sum of the two numbers where x is greater by using a loop that will run while y (the carry value) is greater than 0, each iteration will set x equal to x XOR y, and y equal to x AND y shifted left by 1, if the product of a and b Is negative then compute the difference of the two numbers by doing the same thing as the sum but ANDing y with the negated x, return x multiplied by the sign.

34) https://leetcode.com/problems/number-of-1-bits/

```
#O(1) Time, O(1) Space
class Solution:
    def hammingWeight(self, n: int) -> int:
        count = 0
        while n is not 0:
            count += 1
            n &= n-1
        return count
```

We can solve this problem by using bit manipulation. If we AND our integer with n-1, it will flip the least significant bit in the integer to 0. Each time we do this we can increment a counter and once the integer has gone to 0, we will know we have no more bits to flip.

Initialize a count then traverse the binary integer, while the input integer is not 0 we will increment the count then AND the input integer with n-1, return the final count.

35) https://leetcode.com/problems/maximum-depth-of-binary-tree/

```
#Recursion
#O(N) Time, O(log(N)) Space
class Solution:
    def maxDepth(self, root):
        if root is None:
            return 0
        else:
            left_height = self.maxDepth(root.left)
            right_height = self.maxDepth(root.right)
            return max(left_height, right_height) + 1
#Iterative
#O(N) Time, O(log(N)) Space
    def maxDepth(self, root):
        stack = []
        if root is not None:
            stack.append((1, root))
        depth = 0
        while stack != []:
            current_depth, root = stack.pop()
            if root is not None:
                depth = max(depth, current_depth)
                stack.append((current_depth + 1, root.left))
                stack.append((current_depth + 1, root.right))
        return depth
```

Make a recursive function that returns 0 if the current node is None, otherwise set the left height to a recursive call of the current node's left child, do the same with the right and return the max height of the the left and right height variables plus one (since we're at the parent node in each recursive call).

We can solve this iteratively by making a stack and initially appending the current depth of 1 and the root if it's not None. Then we create a depth counter and loop while the stack isn't empty, setting the current depth and root to the popped depth and root value. If the root isn't None there are still children to move down so we'll first get the depth by comparing the current_depth we just popped to the depth counter and keeping it if greater. Then we'll append the left and right child nodes to the stack incrementing the current_depth by one each time, returning final depth once the stack is empty.

Make a iterative function that uses a depth counter and a stack that keeps depth and root pair values, while the stack isn't empty pop the current depth and root value, if the root isn't None then compare it's current depth with the depth counter and append the left and right children to the stack, incrementing the current depth by one, return the depth counter when stack is empty.

https://leetcode.com/problems/maximum-product-subarray/solution/ TODO

```python
#O(N) Time, O(1) Space
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        prefix, suffix, max_so_far = 0, 0, float('-inf')
        for i in range(len(nums)):
            prefix = (prefix or 1) * nums[i]
            suffix = (suffix or 1) * nums[~i]
            max_so_far = max(max_so_far, prefix, suffix)
        return max_so_far
```

https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/

```python
#O(log N) Time, O(1) Space
class Solution:
    def findMin(self, nums: List[int]) -> int:
        left, right = 0, len(nums) - 1
        while left < right:
            middle  = (left + right) // 2
            if nums[middle] < nums[right]:
                right = middle
            else:
                left = middle + 1
        return nums[left]
```

Perform a binary search to find the element that should be on the very left, but since the array could be rotated our while loop will run while the left pointer is less than right and we'll check if the middle element is less than the right, if so then that portion of the array isn't sorted so we'll move right to the the middle pointer and continue downwards, otherwise we'll set left one to the right of middle so we can move to the sorted portion of the array for the next iteration, return the left pointer when out of loop.

