Technical Interview Study Guide

I – General

- **To do**:
    - Repeat the problem out loud, slowly but confidently.
    - Ask questions, check assumptions, know what the *constraints* are.
    - Talk out the problem!
    - Work out examples on the whiteboard, devising some kind of brute-force algorithm along the way (*Don't stay silent here).*
    - Start with **naive** brute force solution.
    - Check edge cases and do null checks (!) especially.
    - Work out examples using the code/algorithm; keep track of state by hand.
    - Know time/space complexity: http://bigocheatsheet.com/
- **Heuristics**:
    - Always consider **hash tables** (dictionaries) with their O(1)-ness.
        - "Tip: using a dictionary is the most common way to get from a brute force approach to something more clever. It should always be your first thought."
    - If at all array-related, try sorting first.
    - If search-related, consider binary search.
    - Start with a brute force solution, look for repeat work in that solution, and modify it to only do that work once.
    - **Space-time trade-off**! That is, for better time complexity, try using auxiliary data structures. E.g., do something in a single pass over an array—O(N) time—by using a hash table—O(N) space—vs. doing something in multiple passes—O(N^2)—without using any extra space—O(1).
        - What information can I store to save time?
        - Another example: O(1) get_max method for a Stack class stores extra information (the max at and below each element) to save time (instead of iterating through the stack O(N))
    - Try a **greedy** solution:
        - Iterate through the problem space taking the optimal solution "so far" until the end.

        - Optimal if the problem has "optimal substructure," which means stitching together optimal solutions to sub-problems yields an optimal solution.
    - Remember that I can use **two pointers** (e.g., to get the midpoint by having one pointer go twice as fast, or in a sum problem by having the pointers work inward from either end, or to test if a string is a palindrome).
    - If the problem involves parsing or tree/graph traversal (or reversal in some way), consider using a stack.
    - Does solving the problem for size (N – 1) make solving it for size N any easier? If so, try to solve recursively and/or with **dynamic programming**.
        - Using the max/min function can help a lot in recursive or dynamic programming problems.
    - A lot of problems can be treated as **graph problems** and/or use breadth-first or depth-first traversal.
    - If you have a lot of strings, try putting them in a prefix tree / trie.
    - Any time you repeatedly have to take the min or max of a dynamic collection, think **heaps**. (If you don't need to insert random elements, prefer a sorted array.)

II – Bitwise Operations

- Integers are represented as a string of bits (binary digits). Either big endian (akin to most significant place value at left) or little endian (akin to most significant place value at right). So three-hundred twenty-one as 321 or 123.
- If **unsigned integer**, the number of bits used (the width) determines the numbers that can be encoded: 2^n. (Unsigned means that the number is always positive; negatives aren't supported)
- If **signed integer**, 4 methods; most common is 2's complement where negating a number (whether positive or negative) is done by inverting all the bits and then adding 1.
    - This has the advantage of having only one representation of 0 (instead of negative 0 and positive 0).
    - With 2's complement: "a signed integral type with n bits [...] represent[s] numbers from −2^(n−1) through 2^(n−1) − 1." So with 8 bits, the numbers from -128 to 127 can be encoded.
- **Arithmetic**:

        - Adding works as normal. Just remember that 1 + 1 = 10, so gotta carry the 1.
        - With subtraction, as normal but just be careful with borrowing from the left. If substracting 1 from 0 (0 - 1), borrow from the first place to the left where it's 1 - 0. If have to borrow multiple times, the leftmost digit (the one I'm really borrowing from) becomes a 0, the intervening digits get a decimal place added to them and then since they get borrowed from too a 1 is subtracted (so a 0 becomes 10 then 1 and a 1 becomes 11 then 10), and the original digit that needed to be increased becomes a 10.
        - Multiplication can be done with left bitshifts. So because 3 * 5 is equivalent to 3 * (4 + 1), you can bitshift 3 to the left 2 places (which is 3 * 2^2 = 3 * 4) and then add 3 (3 * 1) back in.
        - Division can be done with right bitshifts, but just remember that it's integer division--rounded down basically.
- **Bitwise operations**:
    - & = AND
    - ^ = XOR
    - | = (inclusive) OR
    - x << n = left-shifts x by n places (0s appended at the end). Equivalent to x * 2^n
    - x >> n = right-shifts x by n places using sign extension. Equivalent to x / 2^n (integer division).
    - x >>> n = right-shifts x by n places where 0s are shifted into the leftmost spots.
    - ~ = flips bits (unary operator)
- **Bit facts & tricks**: (Since operations are bitwise (occur bit by bit), these are all equivalent to their series-equivalents. E.g.: x ^ 0 is the same as x ^ 0s. If a statement is true for a single bit, it's true for a sequence of bits)
    - ^ (XOR)
        - x ^ 0 = x
        - x ^ 1 = ~x (~ = negation)
        - x ^ x = 0
    - & (AND)
        - x & 0 = 0

        - x & 1 = x
        - x & x = x
    - | (inclusive OR)
        - x | 0 = x
        - x | 1 = 1
        - x | x = x
    - Swapping two values without a temporary variable:
        - a = a ^ b
        - b = a ^ b
        - a = a ^ b

III – Arrays and Strings

- **Hash tables** are important! Note that insertion and find/lookup proceed through identical starting steps: hash the key and go to that table location. This enables the O(1) complexity for both.
    - Chaining is the common way to handle collisions. Just means each position is actually the head of a linked list.
    - Open addressing is the other method. Deletions are tricky with this.
- Arrays offer easy random access but hard modification (have to move elements around).

