Resources List

Done around 10 array problems + 5 tricks problems

TO DO BEFORE MARCH 10:

Data Structures and Algorithms

- MIT's Introduction to Algorithms, with notes
- Read EPI with notes on:
    - Arrays
    - Stacks/Queues
    - Linked Lists
    - Hash Tables
    - Trees (+BST)
    - (some) Graphs
    - Heaps
    - Recursion
    - DP with DaC + Backtracking
    - Sorting Algorithms (Insertion, Quick, Heap, Merge)
    - Tricks: Two Pointers, Sliding Window, Bit Manipulation
- Implement 5-10 LeetCode easy's for each data structure/algorithm
- Do 25 mediums in interview environment

Systems Design

- Learn vocabulary (list of things to know on Tushar Roy's channel)
- ***bit.ly*** as training example (things to mention and structure of sysD)
- Uber
- Tinder
- Instagram
- Twitter

Projects

- CLI tool
- Workflow macro
- Finish Artale

MISCELLANEOUS NOTES [ALL PROBLEMS SOLVED WITH PYTHON 3]

when asked to do a problem replacing '.' in string with '[.]' (defanging IP address), instead of converting to array, iterating and replacing '.', you can use the built-in .replace(). note that this is also an example of when regex comes in useful (you can do a one-liner with that).

to convert an int to a str simply do str(n). remember that python is so powerful because you can pack lots of functionality into very few lines of code. converting an int, n, to a string is simply:

[int (digit) for digit in str(n)]
alternatively, you could use map(function, list), where the list can come in the form of a string. and since int(n) returns the integer version, int itself is a function, and so you can just do:
arr = map(int, str(n)) to obtain the same result

the python version of an arrow function in JS is a lambda function, declare by prefixing it with *lambda.* you an use these for the native reduce() of a list, or alternatively use reduce(operator.add/operator.mul, list) to reduce it too.

you can check if a string contains another string simply by doing *string1 in string2* to get a boolean —you don't need any .includes() like in JS or anything like that. this is why python is so beautiful and powerful.

when you're iterating through something, like a string, to see if there is symmetry in some way (like if the substring so far has an equal number of L and R, like in balanceString), then maintaining a *balance* or *symmetry* counter and incrementing another counter when the balance/sym counter is 0 (the substring so far is balanced) is a useful tactic, as opposed to a long approach counting the individual L and R in substrings.

enum() of an array (or string) gives you access to not only every individual element as if you'd split() it, but also a corresponding index to go with it. note that there is a proof that every recursive function can be written as an iterative one.

there are loads of useful array/string methods built into python whereby you can grab certain things using 1:3 or 1:4:2 (skipping every two) or 3: which goes from index 3 to the end, and so forth.

In place sorting does not necessarily mean use of constant space—you can recursively sort something "in place" (without explicitly creating another data structure) but you still need more space allocated in the recursive stack.

Important array functionality includes slicing with colons, copies, comprehension, range() for index-value matching.

For I in range() can be used creatively; with range(5,10) going 5, 6, 7, 8, 9, or from range(-2) to not iterate at all, or range(0, 7, 2) goes 0, 2, 4, 6—you always include the first value and exclude the second from the iteration.

You can check for a 0 in a list by its boolean false value

enumerate(start point, iterable) returns can object that has indeces coupled with the values (so for 'abc' as the iterable you get (0, 'a'), (1, 'b') etc. that can be useful in loops.

Important dictionary methods in python:

Lists and dicts are very similar, with the only difference being that you access dict values through keys as opposed to indeces. You can add a value by simply doing dictName['newKey'] = newValue, and delete simply by doing *del* dictName['keyName']

len(dict), dict.clear(), d.get(keyName)—instead of d['keyName'] because it doesn't raise an error if the key doesn't exist, and just returns None instead, d.items() returns a list of tuples, d.keys(), d.values(), d.pop(key) returns the value and removes the pair, d.popItem() removes a *random* key-value pair, d.update(d2) merges dicts

—> problems solved algorithmically but without implementation (go back and implement):

valid palindrome, merge sorted arrays, reverse string, missing number, pascal's triangle, magic grid, degrees of a sub-arrays

Data Structures and algorithms (LeetCode & EPI)

MIT introduction to algorithms notes (incomplete—only core, fundamental ideas)

1. Algorithmic Thinking, Peak Finding

Efficiency is important because scale changes with time. Decades ago, 1000s of users/data points was big data, now it's billions, and this trend will continue and so our solutions need to be as efficient as possible to this end.

In 1-D peak finding problem with an input array of 10M elements, the linear time brute force solution takes 13 seconds, and the logarithmic binary search method takes 0.001 seconds.

Imagine that at scale—efficiency analysis is not a pedantic measure and its not for elegance. It's very, very practical. Binary search is an example of a "divide and conquer" strategy (which is implemented recursively, always).

There's a difference between divide and conquer (a class of algorithms) and dynamic programming (a technique for thinking about/approaching problems). In divide and conquer, the sub-problems don't overlap. In DP, they do, and it's optimal to cache the overlaps via memorization to optimize.

Note that theta(n) is worst case, formally, but colloquially O(n). T(n) is the function that represents the number of steps a computation takes, and is set equal to the big-Oh notation when analyzing an algorithm.

Use of "recurrence relations" (T(n) notation) is a more airtight way of finding the asymptotic complexity of (recursive/DaC) algorithms. You look at the number of operations/steps taken after one recursive "step" and then feed in the end product of that into the next recursive step (like how DaC for 2D peak finding is T(n, m) = T(n, m/2) + O(n) where O(n) is the search of the 1D array to find the max before you compare it to it's horizontal neighbors and do binary search through those columns).