36) https://leetcode.com/problems/product-of-array-except-self/

```
#O(N) Time, O(1) Space
class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:
        length = len(nums)
        answer = [0]*length
        answer[0] = 1
        for i in range(1, length):
            answer[i] = nums[i - 1] * answer[i - 1]
        right = 1
        for i in range((length) -1, -1, -1):
            answer[i] = answer[i] * right
            right *= nums[i]
        return answer
```

We can solve this problem by taking the product of each element from the left and right of each index and multiplying them together. We could create two arrays, one of all the product of all elements to the left of the current index and one of all the product of all the elements to the right of the current index and then multiply the index of the left array with the right away. However, that would be O(N) Time, and O(N) Space. What we could do instead is not make a right product array but have a running right product total. We'll start by saving the length of the array so that we don't have to keep writing "len(nums)". Then we'll initialize an array of 0s equal to the length of the input array. We'll set the first element in the output array equal to 1 because nothing is to the left of the first element in the input array and multiplying an int by 1 will equal itself. We'll then use a for loop to traverse the answer array and multiply each previous element by the element in the input array. After, we'll set right to 1 (for the same reason as setting the first element in our output array to 1). We'll then traverse the answer array backwards now and for each element we'll multiply it by the running right total and then increment the running right total by the product of the current index's element. This way we'll be multiplying each of the two previous elements like we did initially for the answer array. We'll return the answer array once our second traversal is done.

Initialize an array of 0's equal to the length of our input array, traverse the array twice, once forward and multiplying each previous element with the previous element from input array, once backward and multiplying each previous element with the running right product total and then incrementing the total by the product of itself and the current input array index.

37) https://leetcode.com/problems/valid-anagram/

```
#O(N log N) Time, O(1) Space
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        if len(s) != len(t):
            return False
        return sorted(s) == sorted(t)

#O(N) Time, O(1) Space
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        if len(s) != len(t):
            return False
        seen_one = {}
        seen_two = {}
        for char in s:
            if char in seen_one:
                seen_one[char] += 1
            else:
                seen_one[char] = 1
        for char in t:
            if char in seen_two:
                seen_two[char] += 1
            else:
                seen_two[char] = 1
        return seen_one == seen_two
```

Either sort both strings and compare the result, or add the characters of each string to two different dictionaries and compare the result.

38) https://leetcode.com/problems/reverse-bits/

```
#O(1) Time, O(1) Space
class Solution:
    def reverseBits(self, n: int) -> int:
            result, power = 0, 31
            while n is not 0:
                result += (n & 1) << power
                n >>= 1
                power -= 1
            return result
```

We can solve this problem by doing bit-by-bit operations. Since we are given a 32-bit integer, we can keep that in mind when setting the position of our bits in our output binary number. We'll initialize our result to 0 and our power to 31 (binary number places go from 2^0 to 2^31, 32 total). Then we'll create a while loop that will run while we still have bits greater than 0 to compute. We'll then execute an AND operation between our input integer (n) and 1, so that we can get a 1 if the rightmost bit is a 1, otherwise 0. We'll shift it to the appropriate power's place (31 – i) and add it to the result. We'll shift the binary number to the right so that we can focus on the bit to the left of our current bit during the next iteration. Also we will decrement the power so that we can put the next bit to the right of the bit we just placed in the result. Once our input integer is 0 or all power places have been filled, we will return our result.

Initialize result int to 0 and power to 31, do a bit operation of n AND 1 shifted left to the current power, shift n right once and decrement power, return result integer.

39) https://leetcode.com/problems/squares-of-a-sorted-array/ TODO

```
#O(N) Time, O(N) Space
class Solution:
    def sortedSquares(self, A: List[int]) -> List[int]:
        B = [0] * len(A)
        l, r = 0, len(B) - 1
        while l <= r:
            left, right = abs(A[l]), abs(A[r])
            if left > right:
                B[r - l] = left * left
                l += 1
            else:
                B[r - l] = right * right
                r -= 1
        return B
```

