CS Interview Prep

Top Questions: Table of Contents

A. List/Arrays, stacks, queues

1. Implement a Stack with Queues

```
# implements a stack with two queues
import queue

class Stack:
    def __init__(self):
        self.queue1, self.queue2 = queue.Queue(), queue.Queue()

    def push(self, x):
        self.queue2.put(x)
        while not self.queue1.empty():
            self.queue2.put(self.queue1.get())
        self.queue1, self.queue2 = self.queue2, self.queue1

    def pop(self):
        return self.queue1.get();




if __name__ == "__main__":
    stack = Stack()
    stack.push(1)
    print(stack.pop())
    stack.push(2)
    stack.push(3)
    print(stack.pop())
    stack.push(4)
    print(stack.pop())
    print(stack.pop())
```

Push: O(n)
Pop: O(1)

Implement a queue with stacks

```
# implements a queue with 2 stacks

class stack:
    def __init__(self):
        self.stack1, self.stack2 = [], []

    def enqueue(self, x):
        self.stack1.append(x)

    def dequeue(self):
        if len(self.stack2) == 0:
            while (len(self.stack1) != 0):
                self.stack2.append(self.stack1.pop())
        return self.stack2.pop()


if __name__ == "__main__":
    stack1 = stack()
    stack1.enqueue(1)
    print(stack1.dequeue())
    stack1.enqueue(2)
    stack1.enqueue(3)
    print(stack1.dequeue())
    stack1.enqueue(4)
    print(stack1.dequeue())
    print(stack1.dequeue())
```

Enqueue: O(1)
Dequeue: O(n)

Valid Parentheses

Given a string ‘{()}[]’, determine if the string is a valid expression of brackets.

```
    def isValid(self, s):
        stack, match = [], {"(":")", "[":"]", "{":"}"}

        for i in s:
            if i in match.keys():
                stack.append(match[i])
            else:
                try:
                    popped = stack.pop()
                    if popped != i:
                        return False
                except:
                    return False
        return len(stack) == 0
```

Min Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time. IDEA: either have 2 arrays, or an array of tuples (store info for each req)

push(x) -- Push element x onto stack.
pop() -- Removes the element on top of the stack.
top() -- Get the top element.
getMin() -- Retrieve the minimum element in the stack.

Example:
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> Returns -3.
minStack.pop();
minStack.top();   --> Returns 0.
minStack.getMin();   --> Returns -2.

Idea 1: 2 stacks

```
    def __init__(self):
        self.stack = []
        self.min = []


    def push(self, x):
        self.stack.append(x)
        if not self.min or self.min[-1] >= x:
            self.min.append(x)


    def pop(self):
        x = self.stack.pop()
        if x == self.min[-1]:
            self.min.pop()


    def top(self):
        return self.stack[-1]


    def getMin(self):
        return self.min[-1]
```

IDEA 2: USE A SINGLE ARRAY, STORE TUPLES

```
    def __init__(self):
        self.stack = []


    def push(self, x):
        currMin = self.getMin()
        if not self.stack or currMin > x:
            currMin = x
        self.stack.append([x, currMin])




    def pop(self):
        self.stack.pop()


    def top(self):
        if not self.stack:
            return None
        return self.stack[-1][0]


    def getMin(self):
        if not self.stack:
            return None
        return self.stack[-1][1]
```

Find Unique Element in Array

Find an element that appears **ONCE** in an array where all other elements appear **TWICE**
Extension: what about two elements? or other elements appear thrice?

Idea 1: Sorting. First sort array , then traverse and find non-matching pair: O(n log n) time, O(1) space

Idea 2. Hashing: store into dictionary, or array. O(n) time, O(n) space.

Idea 3. Best solution: use XOR. (**XOR for DOUBLY ELEMENTS**)

```
def find_single_element(arr):
      elem = None
      for i in arr:
            elem ^= i
      return elem
```

Time: O(n), space: O(1)

Extension: Two elements appear once, find those two

```
All the bits that are set in xor will be set in one non-repeating element
(x or y) and not in other. So if we take any set bit of xor and divide the
elements of the array in two sets – one set of elements with same bit set
and other set with same bit not set. By doing so, we will get x in one set
and y in another set. Now if we do XOR of all the elements in first set, we
will get first non-repeating element, and by doing same in other set we
will get the second non-repeating element.

Let us see an example.

   arr[] = {2, 4, 7, 9, 2, 4}

1) Get the XOR of all the elements.

     xor = 2^4^7^9^2^4 = 14 (1110)

2) Get a number which has only one set bit of the xor.
   Since we can easily get the rightmost set bit, let us use it.

     set_bit_no = xor & ~(xor-1) = (1110) & ~(1101) = 0010

   Now set_bit_no will have only set as rightmost set bit of xor.
```

```
3) Now divide the elements in two sets and do xor of elements in each set,
and we get the non-repeating elements 7 and 9.
```

Extension: Other elements appear thrice

```
Run a loop for all elements in array. At the end of every iteration,
maintain following two values.

ones: The bits that have appeared 1st time or 4th time or 7th time .. etc.

twos: The bits that have appeared 2nd time or 5th time or 8th time .. etc.

Finally, we return the value of 'ones'

How to maintain the values of 'ones' and 'twos'?
'ones' and 'twos' are initialized as 0. For every new element in array,
find out the common set bits in the new element and previous value of
'ones'. These common set bits are actually the bits that should be added to
'twos'. So do bitwise OR of the common set bits with 'twos'. 'twos' also
gets some extra bits that appear third time. These extra bits are removed
later.
Update 'ones' by doing XOR of new element with previous value of 'ones'.
There may be some bits which appear 3rd time. These extra bits are also
removed later.

Both 'ones' and 'twos' contain those extra bits which appear 3rd time.
Remove these extra bits by finding out common set bits in 'ones' and
'twos'.

def getSingle(arr, n):
    ones = 0
    twos = 0

    for i in range(n):
        # one & arr[i]" gives the bits that are there in both 'ones' and
new element from   arr[]. We add these bits to 'twos' using bitwise OR
        twos = twos | (ones & arr[i])

        # one & arr[i]" gives the bits that are there in both 'ones' and
new element from arr[]. We add these bits to 'twos' using bitwise OR
        ones = ones ^ arr[i]

        # The common bits are those bits which appear third time. So these
bits should not be there in both 'ones' and 'twos'. common_bit_mask
contains all these bits as 0, so that the bits can be removed from 'ones'
```

```
and 'twos'
        common_bit_mask = ~(ones & twos)

        # Remove common bits (the bits that appear third time) from 'ones'
        ones &= common_bit_mask

        # Remove common bits (the bits that appear third time) from 'twos'
        twos &= common_bit_mask
    return ones
```

Find Missing Numbers in Array

Given an array where elements are between 1 and n (length of array), some elements are missing and some appear twice/once. Find elements that do not appear in the array.

E.g.
Input: [4,3,2,7,8,2,3,1]
Output: [5,6]

Idea: Iterate one first, marking the number that the index points to as negative. Then we loop through again and get the indexes that corresponds to a positive number (since nothing on the list points there), indicating the missing element.

Time: O(n), Space: O(1)

```
def findDisappearedNumbers(self, nums):

    toReturn = []

    for i in range(len(nums)):
        index = abs(nums[i]) -1
        nums[index] = -1 * abs(nums[index])
    for i in range(len(nums)):
        if nums[i] > 0:
            toReturn.append(i+1)
    return toReturn
```

Replace Element with Product of Neighbors

Replace every element in index by multiplication of previous and next, except first element of output is product of first and second; last element of output is product of second-to-last and last

E.g. Input = [2, 3, 4, 5, 6]
Output = [6, 8, 15, 24, 30]

Naive solution 1: create auxiliary array with copy of current elements, then traverse copied array and update output. TIme: O(n), space: O(n). Can we do better?

Solution 2: Keep track of previous element in loop. Time: O(n), space: O(1)

```
def multiply_output(arr):
    if len(arr) == 1:
        return

    prev = arr[0]
    arr[0] *= arr[1]

    for i in range(1, len(arr)-1):
        curr = arr[i]
        arr[i] = prev * arr[i+1]
        prev = curr

    arr[len(arr)-1] *= prev
    return arr
```

Reversing an array

```
# iterative way, Time: O(n), space: O(1)
def iterative_reverse(arr):
    front, end = 0, len(arr)-1

    while (front <= end):
        arr[front], arr[end] = arr[end], arr[front]
        front, end = front + 1, end - 1

def recursive_reverse(arr):
    if len(arr) <= 1:
        return
    arr = arr[-1] + recursive_reverse(arr[1:-1]) + arr[0]
```

Rotate Array

Takeaway: nums = nums[-k:] + nums[:-k] doesn’t work → changes by assignment in function! To change it, use **nums[:].**

```
def rotate(self, nums, k):
    k = k % len(nums)
    nums[:] = nums[-k:] + nums[:-k]
```

Intersection of 2 Arrays

(collection.Counter)

Given arr1 = [1, 2, 2, 1], arr2 = [2, 2], return [2, 2]. Set does not work here, but counter has set-like properties.

```
def intersect(self, nums1, nums2):
    counter1 = collections.Counter(nums1)
    counter2 = collections.Counter(nums2)
    return list((counter1 & counter2).elements())
```

Two Sum

Given nums = [2, 7, 11, 15], target = 9, because nums[0] + nums[1] = 2 + 7 = 9, return [0, 1].

Idea: If not seen, store in dict with key as remainder and value as current index

```
def twoSum(self, nums, target):
    dict = {}
    for i, elem in enumerate(nums):
        try:
            return [dict[elem], i]
        except KeyError:
            dict[target - elem] = i
```

3 Sum

Like 2 Sum, but given an array, find all unique combinations of 3 elements that sums to zero.

E.g. Given [-1, 0, 1, 2, -1, -4], return [[-1, -1, 2], [-1, 0, 1]].

Idea: Sort array first. Skip duplicates. For each element, we have 2 pointers in the subarray after (one immediately behind, one from the back. Move two pointers toward each other.

If total sum is less than zero, then we need to increase → move left pointer towards the right (increasing). Similarly, if total sum is more than zero,m then move right pointer towards left to decrease. Skip if element is similar.

```
def threeSum(self, nums):
    nums = sorted(nums)
    toReturn = []

    for i in range(len(nums)-2):
        if i > 0 and nums[i] == nums[i-1]:
            continue

        front, back = i+1, len(nums) - 1
        while front < back:
            total = nums[i] + nums[front] + nums[back]
            if total == 0:
                toReturn.append([nums[i], nums[front], nums[back]])
                while front < back and nums[front] == nums[front+1]:
                    front += 1
                while front < back and nums[back] == nums[back-1]:
                    back -= 1
                front += 1
                back -= 1
            elif total > 0:
                back -= 1
            elif total < 0:
                front += 1
    return toReturn
```

Find Duplicates in array

Set operation can take in an array directly!!

```
def containsDuplicate(arr):
    # Naive solution 1: create set, takes up extra space
    # uniqueSet = set()
    # for i in arr:
    #     if i in uniqueSet:
    #         uniqueSet.remove(i)
    #     else:
    #         uniqueSet.add(i)
    # return len(uniqueSet) != len(arr)

    return len(set(arr)) != len(arr)
```

Time: O(n), space: O(n)

Other ideas: use XOR, modify index (if arr = [1, 2, 4, 3, 3 containing index)

```
def containsDuplicate(arr):
    for i in range(0, size):
        if arr[abs(arr[i])] >= 0:
            arr[abs(arr[i])] = -arr[abs(arr[i])]
        else:
            print (abs(arr[i]))
```

Time: O(n), space: O(1)

Remove Duplicates in Array

Removes duplicates in place, returns number of unique elems

```
def removeDuplicates(self, nums):

    if len(nums) < 2:
        return len(nums)

    numUnique = 0
    while True:
        if nums[numUnique] == nums[numUnique+1]:
            nums.pop(numUnique)
        else:
            numUnique += 1

        if numUnique == len(nums) - 1:
            break

    return (numUnique + 1)
```

Majority Element

Get element which appears more than half in array

Idea 1: Sorting, get middle value O(n lg n)

```
def getMajority(self, nums):
      return sorted(nums)[len(nums)/2]
```

Idea 2: **Boyer-Moore Voting Algorithm.** One pass O(n), we know it's more than half. So use a counter, add when element is encountered, and subtract if it's not element; substitute if count is 0.

```
def getMajority(self, nums):
      count, most = 0, None
      for i in nums:
            # update most if count is reset to 0
            if count == 0:
                  count = 1
                  most = i
            elif i == most:
                  count += 1
            else:
                  count -= 1
      return most
```

See H: Other Useful Algorithms.

Idea 3: Randomization O(inf) time, O(1) space
Because more than n/2 elements in the array are the same, random index is likely to contain the majority element. Possibly we could never select the element randomly, and it's always more than half: (iter s_{mod} is when majority element is exactly half of len(arr))

$$
\begin{aligned}
EV(iters_{prob}) &\leq EV(iters_{mod}) \\
&= \lim_{n\to\infty} \sum_{i=1}^{n} i \cdot \frac{1}{2^i} \\
&= 2
\end{aligned}
$$

The series converges, the expected number of iterations for the modified problem is constant. The expected runtime is linear for the modifed problem and our problem as the upper bound.

```
    def majorityElement(self, nums):
        import random
        while True:
            r = random.choice(nums)
            if nums.count(r) > len(nums)/2:
                return r
```

House Robber: Maximize Subarray Sum without Neighbors

Find maximum sum of $ robbed from elements in array without adjacent elements robbed together.

Input: [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1). Total amount you can rob = 2 + 9 + 1 = 12.

Idea 1: Recursion with memoization. Either take, or dont (like backpack/stairs qn) but loop through element in array, skipping next one if we sum curr one.

```
    def rob(self, nums, housesDict = {0:0}):
        if not nums:
            return 0
        if len(nums) in housesDict.keys():
            return housesDict[len(nums)]
        globalMax = 0
        for i, elem in enumerate(nums):
            currMax = max(elem + self.rob(nums[i+2:]),
                                    self.rob(nums[i+1:]))
            if currMax > globalMax:
                globalMax = currMax
            housesDict[len(nums) - i] = currMax
        return globalMax
```

Idea 2: **Elegant.**

Based of the recursive relation:

f(0) = nums[0]
f(1) = max(nums[1], nums[0])
f(n) = max(nums[n] + f(n-2), nums[n-1])

```
def rob(self, nums):

    prev, curr = 0, 0
    for i in nums:
        prev, curr = curr, max(prev + i, curr)
    return curr
```

Shortest Subarray to be Sorted (Contiguous)

Find the length of the shortest contiguous subarray needed to be sorted. for the whole array to be sorted.