2. Models of Computation, Document Distance

Algorithms are mathematical objects, functions. Code is the computational analogue, in the form of code.

There are two important models of computation (ways of thinking about computers as mathematical constructs): random access machines (big array which is read from/written to, and computed on in between), and a pointer machine (which can dynamically allocate memory and operate on that memory) and manipulates objects which have pointers to connect to each other. You can implement theoretical computation either way.

There is a marked difference between abstract data structures like linked lists, hash tables, and data types in a programming language, like Python. For example, linked lists can be implemented via tuples or objects, and hash tables and implemented through dictionaries or objects, and stacks can be implemented using arrays or objects, and so you use abstract data structures to reason formally about problems and solve them efficiently in theory, and concrete data types to implement those solutions.

Lists in python are arrays, *not* linked lists. We know this because updating an index in a list, or even adding a new index, takes constant time. Linked lists are implemented through objects you create yourself, with a constant (reasonably small) number of attributes. Lists, when you append something to them, in python, take constant time, despite the fact that you'd think they have to copy all the elements and add another space. The reason for this is because they double table length's when they need more space, and therefore only have to do that rarely. Algorithms are so important because there are lot of features whose implementation, under the hood, that involve use of algorithms (table doubling makes lists function in the Python programming language).

You have to be careful when doing complexity analysis, especially with pattern matching. For example, the concatenation of two linked lists is a linear time operation, as you have to copy each element from each into a new linked list. It's true that addition is constant time if you're adding two numbers (or words, etc., which can be represented as numbers), but not so the case with linked lists.

X *in* L requires constant time (lists are necessarily sorted) and *len(*L) is constant because lists have an extra element storing list length in it, so you just look at that. Note how DS/A are used in the *implementation* of programming languages, and how learning those deepens your understanding of that which you type into an IDE.

Dictionaries are generalizations of lists in that the key is no longer just the index, but anything. They are implemented using hash tables.

Python *long*'s can't multiple using the grade school algorithm, as it takes too long, so they developed a faster version, which makes Python so useful for calculations/scientific programming.

Algorithmic improvements are not the only improvements that speed up runtimes of code. Also designing code in such a way that it doesn't re-use operations (uses helper methods), stores things so it doesn't recompute them, etc., improves real world runtime of code.

Document distance algorithms are diffing algos that are used to find how different two bodies of text are (used in cheating checkers, search engines for keyword matching, etc.). If you think of words or phrases in a document as vectors (with each dimension being a word in the document), you can mathematically compute how similar those vectors are (dot product) and use that to come up with a quantitative measure of similarity between two documents. So the overall diffing algorithm involves splitting doc into words, computing frequencies of words, and taking dot products with the vectors you get from the word frequencies (each word being a new dimension).

There are a lot of ways to implement an algorithm that diffs this way. The p-sets have 8 different implementations, taking all the way from 200s to 0.2s for diffing a 1MB text file. This shows how small changes in structuring an algorithm can make real differences to runtime on a computer.

2.5 Interlude—Efficiency Analysis of Algorithms (from MIT's Intro to CS)

Note that saying something is O(n) means that its growth is *bounded* by O(n), not that it really *is* O(n)—a technical detail.

There are multiple ways to do exponential, for example, each of which has a different asymptotic complexity. There's the iterative and recursive methods, both having linear runtimes. Another way is to halve the problem each time by returning (a*a)**(b/2) when b is even (and linear when it's odd, but that's overpowered by binary reduction) which reduces the problem of exponentiation by half each time. Alternatively, you can do repeated addition within two nested *for* loops, which gives quadratic time complexity.

Most proofs of algorithmic optimality are done by contradiction. You assume that a more optimal solution exists, and you show that the assumption leads to a nonsensical mathematical reality, thus implying your current solution is the most optimal.

The "towers of Hanoi" problem is a recursive one, where you have to move a stack of disks from one pole of three to another, with no bigger disk ever atop a smaller one. The beautiful solution to this is that you basically need to move all but the biggest one to the spare pole before moving the spare one to the end destination, and then move the rest back to the end destination. Therefore, the first part is moving all but the biggest one from one pole to another, which is solved recursively. The code for this seemingly complex problem is trivial—a few lines, which is the power of thinking of the problem recursively when you weren't before.

```
def towers(num_rings, fromStack, toStack, spareStack):
    if num_rings == 1:
        print("move disk from " + fromStack + " to " + toStack)
    else:
        towers(num_rings - 1, fromStack, spareStack, toStack)
        towers(1, fromStack, toStack, spareStack)
        towers(num_rings -1, spareStack, toStack, fromStack)
```

If you write out a recurrence relation, you find that each instance of the problem generates two sub-problems (moving the n-1 disks to the spare pole, and then to the end pole after the big disk has been put onto the end pole), which means that for each recursive call, 2 more problems are generated, giving it an exponential runtime. For emphasis, if your processor did a *billion* operations a second, for n=1000 disks a logarithmic, or even quadratic solution, would feel instantaneous, but this exponential algorithmic would take 10^{284} years. Such is the importance of asymptotic analysis and careful algorithmic design. Sure, computers are fast and cheap, but problems grow fast *way* quicker than computers can.

Analysis of implemented algorithms depends on how certain features are implemented. Access of a list may be constant time in Python, but not so in other programming languages if they implement arrays using linked lists, which could mean that actual complexity varies, and so you have to clearly define what you mean to be a primitive/constant time step in your pseudocode.