40) https://leetcode.com/problems/invert-binary-tree TODO

```
#Recursive
#O(N) Time, O(H) Space
class Solution:
    def invertTree(self, root: TreeNode) -> TreeNode:
        if root is None:
            return None
        right = self.invertTree(root.right)
        left = self.invertTree(root.left)
        root.left = right
        root.right = left
        return root
#Iterative
#O(N) Time, O(N) Space
class Solution:
    def invertTree(self, root):
        if root is None:
            return None
        parent = [root]
        while len(parent):
            children = []
            for node in parent:
                node.left, node.right = node.right, node.left
                if node.left is not None:
                    children.append(node.left)
                if node.right is not None:
                    children.append(node.right)
            parent = children
        return root
```

https://leetcode.com/problems/merge-two-binary-trees TODO

```
#Recursive
#O(M) Time, O(M) Space Worst, O(log M) Average
class Solution:
    def mergeTrees(self, t1, t2):
        if t1 is None:
            return t2
        if t2 is None:
            return t1
        t1.val += t2.val
        t1.left = self.mergeTrees(t1.left, t2.left)
        t1.right = self.mergeTrees(t1.right, t2.right)
        return t1
```

41) https://leetcode.com/problems/design-hashmap/ TODO

```
#O(Keys/Buckets) Time, O(Keys+Buckets) Space
class ListNode:
    def __init__(self, key, val):
        self.pair = (key, val)
        self.next = None

class MyHashMap:
    def __init__(self):
        self.map = 1000;
        self.hash = [None]*self.map

    def put(self, key, value):0
        index = key % self.map
        if self.hash[index] == None:
            self.hash[index] = ListNode(key, value)
        else:
            current = self.hash[index]
            while True:
                if current.pair[0] == key:
                    current.pair = (key, value) #update
                    return
                if current.next == None: break
                current = current.next
            current.next = ListNode(key, value)

    def get(self, key):
        index = key % self.map
        current = self.hash[index]
        while current is not None:
            if current.pair[0] == key:
                return current.pair[1]
            else:
                current = current.next
        return -1

    def remove(self, key):
        index = key % self.map
        current = previous = self.hash[index]
        if current is None:
            return
        if current.pair[0] == key:
            self.hash[index] = current.next
        else:
            current = current.next
            while current is not None:
                if current.pair[0] == key:
                    previous.next = current.next
                    break
                else:
                    current, previous = current.next, previous.next
```

42) https://leetcode.com/problems/3sum/ TODO

```
#O(N^2) Time, O(N) Space
class Solution(object):
         def threeSum(self, nums):
                  result = []
                  nums.sort()
                  length = len(nums)
                  for i in range(0, length-2):
                           if nums[i]>0: break
                           if i>0 and nums[i] == nums[i-1]:continue
                           left, right = i+1, length-1
                           while left<right:
                                    total = nums[i]+nums[left]+nums[right]
                                    if total<0:
                                             left += 1
                                    elif total>0:
                                             right -= 1
                                    else:
                                             result.append([nums[i], nums[left], nums[right]])
                                             while left < right and nums[left]==nums[left+1]:
                                                      left += 1
                                             while left < right and nums[right]==nums[right-1]:
                                                      right-=1
                                             left+=1
                                             right-=1
                  return result
```

https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree TODO

```
#O(N) Time, O(N) Space for the output, O(log N) for stack
class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> TreeNode:
        def helper(left, right):
            if left > right:
                return None
            mid = (left + right) // 2
            root = TreeNode(nums[mid])
            root.left = helper(left, mid - 1)
            root.right = helper(mid + 1, right)
            return root
        return helper(0, len(nums) - 1)
```