Input: [2, 6, 4, 8, 10, 9, 15]
Output: 5
Explanation: You need to sort [6, 4, 8, 10, 9] in ascending order to make the whole array sorted in ascending order.

Idea 1: Sorting Array, O(n lg n)

```
    def findUnsortedSubarray(self, nums):

        sortedNums = sorted(nums)
        start, end = 0, len(nums)-1
        while start <= end and nums[start] == sortedNums[start]:
            start += 1
        while end >= start and nums[end] == sortedNums[end]:
            end -= 1
        return end - start + 1
```

Monotonic Stack (Idea 2)

We use a stack that keeps track of indices, and ensures that according to indices of the stack, it gives elements of nums in increasing order. Find position of maximum and minimum element in unsorted array as boundaries.

To find the minumum: we start from left of nums. We we face increasing elements, we push the indices on that stack. Once we see an element that is smaller than the last element in the stack, we know it is out of order. To find its correct position, keep popping from stack until the out-of-order element is larger than the last element on the stack.

We then repeat the same thing from the right to find the max index.

```
def findUnsortedSubarray(self, nums):

    minIndex, maxIndex = len(nums) - 1, 0
    stack = []

    for i in range(len(nums)):
        while (len(stack) != 0 and nums[stack[-1]] > nums[i]):
            minIndex = min(stack.pop(), minIndex)
        stack.append(i)

    stack = []

    for i in range(len(nums)-1, -1, -1):
        while (len(stack) != 0 and nums[stack[-1]] < nums[i]):
            maxIndex = max(stack.pop(), maxIndex)
        stack.append(i)

    if maxIndex > minIndex:
        return maxIndex - minIndex + 1
    else:
        return 0
```

Idea 3 TODO

Move Zeros to End of List

Modify array in place, minimize operations.
Input: [0, 1, 0, 2, 12]
Output: [1, 3, 12, 0, 0]

Idea:
Have a last Zero Index pointer. Loop through array; at the same time keep track of last position of zero. If curr is non zero, then swap (even swap itself), and advance last zero pointer. Otherwise, we keep lastZeroIndex. Move last zero to swap with element that should be in correct position.

```
def moveZeroes(self, nums):
    lastZeroIndex = 0
    for i in range(len(nums)):
        print ((lastZeroIndex, i, nums))
        if nums[i] != 0:
            nums[i], nums[lastZeroIndex] = nums[lastZeroIndex], nums[i]
            lastZeroIndex += 1
```

Time: O(n). Space: O(1).

Contiguous Subarray sums to k

Return indices where contiguous array sums to k. Returns first match.
If not found, return [-1, -1]

Input: arr[] = {1, 4, 20, 3, 10, 5}, sum = 33
Ouptut: Sum found between indexes 2 and 4

More efficient than brute force with O(n^2);
Time: O(n). At most 2 operations are performed on every element in l in the worst case, where the element is added and subtracted. O(2n) is the upper bound; hence time complexity is O(n).
Space: O(1). No data structures used.

```
def answer(l, t):
    # initializes with the first element
    startIndex, runningSum = 0, l[0]
    endIndex = 1

    while endIndex < len(l)+1:
        # if running sum is greater than t, we remove trailing elems
        while runningSum > t and startIndex < endIndex-1:
            runningSum = runningSum - l[startIndex]
            startIndex += 1

        # if running sum of window is t, we have our answer
        if runningSum == t:
            return [startIndex, endIndex - 1]

        # adds current element to window
        if endIndex < len(l):
            runningSum += l[endIndex]

        endIndex += 1

    # if no answer is found, return [-1, -1]
    return [-1, -1]
```

Increasing Triplet Subsequence

Returns whether an array contains an increasing subsequence of length 3.
[5, 5, 5, 1, 5, 2, 5, 5, 4] returns true (1 → 2 → 4)

```
def increasingTriplet(self, nums):

    smallest, middle = float("inf"), float("inf")

    for i in nums:
        # Case 1: i is the smallest element.
        if i <= smallest:
            smallest = i
        # Case 2: here when i is greater than smallest,
        # it can either be second or largest
        elif i <= middle:
            middle = i
        # Here we have smallest < middle and middle < i
        # So subsequence of 3 exist
        else:
            return True

    return False
```

Time: O(n), space: O(1)

Maximum Subarray

Best Time to Buy and Sell Stock

Check Kadane’s Algorithm below. **TO BUY OR NOT TO BUY:**

```
currMax = max(profit + currMax, currMax)
globalMax = max(globalMax, currMax)
```

Search Insert Position

Check Binsert in Algorithms below.

Find Closest Number in Array

Idea 1: Using Pythonic way, O(n) by finding min in array

```
min(myList, key = lambda x:abs(x - myNumber))
```

Idea 2: Using binsert, built in

```
from bisect import bisect_left

def closest (myList, myNumber):
      pos = bisect_left(myList, myNumber)
      if pos == 0:
            return myList[0]
      elif pos == len(myNumber):
            return myList[-1]
      front, back = myList[pos - 1], myList[pos]
      if abs(front - myNumber) < abs(back - myNumber):
            return front
      else:
            return back
```

B. Strings

Ask:

- Input character set and case sensitivity. (lowercase?)
- Empty string
- Single-character string
- Strings with only one distinct character

Compare string when order isn't important (e.g. anagram): use HashMap/counter in Py

Space complexity for counter: **NOT O(n) BUT O(1). Upper bound is no. of characters (26).**

Common string algorithms are

- **Rabin Karp**, which conducts efficient searches of substrings, using a rolling hash
- **KMP,** which conducts efficient searches of substrings

Non-repeating characters: Use a 26-bit bitmask to indicate which lower case Latin characters are inside the string.

```
mask = 0
for c in set(word):
      mask |= (1 << ord(c) - ord('a')))
```

To determine if two strings have common characters, perform & on the two bitmasks. If the result is non-zero, mask_a & mask_b > 0 , then the two strings have common characters.

Anagram (E.g. listen → silent)

Rearranging the letters of a word or phrase to produce a new word/phrase, while using all the original letters only once. To determine if two strings are anagrams, there are a few plausible approaches:

- **Sorting both strings** should produce the **same resulting string**. This takes **O(nlgn) time and O(lgn) space**.
- If we map **each character to a prime number** and we **multiply** each mapped number together, anagrams should have the **same multiple** (prime factor decomposition). This takes **O(n) time and O(1) space**.
- **Frequency counting (dict/set)** of characters will help to determine if two strings are anagrams. This also takes **O(n) time and O(1) space**.

Palindrome (madam, racecar)

Here are ways to determine if a string is a palindrome:

- **Reverse** the string and it should be **equal to itself**.
- Have **two pointers at the start and end** of the string. **Move the pointers inward (recursive/iterative)** till they meet. At any point in time, the characters at both pointers should match.
- ~~Hashmaps~~ (order matters)
- **Insert # between characters** to address even/odd issue **(Manacher's Algorithm)**

Can be even or odd length. For each middle pivot position, you need to check it twice: Once that includes the character and once without the character.

Subsequences → Dynamic Programming

Longest Common Prefix in Str Array

[Flower, flow, flight]: return 'fl'.

Ideas: Vertical scanning, Horizontal scanning, Divide and conquer, Binsearch, Trie

```
    def longestCommonPrefix(self, strs):

        if len(strs) == 0:
            return ''
        prefix = strs[0]
        for i in range(1, len(strs)):
            while strs[i].find(prefix) != 0:
                prefix = prefix[:-1]
                if prefix == '':
                    return prefix
        return prefix

      # Idea 2
        if not strs:
            return ""
        shortest = min(strs,key=len)
        for i, ch in enumerate(shortest):
            for other in strs:
                if other[i] != ch:
                    return shortest[:i]
        return shortest
```

Is Palindrome

Contains punctuation. Use regex, extract alphanumeric. Or just return s == s[::-1] "race, car!" is a palindrome.

```
import re
def isPalindrome(self, s):
    s = re.sub('[^a-zA-Z0-9]', "", s)
    s = s.lower()
    # return s == s[::-1]
    for i in range(len(s)//2):
        if s[i] != s[len(s)-i-1]:
            return False
    return True
```

Is Anagram

Can't simply compare sets. Check for cases like "acc" and "aac". Anagrams need to keep track of counts.   "anagram" is an anagram for "nagaram".

```
def isAnagram(self, s, t):
    if not s or not t:
        return s == t
    dict1, dict2 = {}, {}
    for i, letter in enumerate(s):
        dict1[letter] = dict1.get(letter, 0) + 1
    for i, letter in enumerate(t):
        dict2[letter] = dict2.get(letter, 0) + 1
    return dict1 == dict2
```

O(1) space because dictionary has maximum of 26 entries. Letters, at least.

Smarter way: one increments, the other string decrements.

```
def isAnagram(self, s, t):
    arr = []
    try:
        for letter in s:
            arr.append(letter)
        for letter in t:
            arr.remove(letter)
        return len(arr) == 0
    except:
        return False
```

Group Anagrams

Given a list of strings, group anagrams together.

Input: ["eat", "tea", "tan", "ate", "nat", "bat"],
Output:

[["ate","eat","tea"],

["nat","tan"],

["bat"]]

```
def groupAnagrams(self, strs):
    toReturn, sortedDict = [], {}
    for word in strs:
        key = ''.join(sorted(word))
        if key in sortedDict:
            toReturn[sortedDict[key]].append(word)
        else:
            toReturn.append([word])
            sortedDict[key] = len(toReturn) - 1
    return toReturn


Idea 2:

def groupAnagrams(strs):
    d = {}
    for w in sorted(strs):
        key = tuple(sorted(w))
        d[key] = d.get(key, []) + [w]
    return d.values()
```

First Unique Character in String

Returns the index of the first unique character in string, otherwise return -1.

Idea: String methods: rfind, find, count, index

```
def firstUniqChar(self, s):
    letters = "abcdefghijklmnopqrstuvwxyz"
    uniqueChars = [s.index(c) for c in letters if s.count(c) == 1]
    if len(uniqueChars) > 0:
        return min(uniqueChars)
    return -1


# Idea 2: Using find, rfind. If rfind and find index are the same, then it
is unique
def firstUniqChar(self, s):
    letters = "abcdefghijklmnopqrstuvwxyz"
    minLength = len(s)
    for c in letters:
        leftIndex, rightIndex = s.find(c), s.rfind(c)
        if leftIndex == rightIndex and leftIndex != -1:
            minLength = min(minLength, leftIndex)
    if minLength < len(s):
        return minLength
    return -1
```

Longest Substr without Repeating Characters

Idea: **Keep track in set, move two pointers as windows (returns number with global max) OR one-pass windowing substring**

```
def lengthOfLongestSubstring(self, s):
    i, j, set1 = 0, 0, set()
    maxNum = 0
    while j < len(s):
        if s[j] not in set1:
            set1.add(s[j])
            maxNum = max(maxNum, len(set1))
            j += 1
        else:
            set1.remove(s[i])
            i += 1
    return maxNum


OR

    def lengthOfLongestSubstring(self, s):
        substr = maxStr = ''
        for char in s:
            if char not in substr:
                substr = substr + char
            else:
                # slide window forward up to the end of substring
                substr = substr[substr.index(char) + 1:] + char
            if len(substr) > len(maxStr):
                maxStr = substr
        return len(maxStr)
```

Time: O(n^2) because string index method is O(n), space: O(1)

Best Solution: O(n) time. O(1) Space: Constant space dictionary.

One sweep from left to right, with dict keeping track of index.
If seen: move start to the right
If not seen: update max length
Then add index to dict

```
def lengthOfLongestSubstring(self, s):

    maxLen, start = 0, 0
    usedChar = {}

    for i, char in enumerate(s):
        if char in usedChar and usedChar[char] >= start:
            start = usedChar[char] + 1
        else:
            maxLen = max(maxLen, i - start + 1)
        usedChar[char] = i

    return maxLen
```

After we do start = usedChar[char] + 1, we only want to consider windows after start. There could be some characters before start in dictionary, and we don't wanna consider that. We only consider substrings from **start** to **i** each iter.

Insert Spaces to Separate Words

Idea: DFS, with memoization

```
import copy

def wordBreak(s, wordDict, word = "", lst = []):
    if s == "":
        if word in wordDict:
            lst.append(word)
            print(lst)
        return
    word += s[0]
    if word in wordDict:
        newList = copy.deepcopy(lst)
        newList.append(word)
        wordBreak(s[1:], wordDict, newList, "")
    wordBreak(s[1:], wordDict, lst, word)
```

Drawback: No memoization

Solution 2 with memoization:

```
def wordBreak(s, wordDict, memo = {}):
    if not s:
        return []
    if s in memo:
        return memo[s]
    res = []
    for word in wordDict:
        if not s.startswith(word):
            continue
        if len(s) == len(word):
            return [word]
        else:
            resultOfRest = wordBreak(s[len(word):], wordDict, memo)
            for item in resultOfRest:
                item = word + ' ' + item
                res.append(item)
    memo[s] = res
    return res
```

Can reduce need for helper function by Python declaration of param

Longest Palindromic Substring

Note odd and even lengths!

```
def longestPalindrome(self, s):
    longest = ""
    for i in range(len(s)):

        # Case 1: odd length of palindrome
        temp = s[i]
        front, back = i-1, i+1
        while (front >= 0 and back < len(s) and s[front] == s[back]):
            temp = s[front] + temp + s[back]
            front, back = front - 1, back + 1
        if len(temp) > len(longest):
            longest = temp

        # Case 2: even length of palindrome
        temp2 = ""
        front, back = i, i+1
        while (front >= 0 and back < len(s) and s[front] == s[back]):
            temp2 = s[front] + temp2 + s[back]
            front, back = front - 1, back + 1
        if len(temp2) > len(longest):
            longest = temp2

    return longest
```

Time: O(n^2), Space: O(1)

Check out **Manacher's Algorithm** that inserts # between characters! **O(n) time, O(n) space.**

Word Ladder

Idea: BFS, create visited set, queue, put tuple (new word and length) in queue

```
def ladderLength(self, beginWord, endWord, wordList):
    visited = set(wordList)
    wordQueue = deque()
    chars = 'abcdefghijklmnopqrstuvwxyz'
    wordLength = len(beginWord)

    wordQueue.append([beginWord, 1])
    while wordQueue:
        curr, currLength = wordQueue.popleft()
        if curr == endWord:
            return currLength
        for i in range(wordLength):
            for c in chars:
                newWord = curr[:i] + c + curr[i+1:]
                if newWord in visited:
                    visited.remove(newWord)
                    wordQueue.append([newWord, currLength + 1])
    return 0
```

Prevent TLE: instead of created visited set, and check twice (if visited and if in wordDict), make a set from wordDict and check once (remove if visited)

Time: O(n*d), space: O(d) where d is dictionary length

Find all Anagrams in String given Pattern

Sliding window. Time: O(len(s)). Maintain window of len(p) in s. Add, check, remove. Comparison of counter is O(1), at most 26 characters. Extra space is constant: at most 26 characters.

