Read: http://steve-yegge.blogspot.com/2008/03/get-that-job-at-google.html
http://highscalability.com/
https://triplebyte.com/blog/how-to-pass-a-programming-interview
https://yangshun.github.io/tech-interview-handbook/algorithms/algorithms-introduction/

Data Structures

- Graph
  - Different ways of storing a graph
    - Uncommon ones:
      - A list of edges between numbered nodes, with a weight if needed (edge list)
      - Adjacency matrix (a V * V array with 0 and 1's if the edge exists). Quick edge lookup but thats it. Good if a graph is very connected, or if the edges have weights (because using node as an object or adjacency list, you may have to keep track of edge weights in a different structure and it requires synchronization
    - Common:
      - Adjacency list. Its an array or hash table with a list of edges at each element (key is the node, edges are the values their)
      - Nodes as objects and edges as pointers (OOP, edge list is an array or something)
        - ~~These can work better with directed graphs because you don't have to maintain the storage of edge at each vertex and dont have to worry about becoming unsynchronized~~
  - DFS, BFS (visited array), cycle detection
    - For visited array, it often makes sense to use a set() to keep track. Constant look-up time and easy initialization
    - **For recursion stack visited for cycles, you must use a deep copy when you pass in recursively otherwise you will pass by reference and screw it up**
    - Directed Graph
      - Connected - you can start from any node to DFS, by definition this type of graph has a cycle because you can get to any node FROM any node and for a directed graph it must have a cycle
      - Unconnected - If the graph is unconnected, you will have to iterate through each node and keep a visited set handy. Dont DFS anything visited already in the outer loop or recursive call
        - Cycle detection - create a cycle detection path/set in the recursion stack to see if the current path contains a cycle. Separate from the overall visited set
    - Undirected Graph

            - Connected - Can start from any node to DFS. May or may not contain a cycle
                - Cycle detection - can use the global visited or one in the call stack for it. **Must remember to skip the parent when calling the DFS on each neighbor. It doesn't qualify as a cycle**
            - Unconnected - must iterate through each node
                - Cycle detection - can use the global visited or one in the call stack for it. **Must remember to skip the parent when calling the DFS on each neighbor. It doesn't qualify as a cycle**
        - A graph is considered a tree IFF the following two conditions are met:
            - G is fully connected. In other words, for every pair of nodes in G, there is a path between them.
            - G contains no cycles. In other words, there is exactly one path between each pair of nodes in G.
- Binary Trees
    - Search, insert, predecessor
    - Traversal orders - the value of the order determines when the root is acted upon
        - In order is left, root, right. Done trivially with recursion/stack
        - Preorder is root, left right
        - Postorder is left, right, root
    - Constructing a complete tree with an array (**only works for a completed tree**): The left node of the current index is i * 2 + 1, the right node of the current index is i * 2 + 2. Could do a recursive function creating a TreeNode with value array[i], and setting its .left and .right to the i * 2 + 1 function until i becomes larger than n, then that's your base case and return the node that was passed in. Return the node you created at the end at each step if it's a new node
    - Typically you would want to always use a BST or a balanced BST. A "binary tree" is a class of data structures and wouldn't be implemented without any rules. The only exception would be, for example, a trie tree where there is no concept of "balancing"
- n-ary Trees
    - An n-ary tree is a tree that has no more than n children (conversely, a binary tree is just an n-ary tree where n = 2
    - A trie tree is a commonly used form of an n-ary tree
        - Trie tree - a trie tree is a "re**trie**val tree" and is commonly used to store things such as words or strings. It allows for a quick lookup time to see if a given word exists by the nature of the tree structure. The drawback to the time efficiency is that a trie tree uses up a lot of memory (each node can point to up to 26 others nodes for each letter in the alphabet)

        - Auto-suggest algorithms typically use a trie tree for quick suggestions (if you type “mathe” it could suggest to you quickly the rest of the words from that path
        - The runtime for a retrieval is O(n) where n is the length of the key you are looking for. An insert also takes O(n) time
        - Prefix nodes (the letters one-by-one leading up to the final word) and suffix nodes (the node marking the end of a word) are marked in a Trie
        - Not every node is a key like in a binary tree, if it’s a key it is marked accordingly
- AVL Self Balancing Tree - An AVL tree is a self-balancing binary search tree. At every insertion, it assures that one side of the tree is not to heavily stacked than the other side, eg the heights of two subtrees of any given node will never differ by more than a value of 1
    - This assures that your tree will retain its property of having O(logn) insert, retrieval, and deletion. This is the balance factor and it’s the difference between the height of two subtrees
    - The cost of this is that on an insertion or a deletion, if the balance factor becomes greater than |1|, you need to perform a rebalance operation while maintaining the structure of a binary search tree. The run time for this is O(logn) in the worst case
    - Other self-balancing trees include red/black tree
        - An AVL tree is more balanced than a red black (rb tree means root is black, and a red node can never have a red parent or child), but avl tree is more expensive with its rotations. So if you plan on using a lot of insertions and deletions, a RB tree may be a better choice if you need a balanced tree
- Balanced BST vs Hashmap
    - Hashmap is the efficient storing of unordered items. Exact match, constant time lookup
    - A BST is the efficient storing of ordered items. The lookup time is a little longer, O(logn), but it allows for operations like “find the value nearest to x” whereas a hashmap wouldn’t grant you that
- Linked Lists
    - Know tricky little things with this
- Implementing a hash map
    - Functions required:
        - Constructor (initial size of the array for hash function purposes, and the actual array populated with None values
        - Hash function - this function takes the key and hashes it to an index