3. Insertion & Merge Sort

We care about sorting not for its own direct use (which is plentiful), but also because many problems involving large data sets become trivial if the data is sorted, and so efficient sort lends itself to easy *and* efficient solutions to problems. It's also used as a sub-routine in data compression and computer graphics, amongst other things (document distance involves sorting to find word frequencies in a dict and is a form of compressing that written data), and computer graphics needs you to sort layers of pixels from front to back so that you don't waste resources displaying something that would be behind an opaque pixel.

Choice of algorithm is deeply contextual, depending on how much space complexity you can tolerate vs how much time matters to you, whether you're expecting best/average case input, and things like that—there is no "best" sorting algorithm.

There is a difference between O(n) and theta(n) and omega(n), where O(n) is the *upper* bound (worst case runtime), theta(n) is average/expected case, where it's displaying the upper *and* lower bounds of a function. Note that, colloquially, O(n) is basically used for everything, often incorrectly, even in industry. O(n) is the most common anyways because really care most often about worst-case performance.

Insertion sort is basically pairwise comparison, with swaps in the process. This runs in quadratic time because the pointer moves across each element in the array, potentially having to make *n* swaps to get it to the other end of the array. But in implementation, often comparisons are more computational steps than swaps, and so we want to minimize that. We can do this using binary search to get *logn* comparisons instead of *n* comparisons (that doesn't change the number of swaps we have to make, however, which is still quadratic). It's called insertion because pairwise swapping at scale basically looks like taking each number and inserting it at it's appropriate position. The flavor of insertion sort that makes comparisons using binary search is termed binary insertion sort.

In merge sort, the magic happens when you merge the base cases together, that's when the sorting actually takes place. When merging, think of it using two fingers as a means of making comparisons that form the basis of the sorted list at the end. Each time in each "level" (of which you have *logn*), you have to make *n* comparisons by virtue of having each "finger" or "pointer" point to one number exactly once (no doubling back, like in insertion sort), giving an overall worst case complexity of O(nlogn). The space complexity is O(n) because you're only expanding out one parallel line of the tree at any one given point, not all of them (which would give O(nlogn) additional space).

Merge sort in python has a speed of $2.2n\lg(n)$ microseconds, and insertion sort is $0.2n^2$ microseconds in python, and less than a tenth of that in C.

You can draw recursion trees, where you take the amount of work done at each step as the root, and then split it based on the amount that the recurrence shortens the problem, then find the dominant factor on each horizontal row for the amount of work done, and that's the overall amount of steps taken, or time complexity of the recursive algorithm.

4. Heaps and Heap Sort

—> study queues via MIT/yt/EPI/leetcode 5 before starting this lecture, then finish this, read EPI sections on heaps and sorting before doing all heaps/sorting leetcodes

Data Structures Self-Study

In general, know that space complexity is the *additional* space required by an algorithm.

Stacks

They are literally the same as a list, but just call it by a different name because you do different operations on it (push/pop only vs all list methods) and think of it differently on a conceptual level. Browsing session is stored on a stack in local storage so you can go back to the previous page. The list is the physical implementation of a stack, which is an abstract conceptual idea.
Use .append() and .pop(), both native functions in python.

Queues

An "ADT" is an abstract data type, where you don't specify any implementation details.

A queue is used when there's some form of shared resource that can only handle one request at a time, like a printer on a shared network. I imagine that this would also be used in some shape or form (but based on priority) with CPU scheduling, too.

You can implement a queue in two ways, as an array or linked list. There are, as with everything, tradeoffs involved. For example, while a linked lists can expand indefinitely, there's also less of the space it occupies used for the actual storage of data. We get around the linear traversal time of arrays (enQ, dQ need to be constant time with this data structure) by adding a rear pointer for where we enQ things so we don't have to traverse the whole list. You often add isEmpty and Peek() helper functions as part of the implementation.

In an array based implementation, you simply enQ or dQ things to the rear/front, and when it gets full return that the queue is full, or expand the array so there is more space. To get around the problem of filling up part of the array but not being able to get the rear pointer to the rest which isn't full, because it comes numerically before the front pointer, you can use a "circular" array whereby you insert at the (rear+1)%N position. You should always remember to check for the special cases where the front pointer == rear pointer or when they are both 0/-1 to signal that the array is empty.

A double ended queue is also called a dequeue, much like the operation to remove an item from the front of the queue.

Trees

nodes, edges (pointers, parent, child, siblings, ancestor/descendent (generalised), grandparents, uncle, root, leaves, height/depth.

important to note that trees are recursive data structures, where you have a node, and then several sub trees attached to it. they are unordered, so any numbering system you pick to order the nodes is completely arbitrary and just for clarity.

a binary tree is a tree where each node has <= 2 connections originating from it. this is the simplest and most common type, but more complicated trees with lots of edges originating from each node also exist and are used.

naturally stores hierarchical data, as opposed to linear data like in a array/LL/stack/queue, optimised for quick insertion/deletion, and, especially, search. specific types (trie for example) are used for other things, like dictionaries and network routing algorithms.

Binary Trees

note that by definitions given, a single node is a binary tree, a line of nodes is a binary tree (just with null pointers instead of one child).

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

A full/proper/perfect 2-tree is one that has all branches full.

For a binary tree to be most effective as a data structure, it needs to be as close to balanced as possible so that retrieval of data can be quick. In other words, this minimises the height of the tree, which is proportional to the time taken to search the tree, since binary search goes vertically down the tree. The formal definition of a balanced tree is one where the difference in height between the left and right tree (remember the recursive definition of a 2-tree) is =<1 *for every sub-tree in the recursion*.

you can implement a binary tree using objects, like you'll see in the C++ implementation section, but also using arrays, where you can just take the elements and stick them in the array, giving a formula for how you go from element *n* to element *n + 1* when the tree is labelled with numbers *n* (adhering to a certain convention since it's labelled arbitrarily).