https://leetcode.com/problems/same-tree TODO

```
#Recursive
#O(N) Time, O(log(N)) Best, O(N) Worst for Unbalanced Tree
class Solution:
    def isSameTree(self, p, q):
        if p is None and q is None:
            return True
        if p is None or q is None:
            return False
        if p.val != q.val:
            return False
        return self.isSameTree(p.left, q.left) and \
               self.isSameTree(p.right, q.right)
```

```
#Iterative
#O(N) Time, O(log(N)) Best, O(N) Worst for Unbalanced Tree
class Solution:
    def isSameTree(self, p, q):
        stack = [(p,q)]
        print(stack)
        while stack :
            x,y = stack.pop()
            if x is None and y is None:
                continue
            if x is None or y is None:
                return False
            if x.val != y.val:
                return False
            else:
                stack.append((x.left,y.left))
                stack.append((x.right,y.right))
        return True
```

https://leetcode.com/problems/trim-a-binary-search-tree TODO

```
#O(N) Time, O(N) Space
class Solution(object):
    def trimBST(self, root, L, R):
        def trim(node):
            if node is None:
                return None
            elif node.val > R:
                return trim(node.left)
            elif node.val < L:
                return trim(node.right)
            else:
                node.left = trim(node.left)
                node.right = trim(node.right)
                return node
        return trim(root)
```

https://leetcode.com/problems/balanced-binary-tree TODO

```
#O(N) Time, O(N) Space if Tree is Unbalanced
class Solution:
    def isBalancedHelper(self, root: TreeNode) -> (bool, int):
        if root is None:
            return True, -1
        leftIsBalanced, leftHeight = self.isBalancedHelper(root.left)
        if leftIsBalanced is False:
            return False, 0
        rightIsBalanced, rightHeight = self.isBalancedHelper(root.right)
        if rightIsBalanced is False:
            return False, 0
        else:
            return (abs(leftHeight - rightHeight) < 2), 1 + max(leftHeight, rightHeight)

    def isBalanced(self, root: TreeNode) -> bool:
        return self.isBalancedHelper(root)[0]
```

https://leetcode.com/problems/symmetric-tree TODO

```
#Recursive
#O(N) Time, O(N) Space for Unbalanced Tree
class Solution:
  def isSymmetric(self, root):
    if root is None:
      return True
    else:
      return self.isMirror(root.left, root.right)

  def isMirror(self, left, right):
    if left is None and right is None:
      return True
    if left is None or right is None:
      return False
    if left.val == right.val:
        if self.isMirror(left.left, right.right) \
        and self.isMirror(left.right, right.left):
            return True
        else:
            return False

#Iterative
#O(N) Time, O(N) Space for Unbalanced Tree
class Solution:
  def isSymmetric(self, root):
      if root is None:
          return True
      stack = [(root.left, root.right)]
      while stack != []:
          left, right = stack.pop()
          if left is None and right is None:
              continue
          if left is None or right is None:
              return False
          if left.val == right.val:
              stack.append((left.left, right.right))
              stack.append((left.right, right.left))
          else:
              return False
      return True
```

https://leetcode.com/problems/count-and-say TODO

```
#O(N^2) Time, O(N^2) Space
class Solution:
    def countAndSay(self, n: int):
        def next_sequence(string: str):
            if n == 1:
                return string
            result = ""
            length = len(string)
            i=0
            while i < length:
                count = 1
                while i + 1 < length and string[i] == string[i+1]:
                    count += 1
                    i += 1
                result = result + str(count)
                result = result  + string[i]
                i += 1
            return result
        string = "1"
        for i in range(n-1):
            string = next_sequence(string)
        return string
```

https://leetcode.com/problems/longest-substring-without-repeating-characters/

```
#O(N) Time, O(N) Space
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        seen = {}
        max_length = start = 0
        for i, char in enumerate(s):
            if char in seen and start <= seen[char]:
                start = seen[char] + 1
            else:
                max_length = max(max_length, i - start + 1)
            seen[char] = i
        return max_length
```

We can solve this problem by using a HashMap to keep track of already seen characters and both max length/start integer variables to track the max length we have so far in each iteration and where a valid substring candidate begins. We'll use a for loop enumeration to traverse the input string so that we can keep track of indices and also iterate through the characters (for readability). We'll check if the character in each iteration is in our dictionary, also if it is whether our start value is less than or equal to it's value in the dictionary. If it is then we know we should abandon this substring candidate and set the start counter to the index right after where we last saw the character we're repeating. If the character isn't in our dictionary that means it hasn't been repeated yet so we'll first check if the current substring we're considering's length is greater than the maximum length we have saved. If it is, that will be our new max length. We'll then add our character and index to the dictionary or override it's value. Once we're done with our traversal we'll return the maximum length.