1. Set counter of pattern. Set counter of window.
2. In for loop:
   a. add new char to window counter
   b. check
   c. remove starting char
   d. **Delete if 0 entry.**

```python
def findAnagrams(self, s, p):
    from collections import Counter

    pattern = Counter(p)
    window = Counter(s[:len(p) - 1])
    toReturn = []

    for i in range(len(p) - 1, len(s)):

        left_index, right_index = i - len(p) + 1, i
        left_char, right_char = s[left_index], s[right_index]

        window[right_char] += 1

        if window == pattern:
            toReturn.append(left_index)

        window[left_char] -= 1

        if window[left_char] == 0:
            del window[left_char]

    return toReturn
```

Minimum Window Substring

Sliding Window Algorithm. Window of variable size (begin, end markers)

```
# initialize frequency table
for char c in T do
    table[c]++;
end

counter = table.size()                 # unique chars in T

for char c in string B do
    if(char c in table){
        table[c]--;                    # decrement count for c
        if(table[c] == 0) counter--;
    }
end

if(counter == 0){ # B has every character in T }
```

If table goes negative, (e.g. T has 4 K's but S has 7 K's, thus satisfying requirement for K's)

Algorithm:

1. Initialize frequency table as window, with counter as unique letters in pattern
2. start sliding window, if char in table then decrement count
3. If count == 0 we found an answer, now slide start of window to the right to trim length
    a. store answer if best
    b. If begin char is not in table, we can shorten
    c. Otherwise we increase count
    d. If count goes above zero means that the current window is missing one char to be an answer candidate

S = ADOBECODEBANC

T = ABC

frequency table for T → table

A	1
B	1
C	1

sliding window: end = 0, begin = 0

counter = table.size()

= 3

1]

```
begin →
↓
ADOBECODEBANC
↑
end →
```

A	0
B	1
C	1

counter 2

2]

```
begin
↓
ADOBECODEBANC
 ↑
 end
```

A	0
B	1
C	1

counter 2

3]

```
begin
↓
ADOBECODEBANC
  ↑
  end
```

A	0
B	ϕ
C	1

counter 2

4]

```
begin
↓
ADOBECODEBANC
   ↑
   end
```

A	0
B	0
C	1

counter 1

⋮

5] ...

⋮

]

```
beg
↓
ADOBECODEBANC
     ↑
     end
```

A	0
B	0
C	0

$\boxed{\text{counter } 0}$

if counter == 0 | valid answer |

try minimizing the answer

slide begin and see if some non-essential characters can be dropped.

```
        begin
          ↓
7]    A D O B E C O D E B A N C
                ↑
               end
```

$\begin{Bmatrix} A & 1 \\ B & 0 \\ C & 0 \end{Bmatrix}$ counter 1

```
         ↓beg
8]    A D O B E C O D E B A N C
                  ↑
                 end
```

$\begin{Bmatrix} A & 1 \\ B & 0 \\ C & 0 \end{Bmatrix}$ counter 1

⋮

```
        beg
         ↓
12]   A D O B E C O D E B A N C
                           ↗
                        end
```

$\begin{Bmatrix} A & 0 \\ B & 0 \\ C & 0 \end{Bmatrix}$ $\boxed{\text{counter } 0}$

try minimizing answer by sliding begin.

```
          beg
           ↓
13]   A D O B E C O D E B A N C
                           ↗
                        end
```

$\boxed{\text{counter} = 0}$

```
            beg
             ↓
14]   A D O B E C O D E B A N C
                           ↗
                        end
```

$\boxed{\text{counter} = 0}$

```
def minWindow(self, s, t):

    # We need frequency table and counter for number of letters
    from collections import Counter
    table = Counter(t)
    begin, counter, toReturn = 0, len(table), ""
    answerLen = float("inf")

    # slide right edge of window
    for end, endChar in enumerate(s):
        if endChar in table:
            table[endChar] -= 1
            if table[endChar] == 0:
                counter -= 1

        # slide left edge of window
        while counter == 0:
            if end - begin + 1 < answerLen:
                answerLen = end - begin + 1
                toReturn = s[begin:end + 1]

            startChar = s[begin]
            if startChar in table:
                table[startChar] += 1
                if table[startChar] > 0:
                    counter += 1

            begin += 1

    return toReturn
```

C. Linked List

Print/remove Nth node from Linked List

Idea: Advance front pointer forward N times, them advance both till end.

```
def removeNthFromEnd(self, head, n):
    fast = slow = head
    for _ in range(n):
        fast = fast.next
    if not fast:
        return head.next
    while fast.next:
        fast = fast.next
        slow = slow.next
    slow.next = slow.next.next       #remove Nth node
    return head
```

Time: O(n), space: O(1)

Print middle of linked-list

Idea: Advance fast twice the speed of slow. Check for fast and fast.next.

```
def middleNode(self, head):
    slow, fast = head, head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

Time: O(n), space: O(1)

Traverse linked list with 2 pointers to find middle

Cycles in linked list

Solution 1: General idea of two pointers, one twice as fast as the other, check if they are equal. Condition: **while fast and fast.next**

```
def hasCycle(head):
    """
    :type head: ListNode
    :rtype: bool
    """
    # edge case:
    if !head.next:
        return False


    slow, fast = head, head.next
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
        if slow == fast:
            return True
    return False
```

Better solution 2:

Easier to ask for forgiveness than permission. Use try/catch instead of checking for if statements. (THIS IS SO ELEGANT AND FAST. TAKE NOTE)

```
def hasCycle(head):
    try:
        slow, fast = head, head.next
        while slow is not fast:
            slow, fast = slow.next, fast.next.next
        return True
    except:
        return False
```

Lesson: Easier to ask for forgiveness than permission (try/catch) > Look before you leap (if statements). Especially common in dict.

```
# Inferior LBYL:

if name in names:
    print (names[name])
else:
    print (name + " doesn't exist")

OR

real_name = names.get(name, None)
if real_name:
    print (real_name)
else:
    print (name + " doesn't exist")

# EAFP:

try:
    print (names[name])
except KeyError:
    print (name + " doesn't exist")
```

Reverse Linked List

Idea: 3 pointers: head, prev, cur. Don’t move prev. Swap head and curr and adjust nexts.

```
# Iterative Solution
def reverseList(self, head):
    if not head:
        return head
    curr = head.next
    prev = head
    while curr:
        head, curr.next, prev.next = curr, head, curr.next
        curr = prev.next

    return head

Easier still:
# move current all the way till the end. Set prev to be the end(None) first
# Return prev, when curr is at the end at None
def reverseList(self, head):
    curr, prev = head, None
    while curr:
        curr.next, prev, curr = prev, curr, curr.next
    return prev
```

Pairwise swap elements of singly linked list

```
def swapPairs(self, head):
    dummy = ListNode(None)
    dummy.next = head
    toReturn = dummy
    while dummy.next and dummy.next.next:
        before, after = dummy.next, dummy.next.next
        before.next, after.next = after.next, before
        dummy.next = after
        dummy = before
    return toReturn.next
```

Time: O(n), Space: O(1)

Todo: merge 2/k sorted list, reverse LL

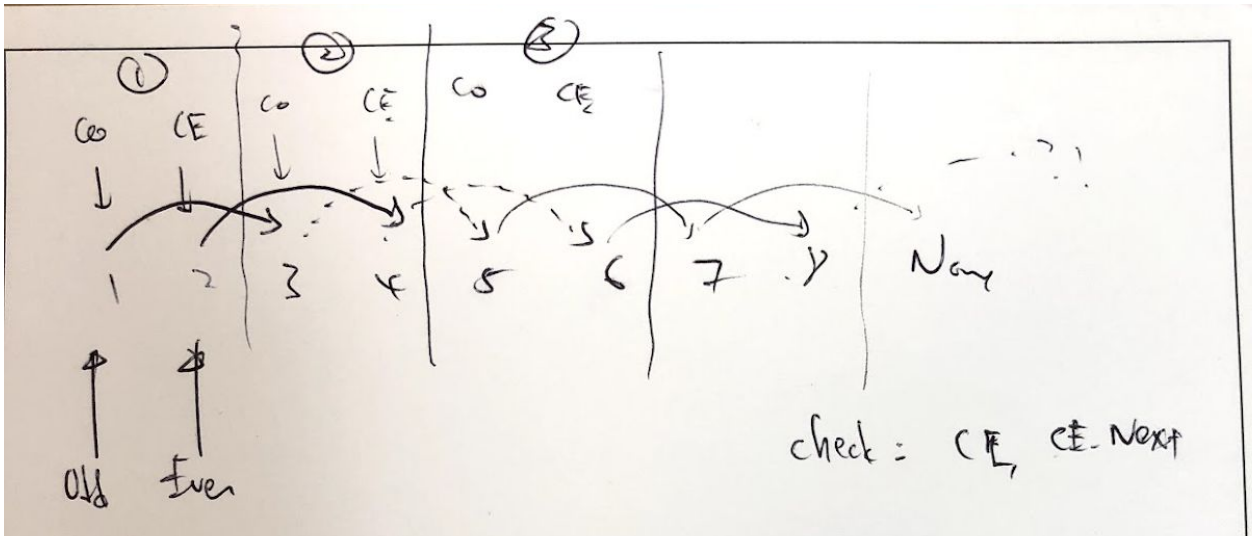
Add Two numbers with linked list

```
    def addTwoNumbers(self, l1, l2):
        dummy = ListNode(0)
        carry = 0
        head = dummy
        while l1 or l2:
            if l1 and l2:
                total = (l1.val + l2.val + carry)
                l1, l2 = l1.next, l2.next
            elif l1:
                total = l1.val + carry
                l1 = l1.next
            else:
                total = l2.val + carry
                l2 = l2.next
            head.next = ListNode(total%10)
            head = head.next
            carry = total//10
        if carry:
            head.next = ListNode(carry)
        return dummy.next
```

Group Odd, Even Linked List

Group odd nodes first, then even.
Input: 1 → 2 → 3 → 4 → 5 → None
Output: 1 → 3 → 5 → 2 → 4 → None

Idea: In place, O(1) space, O(n) time. Loop through once, split into even and odd.



```
def oddEvenList(self, head):

    if head is None:
        return None

    odd, even = head, head.next
    currOdd, currEven = odd, even

    while currEven and currEven.next:
        currOdd.next = currEven.next
        currOdd = currOdd.next
        currEven.next = currOdd.next
        currEven = currEven.next

    currOdd.next = even

    return odd
```

Merge Two Sorted List

First thought: iteratively

```
    def mergeTwoLists(self, l1, l2):
        curr = dummy = ListNode(None)
        while l1 and l2:
            if l1.val > l2.val:
                curr.next = l2
                l2 = l2.next
            else:
                curr.next = l1
                l1 = l1.next
            curr = curr.next
        curr.next = l1 or l2      (this is smart, instead of if l1 else l2)

        return dummy.next
```

Second thought: Recursively

```
    def mergeTwoLists(self, l1, l2):
        if l1 and l2:
            if l1.val < l2.val:
                l1.next = self.mergeTwoLists(l1.next, l2)
                return l1
            else:
                l2.next = self.mergeTwoLists(l1, l2.next)
                return l2
        else:
            return l1 or l2
```

Palindrome Linked List

Good practice! First 2 steps build on previous questions

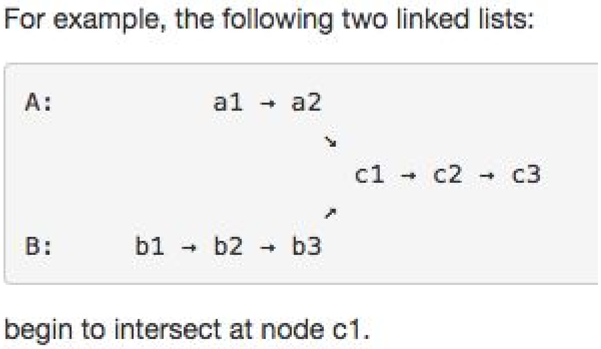
3 Steps to do in O(n) time, O(1) space:

1. Find middle of linked list
2. Reverse the second half
3. Compare values along both halves

```
def isPalindrome(self, head):
    slow, fast = head, head

    # find middle of list
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # reverse second half
    prev, curr = None, slow
    while curr:
        curr.next, prev, curr = prev, curr, curr.next

    # compare the two halves
    while prev:
        if prev.val != head.val:
            return False
        prev, head = prev.next, head.next

    return True
```

Find Intersection of Linked Lists

Using O(n) time and O(1) space, find intersection of singly linked list.



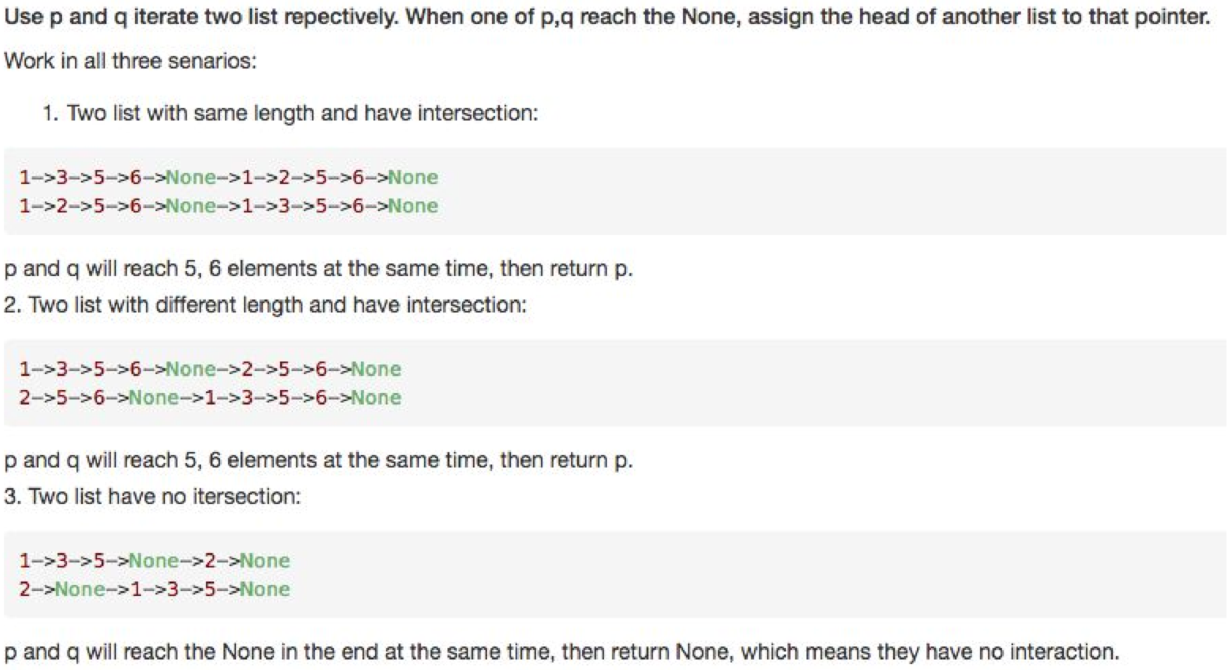
For example, the following two linked lists:

```
A:          a1 → a2
                   ↘
                     c1 → c2 → c3
                   ↗
B:     b1 → b2 → b3
```

begin to intersect at node c1.

IDEA:

Len (A) = 2, len (B) = 10, len (A+B) = len(B+A) = 12



Use p and q iterate two list repectively. When one of p,q reach the None, assign the head of another list to that pointer.

Work in all three senarios:

1. Two list with same length and have intersection:

```
1->3->5->6->None->1->2->5->6->None
1->2->5->6->None->1->3->5->6->None
```

p and q will reach 5, 6 elements at the same time, then return p.

2. Two list with different length and have intersection:

```
1->3->5->6->None->2->5->6->None
2->5->6->None->1->3->5->6->None
```

p and q will reach 5, 6 elements at the same time, then return p.

3. Two list have no itersection:

```
1->3->5->None->2->None
2->None->1->3->5->None
```

p and q will reach the None in the end at the same time, then return None, which means they have no interaction.