BSTs

Unsorted arrays and linked lists have linear time search and removal, and constant time insertion. Sorted arrays shift the problem, giving logarithmic time search, only to increase time taken for insertion to linear. We want a data structure that keeps the logarithmic search time, but also gives a similar logarithmic insertion and removal time.

defined as a 2-tree where the value of all the elements in the left child of a node are strictly less than or equal to the value of the node, and opposite for right. thus, there is some order to the values stored in the nodes (not the nodes themselves). this naturally lends itself to binary search. but now that there is some order to the data, insertion and removal (which require searching) are also logarithmic time.

implementation details (python)

(practise implementing from scratch and make notes here)

memory details

interesting to note that I forgot most of what I learned in cs50, even though all of this was covered there, and haven't had to use it until now (and even that is just for interview prep as opposed to

actual work). that said, re-learning it has been very fast, which is probably the main benefit. so you'd want to optimise for courses where you're most likely to need to re-learn the knowledge in the future, courses that teach you a new way of thinking (first few economics courses for example), or courses with knowledge you're actually going to use, courses that will inspire actionable change in your life as you're taking them, or courses that are very specifically pointed towards educating you towards a problem you're trying to work on in the immediate future.

stack & heap used to be physically separately wired parts of a RAM chip on the motherboard of a computer. much like how cloud computing just squeezes servers in wherever there's space, now as technology has advanced, the space for variables and static memory (stack) can be anywhere in RAM (or even in hard drive virtual RAM) and, equally, the space for specific instances/objects that have been dynamically allocated (heap) can also be everywhere (not spatially separated by much on the metal).

when you have multiple functions in a program, they pile up on the stack as only one can run at a time, before returning its value to the one in the "stack frame" below it, so that can continue running.

the reason that these exist in the first place is because we need to store variables and objects in memory for a program to compute them, since only one computation can happen at a time on a processor. both abstract memory structures have certain upsides and downsides, so combined use can negate both downsides. a stack is used because adding and removing (pushing/popping) variables from it is very mechanically easy to do. but by virtue of making a structure with pushing/popping as easy insertion/deletion, you lose the ability to manipulate things deep inside the stack. and obviously it's limited if you're pushing/popping off the top of it. conversely, the heap can store more stuff and you can access it easily (as opposed to only the top of the stack), but it also comes with more overhead (you have to track the chunks of metal that are free, where they are, how big they are, and more—all done by the OS). *the size of the heap varies*. this "heap" in dynamic memory allocation has nothing to do with data structure, whereas stack does.

C++ is a superset of C—it is backwards compatible (you can use the backwards functionality like malloc() and so it's compatible, but the point is you have better functionality are your disposal).

always remember that all of the tree details on the whiteboard are really memory locations changing values on a metallic level. you have the stack, where all the function calls for Insert(), recursions, and getNewNodes are done, storing the value of local variables in that process in an increasingly high stack frame series, but then you have the heap which is where the variables in each stack frame are pointing, that actually contains the nodes and values that comprise the tree. this implementation on the memory level is often abstracted away as you don't have data types and pointers in Python, but is important to know to fully understand how these data structures work.

height + 40mins to stopwatch

remember that this is defined as the number of edges from too to leaf (max). depth, conversely, is defined as distance from node, so depth + height of any given node = height of tree (of root). conventionally, as well, the height of an empty tree is -1.

binary tree traversal (visiting all the nodes *exactly once*)

due to it's non-linear nature, tree traversal is definitively non-trivial.

Note that trees are special cases of graphs, and BFS/DFS are two major categories of algorithms used for traversal.

BFS = visiting all the children of a node before going to its grandchildren, horizontal scan

DFS = visiting all the descendants of a child before going onto its next child, vertical scan

important BFS if level-order search, scanning horizontally through each level, and there are three main DFS strategies, referring to when the root node (of the current tree in the recursion) is processed relative to its left and right children—preorder is DLR because D (data) in the root node of the current sub-tree in the recursion comes before visiting its left and right children, and then you have LDR and LRD which are Inorder and Postorder, respectively. Inorder returns a sorted array because of the structure of a BST.

Level order traversal (BFS)

How do we implement this given that you can't go back up a node, how do you go to its brother right after visiting it? You use a queue and store the pointers to all a node's children before visiting that node. Then you go to its left child (and enqueue all of *its* children) and then its right child (and all of *its* children) before visiting the left child's children.

Since each node is visited at least once, and at most three times, it runs in linear time since each visit requires only a constant number of enqueueing/processing operations. And wrt space complexity, this has to do with how many things are ensued at a given time since those pointers are what take up memory in this case. In the best case, a linked list 2-tree (straight line of nodes), it's constant time since there's only ever one child in queue. In the worst case, there can be a perfect binary tree, which will have linear time since all of the leaf nodes will be enqeueud at some point (how else would they be visited) and that's n/2 of total nodes.

pre, in, post-order traversal analysis (DFS)
the implementations for all of these are very similar just with a bit of shuffling to move the D (processing) around in order. the memory used here is not extra memory in the form of pointers in a Q, but instead a huge call stack that is proportional to the height of the tree (since it's a DFS algorithm). and therefore the space complexity in the worst case (linked list where n = h) is linear, and logarithmic in the best case, where n >> h. time complexity is linear because there was one function call with one or a couple of manipulations for each node.