                - Hash function example 1: Take every char and get the ord(char) of it, sum them, and then modulo by the size of the array in the hash map
                - Hash function example 2: Could just take the key as a string (use the hashcode of it possibly), convert it into an int, and modulo by table size
                - Basically as long as you manipulate the key in some randomized way, and then modulo by the table size, you'll have a decent hash function. Important thing is to **keep it simple**
            - Get - Take the key, hash it, and find it in the hash map
            - Add - Take the key, hash it, and add it to the hash map if it doesn't exist. If it does exist, replace the value
            - Delete - Take the key, hash it, then delete the key found in the list in that position
        - Load Factor λ - this is the (# of elements / tablesize). When load factor gets too large, you will want to resize the hashmap (80% works)
        - Collision detection
            - Linear probing - this process is defined as: if a collision is found, to find the nearest available slot in the map
                - The Get function needs to follow this same pattern!!
                - This can cause clustering, this can be resolved by jumping around in values larger than 1
                    - Every value in the map must be visited if this is the case otherwise parts will never get used. Make the table size a prime number for remedy this
            - Chaining - preferred method for me of making a list at each index slot and adding it there. And if the load factor gets too large, write a function to re-build the hash map at double the size and re hash + insert every item. This would take O(n) time where n is the number of keys in the map
- Minheap
    - Minheap in python is heapq! So import heapq, then use heapq.heappush(listname, elementtosorton)

    - If you use a tuple as the value to sort on, it will sort on the first value of the tuple by default
        - Then you use heapq.pop(listname) to get the minimum element
        - Use listnameofheap[0] to peek at the minimum element without popping
    - To use a max heap, you invert all the keys into negatives and use it the same way
- Hash table/map (Dictionary)
    - A hash table is a structure used to store arbitrary data, uses hashed keys. A specific implementation of the dictionary concept. It is thread-safe so if you need synchronized methods you would want to use this
    - A hash map is what a python dictionary uses (and is not thread-safe, but it is faster)
        - A map can be implemented using a self-balancing tree, a hash table, etc
    - A hash set/set is implemented by a hash table. Treeset by treemap (if needs to be sorted)
    - OrderedDict() preserves the order in which you added keys/elements
    - map = collections.OrderedDict()

Algorithms

- Recursion
    - Going from instance variable self for recursion to other method:
        - Pass in the returnvariable into the method
        - Make sure to return this variable where necessary so it can be used (in the base case, as the assignment to the recursive calls, and/or at the end of the function
        - If you're doing LinkedList stuff, make sure you're you're making copies of the List head if needed, and making new node objects. If you don't make new node objects, youll be perfomering unexpected actions with the nodes you're iterating with
- Greedy algorithms
    - Basically algorithms that try to find the most optimal solution for each local iteration in order to find the global optimization
        - Ex: Dijkstra's Minimum Spanning Tree Algorithm, traveling salesman
- Dijkstra - Finds all paths and distances between every node in the graph
    - A* uses the smallest of these paths. Gives priority to nodes that may be better rather than going through every single node
- Sorting