```
    def getIntersectionNode(self, headA, headB):

        currA, currB = headA, headB
        if not currA or not currB:
            return None

        while currA != currB:
            currA = headB if not currA else currA.next
            currB = startA if not currB else currB.next
        return currA
```

D. Binary Search Trees

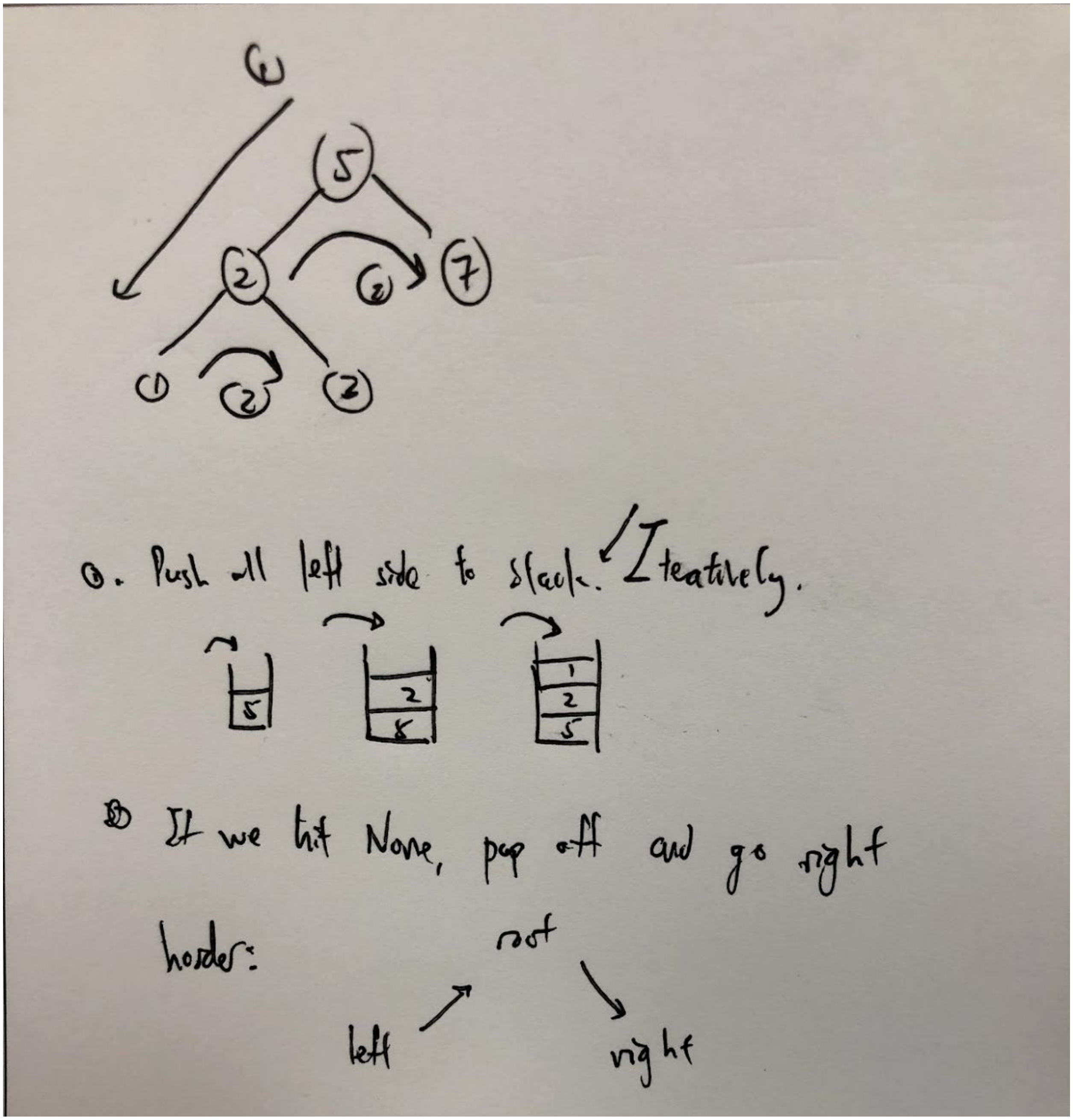
Be familiar with recursive solutions. On top of that learn iterative solution and Morris Traversal.

In-Order Traversal

Recursively:

```
def inorderTraversal(self, root, toReturn = []):

    def dfs(node, arr):
        if not node:
            return
        dfs(node.left, arr)
        arr.append(node.val)
        dfs(node.right, arr)

    toReturn = []
    dfs(root, toReturn)
    return toReturn
```

O(n) time, O(n) space (uses stack in function call)


①
5
2
7
1
2
3
②
① Push all left side to stack. Iteratively.
5
2
5
1
2
5
② If we hit None, pop off and go right
Inorder: root
left
right

Iteratively:

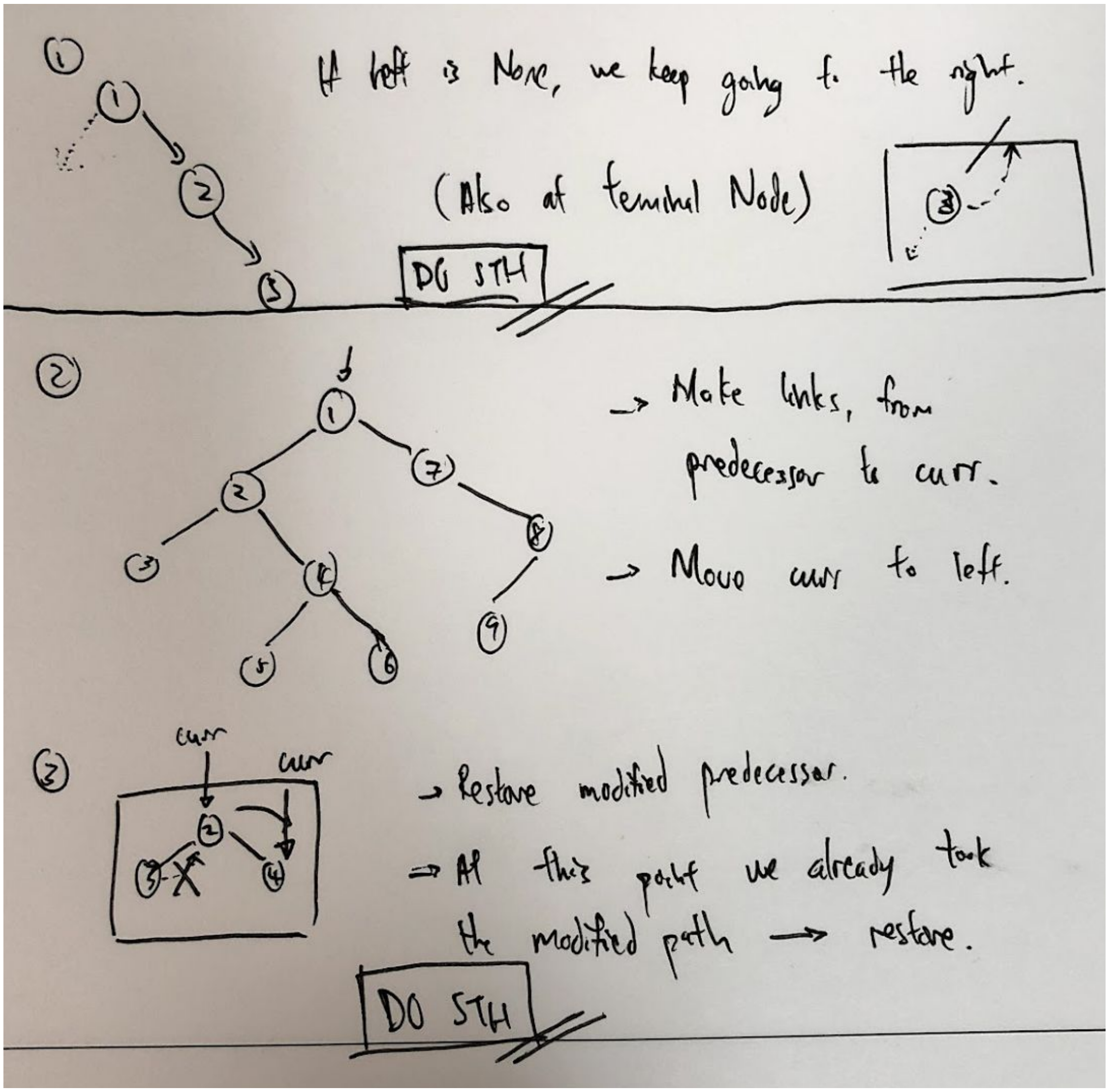
We need 2 things: a stack and an array to append values to.
First, we have a while loop checking for 2 things: currNode and stack.
In the while loop, we keep appending the left child to the stack. ***I***
Then, we pop from the stack, add value to toReturn array, and push right child to stack.

If we have a terminal node (both child are none, we substitute it by popping from the stack).

```
def inorderTraversal(self, root):
    stack, toReturn = [], []
    curr = root
    while curr or len(stack) != 0:
        while curr:
            stack.append(curr)
            curr = curr.left
        curr = stack.pop()
        toReturn.append(curr.val)      # add after all left children
        curr = curr.right
    return toReturn
```

O(n) space, O(n) time.

Idea 3: Morris Tree Traversal.



If left is None, we keep going to the right.

(Also at terminal Node)

DO STH

→ Make links, from predecessor to curr.

→ Move curr to left.

→ Restore modified predecessor.

→ At this point we already took the modified path → restore.

DO STH

Traverse a BST without stacks or recursion. **O(n) time, O(1) space.**

Remove the link made pointing to curr from predecessor. Remove only after moving curr back.

```
def InOrderMorrisTraverse(root):
      # set current to root of BST
      curr = root
      while curr is not None:
            # if nothing is at the left, do sth and move on.
            if curr.left is None:
                  print (curr.val)                    # do sth
                  curr = curr.right
            else:
                  # find the inorder predecessor of current
                  pred = curr.left
                  while (pred.right is not None and pred.right != curr):
                        pred = pred.right

                  # make current the right child of predecessor
                  if pred.right is None:
                        pred.right = curr
                        curr = curr.left
                  # remove the modified BST, fix right child of predecessor
                  else:
                        pred.right = None
                        print (curr.val)
                        curr = curr.right
```

Pre-Order Traversal

Recursively:

```
def preorderTraversal(self, root):

    def dfs(root, array):
        if root:
            array.append(root.val)
            dfs(root.left, array)
            dfs(root.right, array)

    toReturn = []
    dfs(root, toReturn)
    return toReturn
```

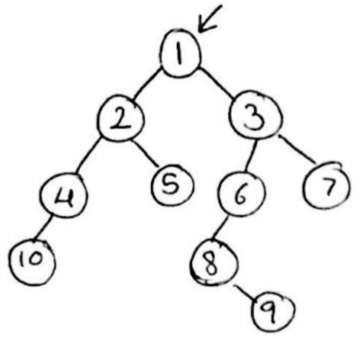
Iteratively:

Same structure as in-order. Just change when to append value to toReturn Array.
Idea: Keep appending left val to stack. *I*
And if it’s null we pop from the stack, and set curr to be curr.right.

```
def preorderTraversal(self, root):

    toReturn, stack = [], []
    curr = root
    while curr or len(stack) != 0:
        while curr:
            toReturn.append(curr.val)
            stack.append(curr)
            curr = curr.left
        curr = stack.pop()
        curr = curr.right

    return toReturn
```

DFS:

The other way: push to stack, pop top, add unvisited neighbors of top, repeat 123 while stack not empty



DFS magic using stack:

initialize stack with node.

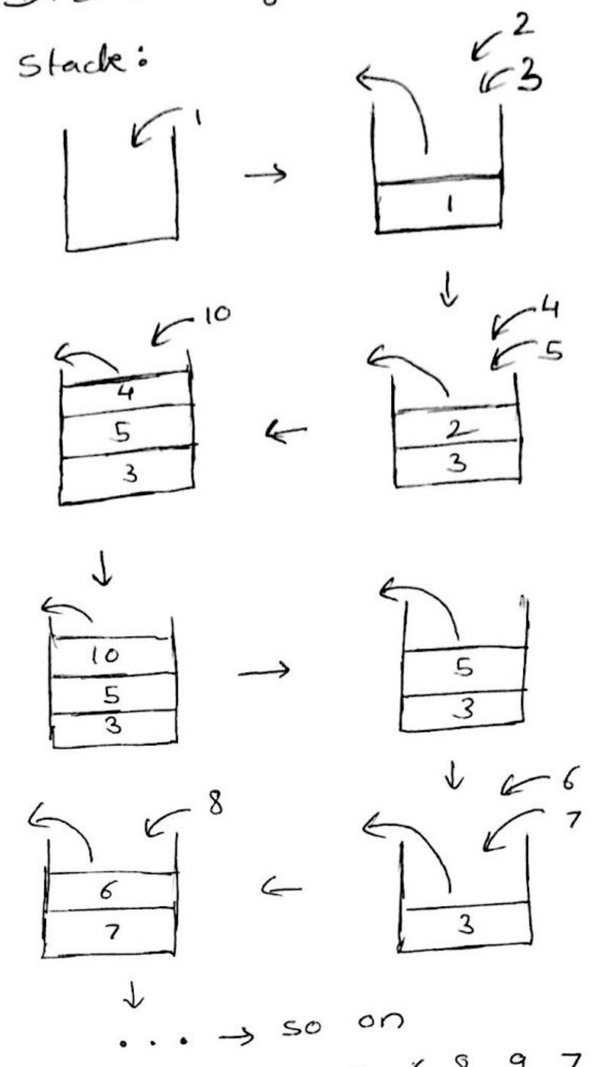
1] pop top, mark visited

2] fetch it's unvisited neighbours

3] push them to stack

4] repeat 1,2,3 while stack is not empty

DFS starting at ①



=> 1 2 4 10 5 3 6 8 9 7

★ See the flow,

<u>2</u> leaves the stack but it's subtrees occupy the stack, only when they are done do we move to <u>3</u>

So left subtree fully visited, then right and root was visited first before these two.

preorder! [root left right]

```
def preorderTraversal(self, root):

    toReturn, stack = [], [root]
    while stack:
        curr = stack.pop()
        if curr:
            toReturn.append(curr.val)
            stack.append(curr.right)
            stack.append(curr.left)
    return toReturn
```

Morris Traversal

Same idea as Morris in-order, except change when to do something to node.