implementing binary trees in python

because python lacks pointer functionality, you have to fundamentally rethink how you define trees in python, as you can't think in granular memory terms with addresses anymore, since python is weakly typed also. instead, just take the common-sense approach of defining the root of a tree to be an actual node (as opposed to the *pointer* to a node) and instantiate a Node object as the root when defining the BST class.

when the user creates a bst, they pass in a value and you simply set self.root = Node(root) and voila, you've created a root node with that value inside it. the root itself is thus a node (object, not a value), but you can get at its value simply by appending .value. as a consequence of the fact that the root is a node, that means it also has left/right properties, and you can add things by simply doing tree.root.left = Node(5) without thinking about pointers or memory at all.

you can thus, using this method, fill in the tree via commands like tree.left.right.right = Node(4)

traversal method looks quite literally this simple:

```
        def inorder(self, start, string):
    if start:
        string = self.inorder(start.left, string)
        string += (str(start.value) + "-")
        string = self.inorder(start.right, string)
    return string
```

these traversal algorithms are not specific to BSTs, and instead just traverse any binary tree.

A useful string method native to python3 is string.count('a') which will return how many a's there are in the given string.

General python syntax to know

To check if a value is in a dict, look for char in dict, or char in dict.values().

.isalnum() is a useful function that returns a boolean telling you whether a character in a string is alphanumeric or not (for example when making comparisons)

sum(list), .isnumeric(string) and int(string), .lstrip([charToRemoveFromLeft])

.join() is a very important python function, used for strings, and it returns a string. You'd use it like '-'.join(list) where you want the elements of the list to be turned into a string joined by a hyphen—often you can use this to convert a list to a string by having an empty join.

findall() from re (regex module) takes two arguments, the pattern you want to search for non-overlapping matches in (in regex) and the actual string you want to query.

5 misc. done before, 10 easy EQ on JS site, 5 stack Qs done, 5 array done, doing 10 more sliding window+two pointer+array today
The .sort() method is implemented using timsort, a hybrid of merge sort and insertion sort, with a worst case complexity of nlogn and best case of O(n).

arrays

list(), map() takes you from one set of values to another, especially useful with lambda functions used as lambda x: 0 if x == 1 else 1 for very pythonic, concise solutions. .reverse() reverses the list in place whereas list[::-1] returns a copy that is reversed so it depends on what programming paradigm/style you're abiding by.

recursion is typically slower than iteration

Common problem solving tricks to know

two pointers

You can often brute force problems without this, but use of this technique usually decreases time complexity to linear time since you're making *n* comparisons between the pointers.

sliding window

commonly used for questions like find the maximum sum of a subarray of given length within a parent array, where you iterate through all the subarrays of that length (subarray being the window, that slides incrementally to the right through the parent array). But to make this more optimal, you stop re-summing the window each time, and instead just remove the first element from the windowSum and add the new element, reducing time complexity by some large constant amount. You have to initialize the windowSum in the beginning to optimize the solution because then you can edit the window sum in each window slide by adding a new element and removing one at the beginning.

bit manipulation

rest upon bitwise operators, ~ & | ^ which convert inputs to binary and then back to input format. There are certain patterns and rules for these bitwise operations that makes their behavior useful for certain classes of problems. eg—for a question asking you to find the missing number in an unsorted list, you could sum the numbers and compare that to theoretical sum, or, more elegantly, use the XOR operator to compare *n*, the number of elements in the list, to the index-value pairs. Since XOR is associative, *n* cancels itself in the list (which must necessarily be there) and all the indices cancel their respective values, except for the index of the missing number.

questions on bit manipulation test conceptual understanding of what happens in computers, implemented at the lowest level—this is how addition, multiplication are implemented in embedded, resource-constrained systems and form the foundation of architecture/CS fundamentals. addition, for example, would simply be:

```
while b != 0:
    a, b = a ^ b, (a & b) << 1
return a
```

These are often powerful tools not just because they allow you to solve specific problems, but because those problems have otherwise very long solutions whereas the time complexity of these solutions is usually constant, if not linear.

You can also solve the problem of recognizing whether a number is power of two in *linear* time (as opposed to the obvious reductionist logarithmic solution) by using the complement and & bitwise operators. Two's complement is just a function that represents the negative version of a binary number. If you have a three-bit number 010, which is 2, then you find it's distance from 2^(bits) = 8, which is 8-2=6, or 110, which is meant to represent -2 in this situation. Note that the two's complement is the same as inverting digits and adding one.

x = 0010000 (power of two)

Not x = 1101111 + 1 to make two's complement = 1110000 and therefore x & -x isolates the right most 1 in the chain, which will mean the result of the computation is equal to x itself if x is a power of two (it only has one 1 in the chain, the right most one) and hence making that comparison is a constant number of steps and therefore O(1)

Dynamic programming

Extension of recursion that further optimists time complexity by storing intermediate values of recursion so they don't have to be re-computed. The array/structure that stores these values is called a memo, and the process of storing them memoization. A "bottom up" approach uses "tabulation" instead of "memoisation". Memoization is "top-down" because, like in the fib(n) example, you compute fib(5) before looking for fib(4) which is needed to compute fib(5), whereas a bottom-up approach finds fib(4) and puts it in a variable/array before using it to computer fib(5) at all.

Tabulation is more efficient when all sub-problems must be solved at least once to get the final result (as in the fib(n) problem) but memoization is faster otherwise because it strictly only computers things that need to be computed. The key insight is that we're trading space for time—by storing partial computations in additional space, we can often reduce problems that naively take exponential time into problems taking linear or quadratic time, a huge improvement. And these improvements are very real—try computing fibonacci numbers using the naive solution and fib(1000) takes literally hours & recursive stack often is too large, causing a crash. Whereas with DP/bottom-up, you get it instantly. Imagine having to make that computation thousands of times a second, like RenTec do, and you see the power of optimisation.