Make a dictionary then initialize a max length and start value, enumerate through the input string while checking if the current character is in the seen dictionary and if it is that the substring start value is less than or equal to the value of the character in the dictionary, if so then set start to the character's value plus 1, otherwise compare the current max length with the current index minus start + 1 (indexing starts at 0), add the current character to the dictionary with the index as the value and return max length once out of the loop.

https://leetcode.com/problems/repeated-substring-pattern/ TODO

```
#O(N) Time, O(N) Space
class Solution:
    def repeatedSubstringPattern(self, s: str) -> bool:
        length = len(s)
        lookup = [0] * length
        for i in range(1, length):
            j = lookup[i - 1]
            while j > 0 and s[i] != s[j]:
                j = lookup[j - 1]
            if s[i] == s[j]:
                j += 1
            lookup[i] = j
        l = lookup[length - 1]
        return l != 0 and length % (length - l) == 0
```

https://leetcode.com/problems/number-of-islands/

```
#O(M*N) Time, O(min(M,N)) Space
class Solution:
    def numIslands(self, grid):
        if grid is None:
            return 0
        count = 0
        for i in range(0, len(grid)):
            for j in range(0, len(grid[0])):
                if grid[i][j] == '1':
                    self.dfs(grid, i, j)
                    count += 1
        return count

    def dfs(self, grid, i, j):
        if i<0 or j<0 or i>=len(grid) or j>=len(grid[0]) or grid[i][j] == '0':
            return
        grid[i][j] = '0'
        self.dfs(grid, i-1, j)
        self.dfs(grid, i+1, j)
        self.dfs(grid, i, j-1)
        self.dfs(grid, i, j+1)
```

We can use Depth First Search to solve this program. We'll first check if we've gotten an empty input and return 0 (no islands) if we have. Next we'll initialize a count that we'll increment each time we come across a '1'. We'll create two for loops for iterating through the sea (the 2D array) and we'll use two variables for the iteration, i (the rows) and j (the columns). If we come across any '1's during our traversal we know we've hit an island we'll run a dfs to sink it. We'll increment the count and return it once we have traversed the full sea. In our dfs helper function we'll first take care of some error handling. We'll first check if i or j are less than 0, or if i or j are greater than the length of the grid's width or height, or if the element we're currently looking at is '0' then we'll return. We do this because for each '1' element we encounter during our DFS, we're also calling DFS on that element which calls it on all the elements surrounding it until we have sunk the island by turning all the '1's to '0's or we've gone out of bounds.

Make a DFS function that first checks if the current index is out of bounds and if not turns the element to a '0' then calls itself on all four surrounding index elements, in the solution function create two for loops that traverse the rows and columns of the grid and increments a count each time it finds a '1' and call the DFS function on the encounter, return final count.

https://leetcode.com/problems/add-two-numbers/

```
#O(max(M,N)) Time, O(max(M,N)) Space
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        dummy = current = ListNode(0)
        carry = 0
        while l1 or l2 or carry:
            if l1:
                carry += l1.val
                l1 = l1.next
            if l2:
                carry += l2.val
                l2 = l2.next
            current.next = ListNode(carry%10)
            current = current.next
            carry //= 10
        return dummy.next
```

Make a dummy node, current node, and a carry value all with value 0, while either input nodes or the carry node aren't none we'll iterate through all lists adding both input node values to the carry value and making new nodes in the third LL of the carry modded by a 10's place stepping to the node and dividing carry by a 10's place, returning everything after the dummy node when done.