```
def preorderTraversal(self, root):
    toReturn, curr = [], root
    while (curr):
        if curr.left == None:
            toReturn.append(curr.val)
            curr = curr.right
        else:
            predecessor = curr.left
            while (predecessor.right != None and predecessor.right !=
curr):
                predecessor = predecessor.right
            if predecessor.right != curr:
                predecessor.right = curr
                toReturn.append(curr.val)
                curr = curr.left
            else:
                predecessor.right = None
                curr = curr.right

    return toReturn
```

Post-Order Traversal

Recursively

```python
def postorderTraversal(self, root):

    def dfs(node, arr):
        if node:
            dfs(node.left, arr)
            dfs(node.right, arr)
            arr.append(node.val)

    toReturn = []
    dfs(root, toReturn)
    return toReturn
```

Iteratively

Idea: Post-order = Reversed pre-order. To do this we traverse the tree in the mirror image form of pre-order, and reverse the array before we return it.

Mirror pre-order, and reverse.

```python
def postorderTraversal(self, root):
    toReturn, stack = [], []
    curr = root
    while (curr or len(stack) != 0):
        while curr:
            toReturn.append(curr.val)
            stack.append(curr)
            curr = curr.right
        curr = stack.pop()
        curr = curr.left
    return toReturn[::-1]
```

Morris Traversal

SAME IDEA AS POST-ORDER ITERATIVE.
- MIRROR IMAGE OF PRE-ORDER, THEN REVERSE.

```python
def postorderTraversal(self, root):
    toReturn, curr = [], root
    while curr:
        if curr.right is None:
            toReturn.append(curr.val)
            curr = curr.left
        else:
            predecessor = curr.right

            while predecessor.left != None and predecessor.left != curr:
                predecessor = predecessor.left
            if predecessor.left != curr:
                toReturn.append(curr.val)
                predecessor.left = curr
                curr = curr.right
            else:
                predecessor.left = None
                curr = curr.left

    return toReturn[::-1]
```

Level-Order Traversal

Idea 1: Use a queue. In the queue we have a tuple of (node, level). Same idea as BFS.

```
def levelOrder(self, root):
    if not root:
        return []

    queue, toReturn = [(root, 1)], [[]]
    while len(queue) != 0:
        node, level = queue.pop(0)
        if level != len(toReturn):
            toReturn.append([node.val])
        else:
            toReturn[-1].append(node.val)
        if node.left:
            queue.append((node.left, level + 1))
        if node.right:
            queue.append((node.right, level + 1))

    return toReturn
```

Idea 2: Similar approach, instead of using queue we reassign level

```
def levelOrder(self, root):
    if not root:
        return []
    toReturn, level = [], [root]
    while level:
        toReturn.append([n.val for n in level])
        temp = []
        for node in level:
            temp = temp + [node.left, node.right]
        level = [x for x in temp if x is not None]
    return toReturn
```

To flip level order traversal (convert to **zigzag traversal**):

```
        for i in range(1,len(results),2):

            results[i].reverse()
```

Same Tree

```
    def isSameTree(self, p, q):
        if not p and not q:
            return True
        try:
            if p.val == q.val:
                return self.isSameTree(p.left, q.left) and
                           self.isSameTree(p.right, q.right)
            return False
        except:
            return False
```

Idea 2:

```
def sameTree(node1, node2):

    if node1 and node2:
        return node1.val == node2.val and sameTree(node1.left, node2.left)
and sameTree(node1.right, node2.right)

    return node1 is node2
# instead of try/catch, check if p and q, otherwise if p is q
```

Symmetric Tree

Idea: Need 2 trees to compare, which is why we have a helper function

```
    def isSymmetric(self, root):

        def isSymmetricHelper(left, right):
            if not left and not right:
                return True
            try:
                if left.val == right.val:
                    return isSymmetricHelper(left.left, right.right) and
                           isSymmetricHelper(left.right, right.left)
                return False
            except:
                return False

        if not root:
            return True
        return isSymmetricHelper(root.left, root.right)
```

O(n) time, O(n) space. Traverse entire tree and pass through all nodes of tree to check. Worst space complexity: O(n) for skewed tree with height n. → recursive calls on stack.

Iterative BFS Solution Symmetric Tree

```
def isSymmetric(self, root):

    queue = [root, root]
    while (len(queue) != 0):
        node1 = queue.pop(0)
        node2 = queue.pop(0)

        if node1 != None and node2 != None:
            if node1.val != node2.val:
                return False

            queue.append(node1.left)
            queue.append(node2.right)
            queue.append(node1.right)
            queue.append(node2.left)

        else:
            if node1 != node2:
                return False

    return True
```

Valid BST

Key: pass in min and max value. Update minval when checking for left, and update maxVal when checking for larger value (right). Also, it cannot be equal (must be unique).

```
def isValidBST(self, root, minVal = -float("inf"), maxVal = float("inf")):
    if not root:
        return True
    if root.val >= maxVal or root.val <= minVal:
        return False
    return self.isValidBST(root.left, minVal, min(maxVal, root.val))
       and self.isValidBST(root.right, max(minVal, root.val), maxVal)
```

Iterative solution:

Similar idea as iterative in-order. Keep track of previous node (smaller).

```
def isValidBST(self, root):
    prev, stack = None, []
    curr = root

    while (curr or len(stack) != 0):
        while curr:
            stack.append(curr)
            curr = curr.left
        curr = stack.pop()

        if prev != None and curr.val <= prev.val:
            return False
        prev = curr
        curr = curr.right

    return True
```

O(n) time, O(n) space

Maximum Depth of BST

One Liner!

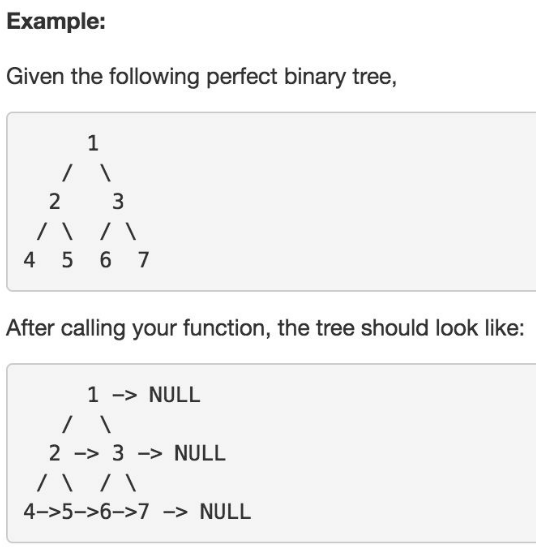
```
def maximumDepth(root):
    return 1 + max(maximumDepth(root.left), maximumDepth(rootright)) if
root else 0
```

Convert Sorted Array to BST

Extension: If array is not sorted, sort it first.

```
def sortedArrayToBST(self, nums):
    if nums == []:
        return None
    midpoint = len(nums)//2
    root = TreeNode(nums[midpoint])
    root.left = self.sortedArrayToBST(nums[:midpoint])
    root.right = self.sortedArrayToBST(nums[midpoint+1:])
    return root
```

Populate Next Right Pointers in Each Node



Example:

Given the following perfect binary tree,

```
     1
   /  \
  2    3
 / \  / \
4  5  6  7
```

After calling your function, the tree should look like:

```
     1 -> NULL
   /  \
  2 -> 3 -> NULL
 / \  / \
4->5->6->7 -> NULL
```

Idea: Almost same as level order traversal. Need to keep track of previous to link.

```
def connect(self, root):
    if not root:
        return None
    prev_node, prev_level, curr = None, 0, root
    queue = [(curr, 1)]
    while (len(queue) != 0):
        curr_node, curr_level = queue.pop(0)
        if prev_node != None:
            if curr_level != prev_level:
                prev_node.next = None
            else:
                prev_node.next = curr_node

        prev_node, prev_level = curr_node, curr_level
        if curr_node.left:
            queue.append((curr_node.left, curr_level + 1))
        if curr_node.right:
            queue.append((curr_node.right, curr_level + 1))
```

Get k-th Smallest Element in BST

Idea: use an array, populate in order, return element at (k-1) index

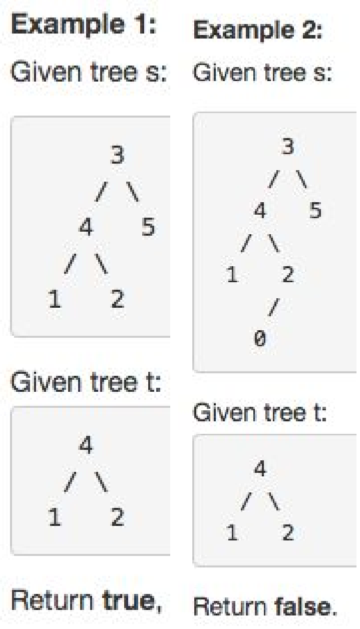
```
def kthSmallest(self, root, k):
    def inOrderTraversal(root, stack):
        if not root:
            return
        inOrderTraversal(root.left, stack)
        stack.append(root.val)
        inOrderTraversal(root.right, stack)


    stack = []
    inOrderTraversal(root, stack)
    return stack[k-1]
```

Iteratively

Or even better, save extra memory: keep track of counter in iterative in-order traversal

```
def kthSmallest(self, root, k):
    number, stack, kth = 1, [], None
    curr = root
    while (curr or len(stack) != 0):
        while curr:
            stack.append(curr)
            curr = curr.left

        curr = stack.pop()
        kth = curr
        if number == k:
            return kth.val
        number += 1
        curr = curr.right

    return None
```

Subtree of Another Tree



Example 1:

Given tree s:

```
     3
    / \
   4   5
  / \
 1   2
```

Given tree t:

```
   4
  / \
 1   2
```

Return **true**,

Example 2:

Given tree s:

```
     3
    / \
   4   5
  / \
 1   2
    /
   0
```

Given tree t:

```
   4
  / \
 1   2
```

Return **false**.

(Cannot Assume BST or ordering of values)

Idea 1: Nodes Comparison. For each node in S, check whether it’s subtree equals t.

```
  def isSubtree(self, s, t):

    def sameTree(s, t):
      if s and t:
        return s.val == t.val and sameTree(s.left, t.left) and
sameTree(s.right, t.right)
      return s is t

    if sameTree(s,t) return True
    if not s: return False
    return isSubTree(s.left, t) or isSubTree(s.right, t)
```

O(s * t), as we check for each node of T in each node of S.

Idea 2: **O(s + t)** with Merkle Hashing (Traverse through each node twice)

Each node is represented with a hash of left subtree’s merkle, node’s value and right subtree’s merkle. Trees are identical if merkle hash of roots are equal.

After finishing hashing T, we then check for each node in S whether node.merkle == T.merkle.

```
def isSubTree(s, t):
   from hashlib import sha256
   def _hash(x):
      S = sha256()
      S.update(x)
      return S.hexdigest()

   def merkle(node):
      if not node:
         return '#'
      m_left = merkle(node.left)
      m_right = merkle(node.right)
      node.merkle = _hash(m_left + str(node.val) + m_right)

   merkle(t)
   merkle(s)
   def dfs(node):
      if not node:
         return None
      return node.merkle == t.merkle
```

```
                  or dfs(node.left) or dfs(node.right)
      return dfs(s)
```

Merge Two Binary Trees

Idea 1: **Recursively** adds: make sure t1 and t2 exist, and add them together, do the same for both children. **If t1 if None then we just return t2, vice versa.**

Time: **O(n).** Need to traverse n nodes, where n is minimum number of 2 trees.
Space: **O(n)**. Depth of recursion goes to n in the case of **skewed tree**. **Average: O(lg n)**.

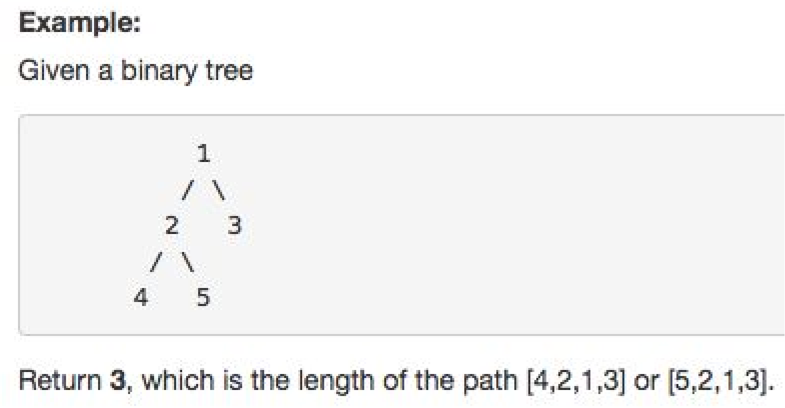
```
    def mergeTrees(self, t1, t2):
        if t1 is None and t2 is None:
            return None
        if t1 is None:
            return t2
        if t2 is None:
            return t1
        node = TreeNode(t1.val + t2.val)
        node.left = self.mergeTrees(t1.left, t2.left)
        node.right = self.mergeTrees(t1.right, t2.right)
        return node
```

Idea 2: Iteratively. Push pair of (Node_{t1}, Node_{t2}) to stack, add them up and assign to t1's val, then push pair of (t1.left, t2.left) to stack, then (t1.right, t2.right) to stack. Keep on popping stack till empty and repeat.

O(n) space (stack of depth n in skewed tree case); O(n) time as we traverse over n nodes.

Diameter (Longest Path) of Binary Tree

Given a binary tree, return the length of the longest path (number of edges) between any two nodes. This path may or may not pass through the root.



Example:

Given a binary tree