You have to understand how to think in terms of DP and breaking a problem down into smaller sub-problems of the same structure. For example, if asked how many ways you can get the number 5 by adding 1, 3, 4, the answer is 6, and you could get that by thinking how many ways could I represent 4 using those numbers then adding one, or represent 2, then add 3 to each of those solutions, or represent 1 and add 4, then add all those distinct ways together.

Systems Design

Designing gym app for advith:

think about prototypical user—"user stories" approach is putting yourself in the shoes of the consumer and thinking of features you have to make that way, as opposed to thinking of it as an engineer in a sporadic manner.

important to interview users, consider both who they are and what their pain points really are before taking a shotgun approach to what you want in the app—engineering second, solution first.

SysD is important in general because the role of a CTO at a startup is to think of the architecture and actually building it out as well as overseeing engineers who are building the endpoints, as well as constantly evaluating design decisions and architecture to see if it's still the best decision in light of all the increasing amount of information you have about the consumer. Later on, Samsara has around 2,000 employees, but the CTO still writes code on critical systems. He has not much say in long term vision of company or which problems they are solving, or not even really the high level of how they're solving them, but really just implementation of those solutions that the CEO/product teams come up with.

Interesting to note that Samsara is sales-heavy because they're business facing, and so it doesn't really spread as much through word-of-mouth, and you really have to build a relationship with enterprises to convey what you're selling and get them to trust you because it's a high risk decision for them to go with your solution. They have around 5x as many sales people as engineers.

In the valley, most engineers just see work as a 9 to 5. The small fraction that are interested in startups are not *aggressively* pursuing ideation, just passively reading and reflecting slowly plodding along. Many of *these* ideas are low-tech solutions like a Barry's bootcamp for India, a booking web app for venues looking for musicians, automating the recording of senior citizens' narratives.

Most software engineers spend a significant chunk of their time rebuilding features that already exist from scratch. When Advith is trying to build a gym app and is thinking of features (booking a class, rewards points, news feed), all of those individual components already exist, but there won't be any quality templates, and even if they are it takes enough time to understand and optimize how the code in that system works that you might as well write it yourself because you need to have a solid understanding of how the code works to extend its functionality. And therefore software engineers spend lots of time separately rebuilding the same things other people are but in their own slightly different way (which requires you to build the *whole* thing yourself, not just take a copy of someone else's thing and tweak it slightly).

Important concepts:

You must practice several examples, understanding the structure of the SySD question—problems and questions to address, below are all important concepts that you'll have to touch on when making design decisions in context of the problem (which database—with sharding? How about caching? How will you interface with 3rd party APIs? Will this be secure? etc.)

CAP theorem

TCS fact that asserts that out of consistency (data being the same across all nodes in a distributed system—get the latest data or no data at all), availability (returning some data even if it's not the latest), and partition tolerance (can still return data when some nodes are down), you can only have two of the three. Realistically, you can only choose between C/A because if a system is partition intolerant and a server goes down, it all goes to shit.

SQL vs NoSQL

ACID = atomicity (transaction either fails or succeeds as a whole—no partial success), consistency (when a transaction completes the DB is structurally sound), isolation (transactions don't conflict because they are monitored by the database, making it seem like sequential changes are made), durability (result of a transaction is permanent, even if it's a failure). Basically this is really nice for a developer because ACID databases are write consistent (means something different to the CAP theorem here) but that requires sophisticated locking protocols under the hood, making is slower, especially at large scale.
BASE = basic availability (data is available most of the time), soft-state (system doesn't have to be write consistent all the time), eventual consistency (changes are not immediately reflected, but eventually will be).

SQL is used for relational databases, characterized by their tables and rows, and facilitates complex read, write operations like JOIN and CASCADING DELETE. Typically vertically scalable (adding more memory/processing power). Use this any time you have complex queries or you need high reliability or fixed schema.

NoSQL stores data in JSON or XML (basically a version of HTML used for data storage), and schemas are dynamic so categories (eq of 'rows' can be created on the fly, stored as graphs or key-value pairs), usually horizontally scaled (add more servers and get them to sync up with each other). Use these where schema are likely to change or you aren't going to have complex queries or you need speed at scale.

Database Sharding

Basically horizontal partitioning a database table so that you can distribute it across multiple nodes in a distributed network. Improves query times and makes database more resilient in case some servers are down, but requires more complex infrastructure with more moving parts to get the shards to communicate that often has to be built out at the application level (though some databases do have it as a feature). Physical shards are the nodes in the distributed server system, and logical shards are the split up rows of the DB/table. There are different strategies to distribute data across shards (structurally similar to load balancing)

Key based sharding takes some column of a DB and puts the values of each row through a hash function to decide which bucket (physical shard) that row should go to, kind of like storing data in a hash map. Pros are that it's deterministic and automated, cons are that adding a new server requires effort because you have to redistribute evenly distributed shards. There are also other methods like simply taking one value of the table (prices of items) and sharding based on variation there (range-based) but that doesn't guarantee they're distributed evenly. Sharding is usually only important if you are or anticipate operating at large scale of requests/data.
Other ways to optimize database performance include setting up a remote DB, caching (for poor read performance)

Locking (optimistic, pessimistic)

Essentially solves problem of concurrent access to database. Pessimistic is easier to implement as you just lock out all other users when one is modifying the table (eg flight seats). In optimistic, you still allow table modifications in parallel—just compare version of change submitted to current

version in table; if they don't match, someone else changed data before you committed code and so you have to go back and make your change to the DB again *with the new data* to avoid corrupting code. Optimistic is default (and requires manual intervention), pessimistic is easier to implement.

