# <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.**

# <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.**

# <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 w hich 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.**

# <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.**

# <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.**

# <https://leetcode.com/problems/valid-palindrome-ii/>

**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.**

<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.**

# <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/reverse-linked-list/>

#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.**

# <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.**

# <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.**

# <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/reverse-integer/>

**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.**

# <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.**

# <https://leetcode.com/problems/add-binary/>

**class** **Solution**:

**def** **addBinary**(self, a, b) -> str:

x, y = int(a, **2**), int(b, **2**)

**while** y:

answer = x ^ y

carry = (x & y) << **1**

x, y = answer, carry

**return** bin(x)[**2**:]

# <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 and continue to the next iteration (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.**

# <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.**

# <https://leetcode.com/problems/roman-to-integer/>

#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.**