```
          1
         / \
        2   3
       / \
      4   5
```

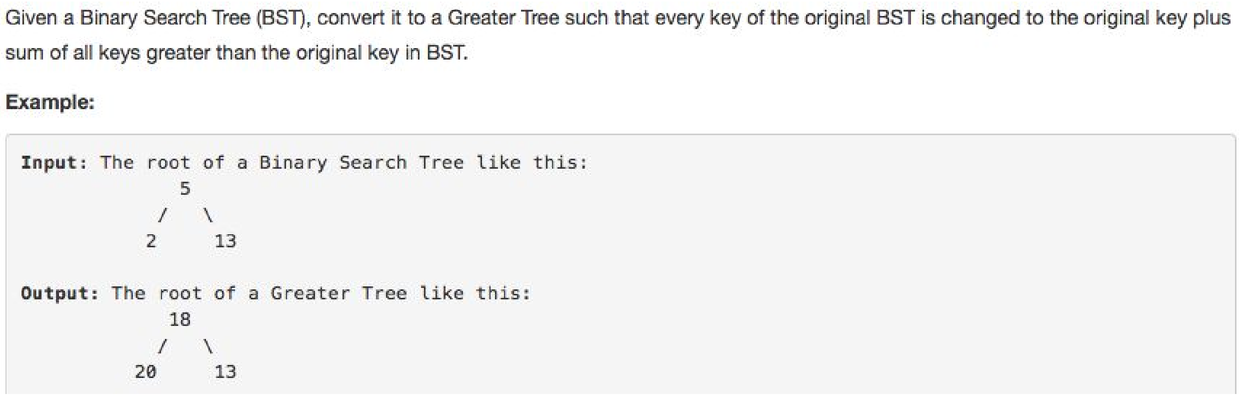
Return **3**, which is the length of the path [4,2,1,3] or [5,2,1,3].

This is like getting depth of binary tree, but comparing between latest answer and Leth Depth + Right Depth + 1. Number of edges = number of nodes - 1. **To simplify things, we calculate number of nodes first, then subtract one to return number of edges.**

```
def diameterOfBinaryTree(self, root):
    # use number of nodes to simplify calculations
    if not root:
        return 0

    self.ans = 1
    def getDepth(node):
        if not node:
            return 0
        leftDepth = getDepth(node.left)
        rightDepth = getDepth(node.right)

        # update self.ans
        self.ans = max(self.ans, 1 + leftDepth + rightDepth)
        return  1 + max(leftDepth, rightDepth)

    getDepth(root)
    return self.ans - 1
```

Time: O(n). Traverse through each node once.
Space: O(n). Depth of recursion goes to O(n) in worst-case of skewed tree, but average O (lg n).

Greater Tree (Reverse In-Order Traversal)



Given a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key plus sum of all keys greater than the original key in BST.

Example:

```
Input: The root of a Binary Search Tree like this:
              5
            /   \
           2     13

Output: The root of a Greater Tree like this:
             18
            /   \
          20     13
```

```python
def convertBST(self, root):

    self.maxSoFar = 0
    def reverse_traversal(node):
        if not node:
            return
        # Traverse node.right
        reverse_traversal(node.right)
        # Do something on curr
        node.val += self.maxSoFar
        self.maxSoFar = node.val
        # Traverse node.left
        reverse_traversal(node.left)

    reverse_traversal(root)

    return root
```

Time: O(n). Each node gets called once.
Space: O(n) for recursion call stack, which is called a linear number of times.

In-Order Traversal with Stack (Reverse)

We first initialize with empty stack and set current to root. Then as long as unvisited nodes in stack, or if node is not null, we push nodes along the rightmost path to the stack.(i.e. guaranteed to process right subtree first).
Then we visit the node on top of the stack, and then consider its left subtree. This is like visiting current node before recursing on node's left subtree.
Terminates when all nodes are visited (stack is empty) and node points to left child of tree's minimum node which is null.

```
def convertBST(self, root):

    maxSoFar = 0
    stack = []
    node = root

    while stack or node:
        # Append curr to stack
        # Then keep going right to get the max in BST
        while node:
            stack.append(node)
            node = node.right
        # When we reach the very right, pop it off
        node = stack.pop()
        # Do something
        node.val += maxSoFar
        maxSoFar = node.val
        # If leaf node, node.left is null so we pop next
        # Otherwise we do the smaller value
        node = node.left

    return root
```

Time: O(n). Each node is pushed onto stack once.
Space: O(n). Each node is pushed onto stack once. Stack has at most n nodes.

Morris In-Order Traversal (Reverse In order)

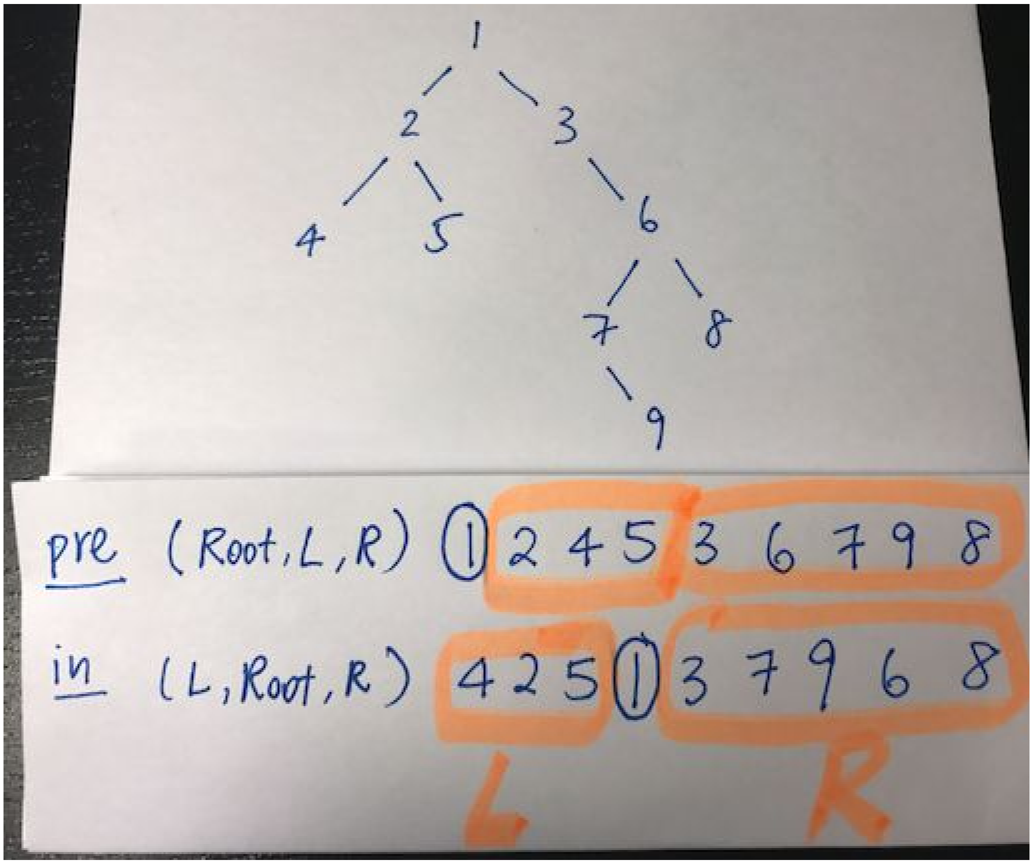
See Algorithms for detailed Morris BST Traversal. This is in-order, but reversed. We do a mirror image flip (left → right, right → left). Replace # do sth.

```
def convertBST(self, root):

    if not root:
        return None

    curr = root
    maxSoFar = 0

    while curr is not None:
        if not curr.right:
            maxSoFar = curr.val = curr.val + maxSoFar
            curr = curr.left
        else:
            pred = curr.right
            while (pred.left is not None and pred.left != curr):
                pred = pred.left
            if pred.left is None:
                pred.left = curr
                curr = curr.right
            else:
                pred.left = None
                maxSoFar = curr.val = curr.val + maxSoFar
                curr = curr.left
    return root
```

Lowest Common Ancestor BST

Build BST from Preorder and Inorder List

Idea 1: Using recursion. Break into subproblems.


1
2
3
4
5
6
7
8
9
pre (Root, L, R) 1 2 4 5 3 6 7 9 8
in (L, Root, R) 4 2 5 1 3 7 9 6 8
L
R

```
def buildTree(self, preorder, inorder):
    if preorder == []:
        return None
    if len(preorder) == 1:
        return TreeNode(preorder[0])

    headVal = preorder[0]
    headIndex = inorder.index(headVal)
    left_inorder = inorder[:headIndex]
    right_inorder = inorder[headIndex + 1:]

    left_preorder = preorder[1: 1 + headIndex]
    right_preorder = preorder[1 + headIndex:]

    root = TreeNode(headVal)
    root.left = self.buildTree(left_preorder, left_inorder)
    root.right = self.buildTree(right_preorder, right_inorder)

    return root
```

Even elegant recursive solution:

We build the left subtree first, so py **popping** it we automatically shorten preorder. Preorder is mutable - we automatically extract out elements from leftsubtree before building right subtree.

```python
def buildTree(self, preorder, inorder):
    if inorder:
        head_index = inorder.index(preorder.pop(0))
        root = TreeNode(inorder[head_index])
        root.left = self.buildTree(preorder, inorder[:head_index])
        root.right = self.buildTree(preorder, inorder[head_index + 1:])
        return root
```

Iteratively:

E. Dynamic Programming, Recursion + Memoization

Climbing Stairs

Each time we can climb one or two steps. How many steps to reach the end?

Idea 1: Recursive, top down with memoization

```
    def climbStairs(self, n, memoi={}):
        if n in memoi.keys():
            return memoi[n]
        if n <= 1:
            memoi[n] = 1
            return 1
        if n == 2:
            memoi[n] = 2
            return 2
        total = self.climbStairs(n-1) + self.climbStairs(n-2)
        memoi[n] = total
        return total
```

Dynamic Programming, Bottom Up approach:

```
    def climbStairs(self, n):
        if n == 1:
            return 1

        arr = [0, 1]
        for i in range(2, n+2):
            arr.append(arr[i-1]+arr[i-2])

        return arr[-1]

OR

    def climbStairs(self, n):
        a, b = 1, 1
        for _ in range(n):
            # a is ways to reach current step, b is ways to reach next step
            a, b = b, a+b
        return a
```

Jump Game: to End of Array

Given array of non-negative int, and initially at start, we can jump at most n steps (n is element in array). Determine if we can reach the end of the array.

E.g. [2, 3, 1, 1, 4] returns true (jump one step from 2 to 3, then jump 3 steps from 3 to end);
[3, 2, 1, 0, 4] returns false because we cannot get past 0.

Idea 1: Recursive top-down: **(NOT EFFICIENT)**

```
    def canJump(self, nums, pos = 0):
        """
        :type nums: List[int]
        :rtype: bool
        """
        if nums[0] >= len(nums)-1:
            return True

        # If it is 0, we return false because we can't go further
        if nums[0] == 0:
            return False

        for i in range(1, nums[0] + 1):
            if self.canJump(nums[i:]):
                return True
        return False
```

Some issues here: what to store in memoization?? Arrays - not space efficient; index: length changes all the time; not a good solution!

Idea 2: Recursive, with memoization:

Idea 3: Dynamic Programming, bottom-up, no recursion:

Idea 4:

F. DFS, BFS, Graph Search Algorithms

DFS Using Stack

1. Push to stack, pop top
2. Push unvisited neighbors to stack
3. Add to visited
4. Repeat 1,2,3 while stack not empty

DFS: STACK. BFS: QUEUE.

DFS USING RECURSION TEMPLATE

```
def traverse(matrix):
      rows, cols = len(matrix), len(matrix[0])
      visited = set()
      directions = [(0, 1), (0, -1), (-1, 0), (1, 0)]

      def dfs(curr_row, curr_col, mat):

            # check if visited first
            if (curr_row, curr_col) in visited:
                  return

            visited.add((curr_row, curr_col))

            for add_x, add_y in directions:
                  next_row, next_col  = curr_row + add_x, curr_col + add_y

                  # add other condition checking here
                  if 0 <= next_row < rows and 0 <= next_col < cols:
                        dfs(next_row, next_col, mat)


      for i in range(rows):
            for j in range(cols):
                  dfs(i, j, matrix)
```

Number of Islands

Return the number of islands, surrounded by ‘1’ in horizontal/vertical regions.

```
def numIslands(self, grid):
    # check for conditions first, then mark as visited then visit neighbors
    # Marking done is not reversed to explore island
    def dfs(grid, row, col):
        if row < 0 or col < 0 or row >= len(grid)
        or col >= len(grid[0]) or grid[row][col] != '1':
            return
        grid[row][col] = '#'
        dfs(grid, row+1, col)
        dfs(grid, row-1, col)
        dfs(grid, row, col+1)
        dfs(grid, row, col-1)

    count = 0
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if grid[i][j] == '1':
                # marks region of i, j
                dfs(grid, i, j)
                count += 1
    return count
```

Time: O(row * col). Each element visited once if valid.

Space: O(row * col).

Get Overlapping Regions

From Twitter, like No. of Islands.

Word Search

Idea: DFS. Check two things at the start.

```
def exist(self, board, word):

    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    def dfs(matrix, currWord, x, y):
        if currWord == '':
            return True
        if x < 0 or x >= len(matrix) or y < 0 or y >= len(matrix[0]) or
currWord[0] != board[x][y]:
            return False
        matrix[x][y] = '#'
        for add_x, add_y in directions:
            next_x, next_y = x + add_x, y + add_y
            if dfs(matrix, currWord[1:], next_x, next_y):
                return True
        matrix[x][y] = currWord[0]
        return False



    for i in range(len(board)):
        for j in range(len(board[0])):
            if dfs(board, word, i, j):
                return True

    return False
```

01 Matrix: Shortest Distance to 0

Shortest Distance. Clue to use BFS, follow template

```
def updateMatrix(self, matrix):

    numRows, numCols = len(matrix), len(matrix[0])
    toReturn = matrix
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    def getDistance(matrix, x, y):
        queue, visited = [(x, y, 0)], set()
        while (len(queue) != 0):
            curr_x, curr_y, curr_dist = queue.pop(0)
            if matrix[curr_x][curr_y] == 0:
                return curr_dist
            for add_x, add_y in directions:
                next_x, next_y = curr_x + add_x, curr_y + add_y
                if 0 <= next_x < numRows and 0 <= next_y < numCols
            and (next_x, next_y) not in visited:
                    visited.add((next_x, next_y))
                    queue.append((next_x, next_y, curr_dist + 1))
        return -1

    for i in range(numRows):
        for j in range(numCols):
            toReturn[i][j] =  getDistance(matrix, i, j)

    return toReturn
```

Word Ladder

Classic Example of BFS.