Caching

Redis/Memcached are basically cache services on someone else's hardware. They store your data that is read often in their RAM for fast read access, so you don't have to check your disk on every operation. Types of Cache include:

Application Server Cache

Literally on back end of web server—can lead to cache mismatch in a system with multiple servers.

Distribute Cache

Cache divvied up between nodes in a server, where parts of cache are assigned to nodes based on consistent hashing function (you can easily trace where a certain part went since the caching function is deterministic). A short aside:

Consistent hashing: a method of uniformly distributing data to different servers for storage that is *independent of the number of servers*. Thus, it doesn't break apart in terms of uniformity when servers are added or taken away, a very useful real-world implication. It thinks of hash tables as rings/circles, with servers and data points being nodes dotted along them, where you simply assign a data point to its (say) closest clockwise neighbor. The advantage is that if one server is removed, its data points need to be assigned to new servers, sure, but all others remain untouched and uniformity maintained.

Global Cache: single cache for all servers to tap into .

CDN: distributing static media across several servers geographically spaced for ease of download.

Note that when data is modified in DB, there should be protocols in place to make sure it's automatically invalidated in the cache. There are several ways to modify things in cache, you have to make design decisions about whether your system needs LIFO, FIFO, LRU, least frequently used, etc.

Load Balancers

These can be hardware (custom build units, expensive but performant) or software (just a server repurposed to load balance incoming requests). And it can balance request loads at any point in the stack—to the front end, back end, or backend making requests to the DB. Most companies (startups) use cheap/open source software and repurpose a server to act as the load balancer. Popular load balancing algorithms include round robin (can be weighted), hashing the source IP/ destination URL, or just to the node that has the least response time.

Data centres

These are just thousands of processors taking in, processing, and distributing data (hence the networking algorithms must be very efficient because they are all constantly sharing load and talking to each other). It's interesting to see the scale of these centers, some miles across, with millions of processors serving requests for billions of people. I shudder to think how complex and difficult it must be to get all these to networks seamlessly and reliably with each other. Networking and devOps engineers do some really nitty-gritty, complex work. It's also important to note that at no point will I have to know or understand how the processor actually works inside, as I'll never be trying to build a better one or even really optimize for performance on it.

One of the most popular models/frameworks used to distribute data to clusters efficiently is MapReduce, made by Google and popularized by an open source software with the same name (by Hadoop). The premise is about mapping data points to the correct nodes where they can be

processed, then reducing all the outputs of the nodes into one output of the entire computation. The classical example is counting word amounts in a set of documents. Map() distributes documents across nodes, each independently counting the occurrences of CAT and DOG in their set of documents. The hidden “shuffle” phase prepares the system to computer reduction, putting all the key-value pairs (CAT: 21, CAT: 14) from the different servers in the map stage so they can be reduced to give one final output for the system.

There are also other considerations in designing and running a data centre—from temperature regulation to physical security to automating alerts for employees on duty in case something goes wrong. Racks of servers (vertical stacks) are arranged in long rows that employees often navigate using scooter or bicycle.

CPU, Memory, Hard Drive, Network Bandwidth—always think about how much of each of these we have in the system when designing a system and making design decisions. EG—Transcend might not want to buy more memory because that’s hella expensive, but may have enormous hard drive stores and CPU power because they store data indefinitely, so re-architecting a system to take advantage of this makes it more easily integrable with existing infrastructure.

Random vs sequential i/o on disk—know that random i/o is an OOM slower because you have to find the slot and then read/write one chunk of data then repeat, with the search taking more time and the i/o, whereas with a contiguous block of memory you just seek once and then i/o. This is why there’s often a large performance difference of database hard disks/their associated software on the package vs in production.

Http/2 vs websockets

Know that HTTP is just a set of rules that computer networks use to move data efficiently, much like a set of rules you obey when ordering at a restaurant, none of which you have to be told to obey every time, they are just programmed into you. HTTPS just encrypts data, and that’s done via SSL/TLS which are protocols used to implement this encryption.
Problem with http is that it was invented when the internet was very young—when data transmitted between client and server was really just markdown. Now, to handle the growing complexity of web pages, there have to be numerous changes to how computer handle, process, and communicate data to reduce latency with the original http. If both browser and server support h2, then they package and send data across the network differently (for optimization of complex data) than they did before. Examples of improvements made by the new protocol include multiplexing data (receiving several chunks of data through one connection), sending data in binary instead of cleartext, optimizing the movement of header information to reduce redundancy, and more. H2 is fully backwards compatible with previous versions, and does away with some of the workaround h1 used to get past problems it had.

Simplex is unidirectional over one channel, half-duplex is unidirectional at any one given time, and full duplex communication is bi-directional at the same time.

Know the meanings and relationships between IP/TCP/HTTP/2/WebSockets/SSL/TLS/DNS/DHCP

IP
HTTP/2
WebSockets
SSL
TLS
DNS
DHS

Transcend notes:

Data brokers are those who sell information about you to other enterprises so they can issue better ads. They collect this from public domain, social media, and customer databases, combining this with algorithms that output your likely interests, personality, etc., to businesses, who then issue scarily well targeted ads. This is all legal.