    - Know when to use different ones in different scenarios. Quicksort or timsort (merge + insertion) usually
    - Learn the generals of the sorting algorithms
        - Insertion Sort - https://www.geeksforgeeks.org/insertion-sort/ (best on already mostly sorted data)
            - Basically the same as how you would sort cards
            - Keep a “sorted” pointer to the end of the left sorted array, and everytime you encounter an element on the right side that is less than the sorted point, you iterate backwards moving each element up until that isn’t the case. When the unsorted side is 0, it is sorted.
            - Stable (meaning in the event of a tie, order remains the same)
            - Great on already sorted data (O(n))
        - Divide and conquer algorithms
            - Merge Sort - https://www.geeksforgeeks.org/merge-sort/
                - Merge sort recursively splits the array into two halves. Once the subarray equals 1, it sorts with its neighbors on the recursive trip back up (using the merge method). O(logn)
                - Once you have two sorted halves, you iterate through each half incrementing the side with the smaller number, adding it to an output array. O(n)
                - Stable
            - Quick Sort - https://www.geeksforgeeks.org/quick-sort/
                - Choose a pivot index, divide and conquer
                - Not stable
    - O(nlogn) is typically what you’re going to get with a sorting algorithm
        - list.sort()
        - list.sort(key=lambda x: x.start)
- Binary search
    - Know how to implement. Its a log(n) search
    - https://www.geeksforgeeks.org/binary-search/
    - start + (end - start) / 2
- Dynamic programming and memoization
    - Top Down vs Bottom Up
        - Top Down - nothing more than using recursion + memoization. You come up with a naive recursive solution, and then there will be many repetitive computations, so you use a memoization cache to store these for efficiency (tabulation)
            - You are starting with the problem itself (THE TOP), and breaking it down into the pieces needed to solve the top (and, the subproblems that arise throughout incidentally)

                - Another way to see it is that at each step you are finding the local solution, and aggregating them to solve the top
                - **Benefits**: easier to code/more intuitive
            - Bottom Up - this approach starts from the source, the smallest subproblem, what you know to be true (base case for example). Then using loops you store each result and eventually build the whole answer out
                - **Benefits**: sometimes problems with top down can't be optimized properly
    - Memoization is the solution of using a cache to store already computed results in order to prevent redundant calculations
    - When to use
        - A traditional bottom up DP algo is good when each subproblem must definitely be solved at least once (because you will be going in order anyway, so if each subanswer depends on the last, this is a good approach)
            - Also may want to use this if the stack for recursive calls gets too large
        - If some of the subproblems don't need to be solved, using top down/memoization may be better so you only have to compute the outputs that matter
    - Backtracking
        - Usually if backtracking/dfs is way too inefficient, dynamic programming is the answer. In some cases, a brute force with recursion is not nearly elegant enough
- Two pointers
    - If it seems like a two pointer problem, consider two options
        - Starting from i = 0, k = 1 and determining rules for when to increment
        - Starting pointers from both ends of the list and determining rules for when to increment or decrement
        - Starting from the middle
- Bit Manipulation
    - 583 in base 10 = (5 * 10^2) + (8 * 10^1) + (3 * 10^0)
        - So 101 in base 2 = (1 * 2^2) + (0 * 2^1) + (1 * 10^0) = 5
        - Base 2 you pretty much just choose from 1 or 0 and do the following like above
    - Signed vs unsigned pretty much means "is there a first bit sign attached to this number"
        - An unsigned number is assumed to be positive
    - Two's Complement for storing negative numbers: The first bit determines the sign
        - 0 is positive
        - 1 is negative

        - To get the negative representation of some bits, add the inverse of the bits, and then add 1 (binary) to the end of it
    - Binary to Decimal: Do the base2 thing above, break it up by column
    - Decimal to binary: Divide by 2 until you get to 0, rounding down each time. If it doesn't divide evenly, a 1 goes in that spot. The reverse of the output you just got is the number in binary
    - Shifts
        - Left Shift << (satisfies both signed and unsigned)
            - A single left shift moves every bit to the left by one
            - This multiplies the value by 2 in base 10, and that's because each bit becomes "more significant" given its placement
            - Only drop values off if it goes past 32 or 64 or 8
        - Right Shift >> (signed right shift, unsigned is >>>)
            - A single right shift moves every bit to the right by one
            - This divides by 2, because you are losing a significant bit
        - These shifts work for positive numbers and are LOGIC shifts
        - If your value is negative, you need to do an arithmetic shift to preserve the signage
            - This pretty much means you do the right shift, and then change the first bit to 1
    - XOR - is true when exactly ONE of the inputs is true. False every other time

Other

- Discrete Math (n choose k, basic probability etc)
    - N choose K stuff, familiarize
        - N choose K is given N unique elements, you can choose K of them. How many possibilites are there? This is the binomial coefficient
        - The formula is: (n!) / k! (n-k)!
- Bit Manipulation
    - https://wiki.python.org/moin/BitManipulation
- System Design
- Know to use regular expression (regex) and grep/sed to do a quick search and replace of a string
- Mapreduce (the algorithm for taking data from distributed sources, mapping them together onto the same memory space, and then reducing the redundant occurrences
- Object Oriented Programming
- Concurrency
    - Semaphore vs Mutex
        - Semaphore: A binary semaphore can be given and taken by any task.