```
def ladderLength(self, beginWord, endWord, wordList):
    letters, visited , wordList = "abcdefghijklmnopqrstuvwxyz", set(), 
set(wordList)
    queue = [(beginWord, 1)]
    while len(queue) != 0:
        word, length = queue.pop(0)
        if word == endWord:
            return length
        for i, char in enumerate(word):
            for sub in letters:
                newWord = word[:i] + sub + word[i+1:]
                if newWord in wordList and newWord not in visited:
                    visited.add(newWord)
                    queue.append((newWord, length + 1))
    return 0
```

Space Complexity: O(n).
Time Complexity: ??

Idea 2: Bidirectional BFS, or maybe A*?

Number of Connected Components in Undirected Graph

Given n number of nodes and a List [List [int]] of edges, find number of connected components (similar idea to no. of islands).

Idea 1: DFS. Using recursion, modify edges directly.

```
def dfs(number, edges):
      for edge in edges:
            if number in edge:
                  edge.remove(number)
                  otherNumber = edge[0]
                  edges.remove(edge)
                  dfs(otherNumber, edges)

def find_connected_components(n, edges):
      toReturn = 0
      for i in range(n):
            for edge in edges:
                  if i in edge:
                        dfs(i, edges)
                        toReturn += 1
      return toReturn
```

G. Backtracking

Key idea: Avoid double counting. **For loop. In for loop, append. recurse, pop**. Think about how it works in the memory stack! ALSO APPEND **curr[:] instead of curr** because lists are passed by reference.

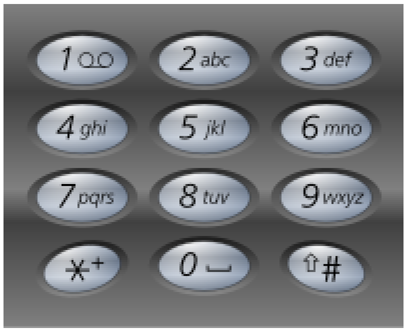
In each backtrack stack:

Do A
Do B

GENERAL TEMPLATE

```
def problem(self, nums):
    toReturn = []

    def backtrack(temp, arr):
        # check if fulfilled requirements
        if meet conditions:
            toReturn.append(temp[:])

        for i in arr:
            temp.append(i)
            backtrack(temp, arr)
            temp.pop()

    backtrack([], nums)
    return toReturn
```

Letter Combinations of Phone Number



Input: "23"

Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

Idea 1: DFS, recursive approach

```
def letterCombinations(self, digits):

    table = {"1":"", "2":"abc", "3":"def", "4":"ghi", "5":"jkl", "6":"mno",
"7":"pqrs", "8":"tuv", "9":"wxyz", "0":" ", }

    toReturn = []

    if digits == "":
        return toReturn

    def generateString(num, keypad):
        ans = []
        if num == "":
            return [""]
        if num[0] in keypad:
            candidate = keypad[num[0]]
            for c in candidate:
                for rest in generateString(num[1:], keypad):
                    ans.append(c + rest)
            return ans

    return generateString(digits, table)
```

Idea 2: BFS.

```
def letterCombinations(self, digits):

    table = {"1":"", "2":"abc", "3":"def", "4":"ghi", "5":"jkl", "6":"mno",
"7":"pqrs", "8":"tuv", "9":"wxyz", "0":" "}

    if digits == "":
        return []

    toReturn = [""]
    while len(toReturn[0]) != len(digits):
        curr = toReturn.pop(0)
        next_digit = digits[len(curr)]
        for candidate in table[next_digit]:
            toReturn.append(curr + candidate)

    return toReturn
```

Generate Parenthesis

Backtracking key! Review this again

Permutations

Follow template. Terminate condition when the lengths are the same.

```
def permute(self, nums):
    toReturn = []

    def backtrack(temp, arr):
        if len(temp) == len(arr):
            toReturn.append(temp[:])

        for i in arr:
            if i not in temp:
                temp.append(i)
                backtrack(temp, arr)
                temp.pop()

    backtrack([], nums)
    return toReturn
```

Subsets

Backtrack template:

For loop. In for loop, append. recurse, pop.

```
def subsets(self, nums):
    toReturn = []

    def backtrack(curr, start):
        toReturn.append(curr[:])
        for i in range(start, len(nums)):
            curr.append(nums[i])
            backtrack(curr, i+1)
            curr.pop()


    backtrack([], 0)
    return toReturn
```

Word Search

H. Matrices

Is Valid Sudoku

Idea 1: Use strings, loop each elem once, do check in set

```
def isValidSudoku(self, board):

    seen = set()

    def isValidSudoku(elem, i, j):
        if (str(elem) + " in row " + str(i)) in seen or
             (str(elem) + " in col " + str(j)) in seen or
             (str(elem) + " in box " + str(i//3) + ", " + str(j//3)) in
seen:
            return False
        return True

    for i in range(9):
        for j in range(9):
            elem = board[i][j]
            if elem != '.':
                if not isValidSudoku(elem, i, j):
                    return False
                seen.add(str(elem) + " in row " + str(i))
                seen.add(str(elem) + " in col " + str(j))
                seen.add(str(elem) + " in box " + str(i//3) + ", " +
str(j//3))

    return True
```

Idea 2: Make each stand out with tuples, add to set

```
def isValidSudoku(self, board):

    arr = []
    for i, row in enumerate(board):
        for j, elem in enumerate(row):
            if elem != '.':
                arr.append(('row' + str(i), elem))
                arr.append(('col' + str(j), elem))
                arr.append((i//3, j//3, elem))
    return len(arr) == len(set(arr))
```

Rotate Image

Algorithm to rotate image clockwise: first mirror along horizontal axis, then reflect along diagonal.

1 2 3		7 8 9		7 4 1	
4 5 6	→ Flip →	4 5 6	→ reflect →	8 5 2	
7 8 9		1 2 3		9 6 3	(Clockwise rotation)

For anticlockwise, flip along vertical axis, then reflect

1 2 3		3 2 1		3 6 9	
4 5 6	→ Flip →	6 5 4	→ reflect →	2 5 8	
7 8 9		9 8 7		1 4 7	(Anticlockwise rotation)

```python
def rotate(self, matrix):

    def horizontalMirror(matrix):
        for i in range(len(matrix)//2):
            matrix[i], matrix[len(matrix)-i-1] = matrix[len(matrix)-i-1],
matrix[i]
        return matrix

    def symmetricFlip(matrix):
        for i in range(len(matrix)):
            for j in range(i, len(matrix[0])):
                matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
        return matrix

    matrix = horizontalMirror(matrix)
    matrix = symmetricFlip(matrix)
```

Idea 2: Pythonic reverse and transpose.

Transpose: use zip.

```python
def rotate(matrix):
      matrix[:] = zip(*matrix[::-1])
```

Set Matrix Zeroes

If an element is zero, set the col and row to be zero. Issue: by setting and without using additional space, we overwrite other data. To solve this, we use first row and col as an indicator.

First we check if we need to set first row and first col to be zero, using two booleans. Next for the smaller submatrix starting from [1:] of size (m-1) * (n-1), we set the corresponding index to be zero in the first col and first index.

1. Scan first row, see if we need to set firstRowZero to zero.
2. Scan first col, see if we need to set firstColZero to zero.
3. Using first col and first row as auxiliary arrays, consider submatrix from second row onwards and second col onwards, update index of auxiliary arrays accordingly.
4. Finally using firstRowZero and firstColZero update first col and first row.

```
def setZeroes(self, matrix):
    firstRowZero, firstColZero = False, False
    if 0 in matrix[0]:
        firstRowZero = True
    if 0 in [row[0] for row in matrix]:
        firstColZero = True

    for i in range(1, len(matrix)):
        for j in range(1, len(matrix[0])):
            if matrix[i][j] == 0:
                matrix[0][j] = 0
                matrix[i][0] = 0

    for i in range(1, len(matrix)):
        if matrix[i][0] == 0:
            matrix[i] = [0] * len(matrix[0])

    for j in range(1, len(matrix[0])):
        if matrix[0][j] == 0:
            for k in range(len(matrix)):
                matrix[k][j] = 0

    if firstRowZero:
        matrix[0] = [0] * len(matrix[0])
    if firstColZero:
        for row in matrix:
            row[0] = 0
```

Time: O(mn), Space: O(1). Modifying in place.

I. Sorting and Searching

Merge 2 sorted arrays

Input:
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],   n = 3
Output: [1,2,2,3,5,6]

Idea: start from back of array, merge it into nums1 from the back. Think about 107 memcpy!

```
def merge(self, nums1, m, nums2, n):
    while n > 0 and m > 0:
        if nums1[m-1] >= nums2[n-1]:
            nums1[m+n-1] = nums1[m-1]
            m -= 1
        else:
            nums1[m+n-1] = nums2[n-1]
            n -= 1
    if n > 0:
            nums1[:n] = nums2[:n]
```

Sort Colors

Given an array with *n* objects colored red, white or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note: You are not supposed to use the library's sort function for this problem.

Example:

```
Input: [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]
```

Follow up:

- A rather straightforward solution is a two-pass algorithm using counting sort.
- First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.
- Could you come up with a one-pass algorithm using only constant space?

```
def sortColors(self, nums):
    next_0_to_swap, next_2_to_swap = 0, len(nums) - 1
    i = 0
    while i <= next_2_to_swap:
        if nums[i] == 0:
            nums[next_0_to_swap], nums[i] = nums[i], nums[next_0_to_swap]
            i += 1
            next_0_to_swap += 1
        elif nums[i] == 1:
            i += 1
        elif nums[i] == 2:
            # don't increase i here, after swap it might be 0 or 1
            nums[next_2_to_swap], nums[i] = nums[i], nums[next_2_to_swap]
            next_2_to_swap -= 1
```

Top k Frequent Elements (Bucket Sort)

Given non-empty array of int, return top k frequent elements.

E.g. Given [1, 1, 1, 2, 2, 3] and k = 2, return [1, 2]
Given [1] and k = 1, return [1]

```
def topKFrequent(self, nums, k):
    from collections import Counter
    nums_counter = Counter(nums)
    return [i for i, j in nums_counter.most_common(k)]
```

Binsert, Insert in Sorted Array

Given sorted array and target, return position where target belongs to.
E.g. Insert([1, 3, 5, 6], 5) returns 2; Insert([1, 3, 5, 6], 7) returns 4

Idea 1 O(n):

```
    def searchInsert(self, nums, target):
        for i, elem in enumerate(nums):
            if elem < target:
                continue
            return i
        return len(nums)

OR

        return len([x for x in nums if x < target])
```

Idea 2 O(lg n) Binsert:

```
    def searchInsert(self, nums, target):
        # base case
        if len(nums) == 1:
            if target > nums[0]:
                return 1
            else:
                return 0

        # splice array in half, does binsert recursively
        midpoint = len(nums)//2
        if target <= nums[midpoint - 1]:
            return self.searchInsert(nums[:midpoint], target)
        else:
            return midpoint + self.searchInsert(nums[midpoint:], target)
```

First Bad Version (Similar to Binsert)

Watch out for overflow: instead of doing (low + high), do low + (high - low)/2.

Idea: Like binsearch, but watch for boundaries. If it is in bottom half, we simple move top down (right to midpoint). Otherwise, we move left to midpoint+1, but before that check for the one on the left.

| Left | | Right |

1   2   3   4   5   6   7   8   9

```
def firstBadVersion(self, n):
    left, right = 1, n
    while left < right:
        midpoint = (left + right)//2
        if isBadVersion(midpoint):
            right = midpoint
        else:
            if isBadVersion(midpoint+1):
                return (midpoint+1)
            left = midpoint + 1
    return left
```

Kth Largest element

E,g, Given nums = [3, 2, 1, 5, 6, 4], target = 2, return second largest element, 5.

Idea: call quickselect.

Quickselect O(n) in place

Selects the kth index. E.g. nums = [3, 2, 1, 5, 6, 4], target = 4, returns 5
because sorted is [1, 2, 3, 4, 5, 6] and at 4th index, we return 5.

```
import random

def quickselect(arr, k):
      def select(nums, target_idx, low, high):
            if low == high:
                  return nums[low]

            pivot_idx = random.randint(low, high)
            pivot = nums[pivot_idx]

            # swap pivot to front
            nums[pivot_idx], nums[low] = nums[low], nums[pivot_idx]
            pivot_idx = low

            # partition
            for i in range(low + 1, high + 1):
                  if nums[i] < pivot:
                        pivot_idx += 1
                        nums[pivot_idx], nums[i] = nums[i], nums[pivot_idx]


            nums[pivot_idx], nums[low] = nums[low], nums[pivot_idx]

            if pivot_idx == target_idx:
                  return pivot
            elif target_idx < pivot_idx:
                  return select(nums, target_idx, low, pivot_idx - 1)
            else:
                  return select(nums, target_idx, pivot_idx + 1, high)

      return select(arr, k, 0, len(arr) - 1)
```

```
[3, 2, 1, 5, 6, 4]

[2, 3, 1, 5, 6, 4] 0
[2, 1, 3, 5, 6, 4] 1
[2, 1, 3, 5, 6, 4] 1
[2, 1, 3, 5, 6, 4] 1
[2, 1, 3, 5, 6, 4] 1

[1, 2, 3, 5, 6, 4]
```

J. Other Useful Algorithms

Kadane's Algorithm

Applications: Largest Contiguous Sum Subarray, Best time to Buy/Sell Stock

Idea: **To include or not to include, that is the question.** Does including the next element increase the current sum? Compare to global sum?