Transcend helps you "take back control of your data" by choosing which sites track online and offline data about your use (and sell it to brokers, etc.) Opting out otherwise is very difficult because it's layered within the fine print, and even if you opt out of one broker, others trade with each other so they still have access to your data. It's business facing, and when a user wants to download/delete all their personal data from a company (and everywhere else it's littered around, like with other companies they use like Stripe)—it deletes everything cleanly, making it easy for the company to be compliant with increasing amounts of regulations, a "turn-key" solution, of sorts.

All big companies use data brokers to learn more about their consumers.

Questions to ask:

how do you hire as a small startup—what are technical interviews like and what is timeline

what are the tangible steps from idea to having made hires and gotten funding (50K causing them to move out)

why did Accel fund you after you'd been rejected so many times

what about your approach to investors initially was "naive" as you put it

business model tied to regulations, so if privacy restrictions loosen up less customers

TLDR—privacy dashboards for GDPR compliance.

—> timeline is fast because they are small they are very agile. He said if I'm free in the next couple of weeks we should do an onsite ASAP, so they can set up an interview in days. Which means I can reach out after I've done more prep (reach out *after* the onsite with Transcend, knowing what to change in my preparation).

Onsite prep plan (1 month)

- ask everyone I know about how small startups (10 people) do technicals
- start banging LC easies at a high rate, consulting EPI and YouTube
- go through half to one video for Artale a day
- ask roomies for in depth rundown on how each company interviewed, and how it varies for smaller companies
- watch walkthroughs done right on YouTube
- find 4 other elite companies that are on exit path with very strong founders to interview with and set up connections to those so I can apply after my onsite with transcend
- understand Transcend's business model and backend architecture to think about what kind of tasks they might need me for and therefore

—> till Feb 8, 2020:

—> no social media during day

—> no thinking of cold approach/pickup, simply straight compliment girls on the street and strike up conversation in warm settings, even if you don't escalate—no hesitation

—>

How to prepare for an onsite notes and things to know

An onsite does not mean 1 interview. It means several interviews, up to 1 hour each.

Likely be given white boarding challenge

You could be asked:

- what's important to you in a company
- what culture are you looking for
- debugging interview, behavior check (understand what they're looking for), project-based interview (design decisions—practise in front of roomies presenting a project), usually *will* still have a DS/A question, pair programming, brainteaser
- brush up with Javascript & React since I've mentioned my use & that's their stack
- they ask questions based on your background—they will ask me questions knowing my own background that I have raw intelligence and worked on some cool projects
- much like the Cambridge interview
- be ready to explain projects that failed
-
- understand stitching APIs, encryption and really hash out what they do and how they work

—> okay to ask what interview process is like

tips on the day

- refer to things unambiguously and formally
- always make comments on how you'd improve time and space complexity
- always refer to how the problem you're solving would interface with larger infrastructure/ other parts of the system
- gotta get through several questions, so pack in lots of insight *fast*
- don't hand wave, actually code things out
- come in with suggestions for improvements
- have a very clear and researched answer to both why you want this job and what you're going to contribute going forwards
- have examples of times you demonstrated the 2-3 qualities you're most championing

—> bring up 2 technical and 2 personal stories about how you work well

Startups tend to ask questions aimed towards building or debugging code. ("Write a function that takes two rectangles and figures out if they overlap."). They'll care more about progress than perfection.Having a technical interview cheat sheet, reading books like "Cracking the Coding Interview," and doing online coding challenges on sites like Interview Cake are all amazing, but if you want to take it even further, live coding interview prep might be what you need to really get comfortable and polished."Pramp.com is an excellent place for getting realistic coding interview practice–and it's free!" says CEO Refael. "While solving coding problems can help you improve your technical abilities, mock interviews with peers can also enhance your soft skills: the way you communicate, your body language, etc."

Think in terms of vocal they used—encryption gateways, integrations, end to end encryption, and more.

Coding

1. Find word in a 2D matrix
2. Top K frequent words in a list
3. Find median for a stream of integers

4. LRU cache

5. Least Common Ancestor in n-ary tree

6. Closest element in BST

7. Print numbers in sequence using 2 threads - one for odd and one for even

CS + Programming Language fundamentals (My language of choice was Java)

1. How are indexes implemented in a SQL database? How do reads/ writes work with indexes? (This was the most common question asked across all the companies)

2. What is the `volatile` keyword in Java?

3. What are the different transaction isolation levels in a SQL database? (Followed by probing questions on some of them)

4. How does a `ConcurrentHashMap` work in Java?

5. Explain CAP theorem and eventual consistency.

6. Garbage collection

System Design

1. Design a pub-sub system without persistence

2. Design a URL shortener

3. Design a email/messaging system

4. Design Quora feed page

5. Design a system for processing jobs(whose information is stored in a database) in an exactly-once fashion

6. Design a Youtube clone

7. Design a Json Parser

8. Design a data pipeline for a Machine Learning system

9. Design a multi-level cache

Behavioral/Managerial Rounds

1. What is the most difficult technical challenge you've solved?

2. Tell me about a recent project that you are proud of. Why are you proud about it?

3. How do you mentor other engineers in your team?

4. A time when a deliverable got delayed - 1. not because of your fault (eg. requirements changing) 2. because of your fault - How did you handle it? How did you communicate this to impacted teams? What did you do to ensure it doesn't repeat?

5. What challenges are you looking for, in your next company?

6. Tell me something that's not on your resume.

API is just an endpoint—an abstraction of a particular function. An SDK is a set of code that is used to initiate applications on a particular platform—the iPhone SDK is the set of code involved in writing the backbone for any iOS app, for example, and other companies like Facebook (which allows people to make games on their platform) also release SDKs.