            - If you take a semaphore you're waiting for something, and if a different tasks gives it, operation A continues. Its used as a trigger instead of protecting a resource
        - Mutex: Used to protect a shared data resource (array, list, database, etc)
            - Owned by the task that takes it, only released by that task
- Distributed hash table system
- Prime numbers (find first 1000 prime numbers,or first 1000 fibonacci numbers)
    - Prime number is only divisible evenly be 1 and itself
        - Would need to do a loop probably for each num and print it. Would be super slow
    - Fibonacci recursive and iterative
- NP Completeness - nondeterministic polynomial time. Problems in which solutions can be verified quickly but the solutions themselves cannot be found quickly as the time complexity increases rapidly as the size grows. NP and NP hard fall into this category
    - Traveling Salesman Problem - Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city? DP algo can solve it in O(n^2*2^n) [yuckkkk]

Python

- YOU CAN HASH A COUNTER BY MAKING IT A TUPLE!!!!!! HASH ANYTINH THAT Way. use tuple((countername)) with double parenthesis
- Use zip() to create an iterable of some sets of elements
    - Eg, if two datasets coordinate with each other by index, instead of looping and combining them manually you use iter = zip(list1, list2) and then you can do listname = list(iter) or setname = set(iter)
        - You can go from an iterable to their respective lists by doing list1, list2 = zip(*iter)
- String manipulation
    - You can reverse a string or list with stringname[::-1]
    - You get the last element of a list or string by using stringname[:-1]
- Sort
    - You can sort lists etc by a specific value using lambda notation listname.sort(key=lambda x: (x.start)) or to sort by two values listname.sort(key=lambda x: (x.start, x.end))
- Iterate a list you're removing from
    - Putting [:] at the end of the list name will fix this, but usually want to avoid this unless absolutely necessary
    - Better to just use a while loop where you have full control of when the loop ends
- Iterate python list in reverse - set the start of the range to the len(list) - 1, end of range to -1 (since its exclusive), and add a third parameter of -1 to decrement in the index of i by 1 each time
    - for i in range (len(listname) - 1, -1, -1):
- Python trivia/memory stuff
    - Python strings are immutable!

    - Doing list[i:k] is inclusive on the left and exclusive on the right!
        - Same as doing for i in range(start, end) it is inclusive on the right and exclusive on the right! That is to prevent index out of bound errors. Need to be cognizant of that
    - Saying a = somelist sets a as a reference to that list
    - Saying a = somelist[:] creates a shallow copy of that list (its a different list, but the elements inside of it are the same spot in memory
    - Saying a = somelist.copy() creates a full deep copy of the list
        - This is all why in a recursive function, the same value can be shared around because it can be updated at any point in the recursive stack in will inherently be global. Need to use this in those cases
    - For a set, using setname.copy() creates a shallow copy
- dict = collections.defaultdict(list) - hash table with a list by default
    - Never really had to use this
- Count = collections.Counter(listname) - hash table with counts
    - Use counter.most_common() to sort the counter in increasing order, this uses a heap behind the scenes
    - If you need least common, it would be better to just explicitly use a heapq (import heapq, heapq.push(list, tuple with occurencesfirst, value)
- Python's flaw for interviews is that it doesn't have a native TreeMap class (self-balancing sorted tree, like an AVL. this would be used to balance a bst) (O(logn)). Can just mention that and work around it
    - You want to use a TreeMap if you want to have quick retrievals but need the keys (or some values) to be sorted as well (eg if you need to find the next nearest or something)
        - You could make the hash map, then sort it. But need to compare that nlog(n) sort + n generation vs n * log(n) for each add. Also need to use TreeMap if you need the values to be sorted at each step instead of at the end (stream, etc)
    - A TreeMap's operations (including lookup) is log(n) by the nature of the tree structure
    - A TreeMap makes implementing a red-black tree relatively trivial
        - Red black tree is a self-balancing BST where the root is always black, and a red node cannot have red parent or red children
        - Height is log(n) and so are the operations
- Range vs xrange
    - Range creates a static list at run-time and returns that to iterate with
        - Want to use range if you'll be iterating through the same list multiple times because you won't have to generate each time
    - Xrange generates each value as you need it through a technique called yielding, and iterates using that
        - Want to use xrange if the list/range will be too long that it cannot fit in memory

- isalnum() - returns if a given character is alpha-numeric
- Binary
    - Use bin() to show the representation of any number in binary
        - Use yb = bin(y)[2:] to strip the 0b and then do yb = yb.zfill(8) to fill the front with 0's
            - Make sure to set the zfill to whatever the number restraints are. Doesn't hurt to use 32
    - You can go from a string of an integer in base10 to binary with int('100', 2)
    - Addition of two base10 integers: `c = bin(int(a,2) + int(b,2))`
        - int(string, base) allows you to say what base the string is in for your conversion to an int
    - Use int('0b1011000', 2) to go back to decimal
        - Do (bin) with these operators
            - x << y - shift x left by y positions
            - x >> y - right shift y times
            - x & y - bitwise &
            - x | y - bitwise or
            - ~x - complement of x
            - x ^ y - exclusive or