```
def longestContiguousSubarray(arr):
    # Note: don't start with 0 (fails if arr is all negative)
    globalMax, currMax = arr[0], arr[0]
    for i in arr[1:]:
        currMax = max(i, currMax + i)
        globalMax = max(currMax, globalMax)
    return globalMax
```

Time: O(n)

Best time to buy/sell stock:
Input: [7,1,5,3,6,4]
Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5. Not 7-1 = 6, as selling price needs to be larger than buying price.

Idea: Same as continuous subarray problem, but is based on profit and not prices

```
def longestContiguousSubarray(arr):
    # Note: don't start with 0 (fails if arr is all negative)
    globalMax, currMax = 0, 0
    for i in range(1, len(arr)):
        profit = arr[i] - arr[i-1]
        currMax = max(0, currMax + profit)
        globalMax = max(currMax, globalMax)
    return globalMax
```

Manacher’s Algorithm

Longest Palindromic Substring

We make use of symmetric property (WHICH ONLY HOLDS UNDER LONGEST PALINDROME). Hence we have to keep checking for bounds (symmetric property only applies to element in bounds of the palindrome so far).

Here we have 5 steps: (Initialize center and right to be zero first)

1. preprocess string to take care of even/odd lengths. Append EOF/SOF to both ends, dashes between chars. (abba → !-a-b-b-a-?)
2. Skip SOF/EOF, and loop through. For each char, get the mirror.
3. See if i is less than right. If so we can use the mirror. Take the min of mirror and right-i.
4. Now we have the minimum length of palindrome centered at i. Keep expanding out from this minimum length to get longest palindromic substring at i, store in array.
5. Update right and center if necessary. If right edge of palindrome at i extends beyond previous right edge, we set i to be the new center, and update right.

To get the start, we have maxLen and Center. We take center, minus SOF (!) and divide by 2 to get rid of dashes. Also minus maxLen/2 to get the start.

```
def preprocess(s):
      if s == "":
            return '!?'
      toReturn = '!'
      for char in s:
            toReturn = toReturn + '-' + char
      return toReturn + '-?'

def longestPalindromicSubstring(s):
      # 1. Preprocess first
      t = preprocess(s)
      palindromeArr = [0] * len(t)

      # initializes center, right to be zeroes
      center, right = 0, 0

      for i in range(1, len(t) - 1):
            # 2. get mirror
```

```
        mirror = 2 * center - i

        # 3. check if we can use mirror. Always take minimum.
        if i < right:
            palindromeArr[i] = min(palindromeArr[mirror], right - i)

        # 4. keep expanding from minumum if both ends match
        #finds the longest palindrome centered at i using previous info

        while t[i - palindromeArr[i]-1] == t[i + palindromeArr[i]+1]:
            palindromeArr[i] += 1

        # 5. Updates center and right boundary if necessary
        #update when current right edge extends beyond previous right
        if right < i + palindromeArr[i]:
            right = i + palindromeArr[i]
            center = i
    print (palindromeArr)
    maxLen = max(palindromeArr)
    center = palindromeArr.index(maxLen)

    # minus 1 to remove START (!)
    # divide by 2 to remove dashes
    # also subtract half of maxLen
    start = (center - 1 - maxLen)//2
    end = maxLen + start
    print (start, end)
    return s[start:end]
```

Time: O(2n) = **O(n)**. It is amortized worst case, only move right towards the end without repeating hence O(n). Along by looping through each element, we have O(2n).

Could be O(3n) by finding max. Space: **O(n)**.

Todo: sliding window technique, Rabin Karp, KMP

Boyer-Moore Voting Algorithm

Applications: Majority element in array (more than half)
Intuition

If we had some way of counting instances of the majority element as +1+1 and instances of any other element as -1–1, summing them would make it obvious that the majority element is indeed the majority element.

Algorithm

Essentially, what Boyer-Moore does is look for a suffix, suf of nums, where suf[0] is the majority element in that suffix. To do this, we maintain a count, which is incremented whenever we see an instance of our current candidate for majority element and decremented whenever we see anything else. Whenever count equals 0, we effectively forget about everything in nums up to the current index and consider the current number as the candidate for majority element. It is not immediately obvious why we can get away with forgetting prefixes of nums - consider the following examples (pipes are inserted to separate runs of nonzero count).

[7, 7, 5, 7, 5, 1 | 5, 7 | 5, 5, 7, 7 | 7, 7, 7, 7]

Here, the 7 at index 0 is selected to be the first candidate for majority element. count will eventually reach 0 after index 5 is processed, so the 5 at index 6 will be the next candidate. In this case, 7 is the true majority element, so by disregarding this prefix, we are ignoring an equal number of majority and minority elements - therefore, 7 will still be the majority element in the suffix formed by throwing away the first prefix.

[7, 7, 5, 7, 5, 1 | 5, 7 | 5, 5, 7, 7 | 5, 5, 5, 5]

Now, the majority element is 5 (we changed the last run of the array from 7s to 5s), but our first candidate is still 7. In this case, our candidate is not the true majority element, but we still cannot discard more majority elements than minority elements (this would imply that count could reach -1 before we reassign candidate, which is obviously false).

Therefore, given that it is impossible (in both cases) to discard more majority elements than minority elements, we are safe in discarding the prefix and attempting to recursively solve the majority element problem for the suffix. Eventually, a suffix will be found for which count does not hit 0, and the majority element of that suffix will necessarily be the same as the majority element of the overall array.

Traversing BST with Stack

Morris In-Order BST Traversal (Also applied to reverse in-order)

Traverse a BST without stacks or recursion. **O(n) time, O(1) space.**

Remove the link made pointing to curr from predecessor. Remove only after moving curr back.

```
def InOrderMorrisTraverse(root):
      # set current to root of BST
      curr = root
      while curr is not None:
            # if nothing is at the left, do sth and move on.
            if curr.left is None:
                  print (curr.val)                    # do sth
                  curr = curr.right
            else:
                  # find the inorder predecessor of current
                  pred = curr.left
                  while (pred.right is not None and pred.right != curr):
                        pred = pred.right

                  # make current the right child of predecessor
                  if pred.right is None:
                        pred.right = curr
                        curr = curr.left
                  # remove the modified BST, fix right child of predecessor
                  else:
                        pred.right = None
                        print (curr.val)
                        curr = curr.right
```

Check out Morris PreOrder/PostOrder Traversal

Python Useful Tips

Extract alphanumeric:

```
s = re.sub("[^a-zA-Z0-9], "", s)
```

Get values from dictionary:

```
dictionary.values()
```

Get values from dict and increment:

```
foo[bar] = foo.get(bar, 0) + 1
```

Remove element from array:

```
arr.remove(elem)
```

Remove from index, array:

```
arr.pop(index)
```

Remove element of 1 list from the other:

```
[x for x in l1 if x not in l2]
```

Reverse array, string:

```
arr = reversed(arr) or arr[::-1]
```

Combine multiple list, get first entry all at once:

```
zip(arr1, arr2), zip(*[args0)
```

Convert list to dict

```
>>> list_to_dict(['alpha:1', 'beta:2', 'gamma:3'])
{'alpha': '1', 'beta': '2', 'gamma': '3'}
```

```
 def listToDict(list):
      return dict(map(lambda s: s.split(';'), rlist))
```

```
Create 2d matrix
[[0 for j in range(numCols) for i in range(numRows)]
```

```
Counter Operations
      Subtract one from the other: counter1.subtract(counter2)
      get most common: counter1. most_common()
      iterate through and get all like list: counter1.elements()
```

Get column j from matrix:

```
[row[j] for row in matrix]
```

Gusto Interview

```
#
# get_nth_leaf()
#
# Given a tree data structure, write a method to return the Nth leaf of the the tree
#
#          A
#      B        C
#    D E F    G   H
#           J K
#
# 1. Make it work
# 2. Make it right
# 3. Make it fast
#
# Is it a binary search tree? No. Nodes can have multiple children
#
# Is the input always guaranteed to be valid? Let's worry about "guard" cases until the
end
#
# get_nth_leaf('hello', 0)
#
# idea: in order traversal of the trees (Recursive/Iterative)
#
# get_nth_leaf(node, 0) => Node('D')
# get_nth_leaf(node, 1) => Node('E')
# get_nth_leaf(node, 2) => Node('F')
# get_nth_leaf(node, 3) => Node('J')

"""
node {
   int
   # left
   # right
   (0 - n children)
   [node pointer1, node pointer 2]
}

get all the leaves
D, E, F
"""


#          A
#      B        C
#     D E F    G   H
#            J K

class Node(object):
   def __init__(self, children, value):
       self.children = children
       self.value = value

def find_nth_leaf(startNode, n):
   stack = []
   counter = 0
   curr = startNode

   # for making it right
   try:
       # in order traversal, recursively
       while (curr or len(stack) != 0):

           while (curr):
               # append (node, index of children exploring) to stack
               stack.append((curr, 0))
               if curr.children != []:
                   curr = curr.children[0]
               else:
                   curr = None

           # stack is [(A,0), (B, 3)]
           curr, idx = stack.pop()

           # curr is F, 0
           if curr.children == []:
               if counter == n:
                   return curr.value
               counter += 1
               curr = None

           else:
           # curr is not a lead
           # idx is currently 0
               if idx < len(curr.children) - 1:
                   stack.append((curr, idx+1))
                   curr = curr.children[idx+1]

               else:
                   curr = None

   except Exception as e:
        print (e)


#           A
#      B        C
#    D E F    G   H
#            J K


node_D = Node([], 'D')
node_E = Node([], 'E')
node_F = Node([], 'F')
node_J = Node([], 'J')
node_K = Node([], 'K')
node_H = Node([], 'H')
node_B = Node([node_D, node_E, node_F], 'B')
node_G = Node([node_J, node_K], 'G')
node_C = Node([node_G, node_H], 'C')
node_A = Node([node_B, node_C], 'A')

empty_node = Node([], 'Empty')
start = node_A
print (find_nth_leaf(empty_node, 1))

# "Make it fast" -> Let's talk about the performance characteristics
#
# What do we expect the timing aspects of our solution to be?
# What do we expect the memory aspects of our solution to be?
# n is number of nodes
# Time: O(n), since we are exploring every node at least once
# Memory: O(2 lgn) == O(lg n)

def get_leaves(node):
    leaves = []
    if node.children == []:
        leaves.append(node)
    else:
        for child in node.children:
            leaves += get_leaves(child)

    return leaves

def find_nth_leaf_2nd(node, n):
    return get_leaves(node)[n]


# Time: O(n), since we are also exploring every node
# Memory: O(n), since there are n calls to the recursive function

print (find_nth_leaf_2nd(node_A, 0).value)
```

Google Interview

Wrap-Up

Very useful:
https://workflowy.com/s/wGqavcPQFm

http://i.imgur.com/JbVwvxo.jpg

Algorithms, Data Structures, Computer Science, Math
Java
Concepts
JVM
Heap vs. Stack
Comparable vs. Comparator
HashCode() and equals() Contract
Simple Java Review
Java Collections
Collection
Collection API
Subclasses
List API
ArrayList
LinkedList
Set API
HashSet
LinkedHashSet
TreeSet
EnumSet
Queue API
Dequeue
Queue
Stack
PriorityQueue
Map
Map API
HashMap
TreeMap
LinkedHashMap
IdentityHashMap
WeakHashMap
Sorted Map API
Hash Table
Data Structure Comparisons
ArrayList vs. LinkedList
ArrayList
LinkedList
HashSet vs. TreeSet
CS Concepts
Big O & Time Complexity
Big O
Truth Table
Base 2 Table
Binary to Decimal Table
Boolean Properties
Bit Manipulation
Bitwise Operators
AND (&)
OR (|)
NOT (~)
XOR (^)
Left Shift (<<)
Right Shift (>>)
Useful Operations
XOR Properties
Dynamic Programming
DP Explanation
Solving a DP problem
Greedy Algorithm
Backtracking
General Algorithm
System Design Concepts
Concurrency
Parallelism
Race Condition
Locking
Livelock
Deadlock
Pessimistic
Optimistic
Optimistic vs. Pessimistic
Semaphore
Mutual Exclusion
Mutex vs. Semaphore
Commits
Two-Phase
Three-Phase
One-Phase
Design Patterns
Creational
Structural
Behavioral
P vs NP Problem
Deterministic vs Non-Deterministic
Deterministic
Nondeterministic
Flow of Control
Iteration
Recursion
Traditional Recursion
Tail Recursion
General Knowledge
ASAP Method
Fermi Problems
Solve by Dimension Analysis
Math
Math Theory
Data Types
Math Properties
Logical Formulas
Set Theory
Set Operations
Set Theorems
Counting
Pieonhole Problem
Numbers
Prime Numbers
List of Primes
GCD of Two Numbers
Algorithm Theory
Classifying Algorithms
By Problem Domain
Numeric
String
Sorting
Searching
Combinatoric
Partitioning
Network
By Design Strategy
Brute Force
Pre-Structuring
Divide and Conquer
Decrease and Conquer
Transform and Conquer
Input Enhancement
Dynamic Programming
Greedy Algorithm
Backtracking
Branch and Bound
Other Strategies
Hill Climbing
Particle Swarm
Las Vegas
Monte Carlo
By Complexity
Constant O(1)
Logarithmic O(log n)
Linear O(n)
Linearithmic O(n log n)
Polynomial
Quadratic O(n^2)
Cubic O(n^3)
Exponential O(k^n)
By Data Structure
Organizational
Set
Linear
Hierarchical
Network
Implementational
Hash Tables
Trees
Skip Lists
Data Structures
Choosing the Right Data Structure
API Methods to Know
General
Boyer-Moore Algorithm
Concept
Stable vs Unstable
Quicksort vs Mergesort

Guides

https://www.reddit.com/r/cscareerquestions/comments/6278bi/

https://www.reddit.com/r/cscareerquestions/comments/8235gs/got_my_dream_job_offer_thanks_to_this_sub_tips/

All processes including resume etc:
https://www.reddit.com/r/cscareerquestions/comments/5rc9z8/a_comprehensive_guide_to_getting_an_internship/

Leetcode Grinding Guide:
https://www.reddit.com/r/cscareerquestions/comments/6luszf/a_leetcode_grinding_guide/

Leetcode Explore

Internships:
https://www.intern.supply/

https://github.com/j-delaney/easy-application

https://docs.google.com/spreadsheets/d/1z3_OfZdPXiOOgHi7uY5nvLuQX_qbpaxmYNXHHiuuV0I/edit#gid=0

https://docs.google.com/spreadsheets/d/1XnjJMX2PGLbwhnCDSCrSejOsUddv9mr9hBt3h5D6_kk/htmlview

https://github.com/rootshaw/Internship-Preparation/blob/master/Companies.md

https://docs.google.com/spreadsheets/d/15eqdQCWwXOQZvl6TNehn12BuMTVNpAG7Q_ib3q-BS1Y/edit?usp=sharing

https://www.reddit.com/r/cscareerquestions/comments/468d0o/the_152_companies_i_applied_to_if_its_of_any_use/

Twitter coding: number of islands (identical islands), matrix (Mars Rover), collections counter of voters + get last name, huffman mapping (see how to convert arr to dict)