IV – Linked Lists

- All a linked list *really is* is the head node (which points to the next node, and so on).
- **Deleting a node** n is just setting the references to skip n: prev.next = n.next;
    - Can also delete a curr node even without a prev pointer: curr.data = curr.next.data; curr.next = curr.next.next;
- Just make sure to check for the null pointer (!!) and be aware of the head pointer.
- Linked lists offer easy modification but non-existent random access.
- An important technique is **the "runner" one**:
    - Have two pointers iterating through the list, with one pointer either ahead by a fixed amount or actually moving faster.
    - For example, to determine the midpoint of a linked list, have two pointers such that one of them jumps 2 nodes at once and the other just 1. When the fast pointer hits the end, the slow one will be in the middle of the list.

- Recursion can often help.

V – Stacks and Queues

- **Stacks** are last-in first-out (LIFO), like a stack of dinner plates or trays.
- **Queues** are first-in first-out (FIFO), like a queue at an amusement park ride.
- Both are easily implemented with a linked list.
- With a stack, there's only one access point: the top (the linked list's head), and nodes point down / toward the tail.
- With a queue, there are two: the first node (the head [of the line]) and the last (the tail). Nodes point back / toward the tail and are added to the tail (!).
- **Priority queues** are neat structures (technically ADTs) where ordering is determined not by insertion time, but by some priority field:
  - Support insertion, find-min (or find-max), and delete-min (delete-max) operations.
  - The idea is that if I consistently just want the highest priority element, the priority queue will take care of that for me at insertion time, instead of my having to manually resort at extraction time.
  - Backing data structures include heaps (and balanced binary trees).
  - Generally implemented by keeping an extra pointer to the highest priority element, so on insertion, update iff new element is higher priority, and on deletion, delete then use find-min (or find-max) to restore the pointer.
- Popping everything from a stack and pushing it all to another stack *reverses* the ordering (Pushing everything back of course reverts the ordering).

VI – Trees

- A **tree** is in part defined as a node (the root), which holds some value, together with references to other nodes (the children, themselves trees). Because this defines a directed graph, these conditions are added:
  - at most 1 reference can point to any given node (i.e., a node has no more than 1 parent)
  - and no node in the tree points toward the root.
- As such, a tree is really just a special case of a directed graph (digraph):
  - The graph definition: a connected (directed in this context, but undirected in, e.g., the context of a spanning tree) graph with no cycles.

- **Binary trees** (branching factor = 2) are recursively built from nodes that have pointers to a left and right child.
- Binary trees aren't necessarily **binary search trees** (BSTs), which require that the left children are less than or equal to the current node which is less than all the right nodes.
- Balanced trees are better. It also doesn't mean perfectly balanced.
- **Depth and height**:
  - A node's depth = the number of *edges* from the node to the tree's root node. A root node will have a depth of 0.
  - A node's height = the number of edges on the *longest* path from the node to a leaf. A leaf node will have a height of 0.
  - The depth and height of a *tree* are equal.
- **Full and complete** is a tree where all leaves are at the bottom and each non-leaf node has exactly 2 children
  - Full = every node other than the leaves has 2 children. I.e., every node either has 0 or 2 children.
  - Complete = every level (except maybe the last) is *completely* filled and all nodes are as far left as possible. I.e., the tree is as compact as possible.
  - In a perfect tree, $h = O(\log n)$ because $h = \log_2 (n + 1) - 1$ and we drop the less significant terms.
- **Depth-first traversal (DFS)** prefixes are in reference to when the root/current node is visited. So pre-order traversal means first root, then left children, then right; post-order means first left, then right, then root. Make sure to recurse fully when traversing.
- Only one kind of breadth-first traversal -- the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right.
- **In-order successor**:
  - Important in a lot of binary tree operations.
  - The next node in an in-order traversal; i.e., the node with the smallest key that's still greater than the current node's key.
  - Pseudo-pseudocode:
    - If right child, go to it then as far left as possible. Must be smallest key in right subtree.

        - Otherwise, must travel up the tree (left child / subtree by definition smaller, i.e., can't be successor):
            - If current node is the left child of its parent, then the parent must be the successor.
            - Otherwise, keep going up until escape the right subtree and hit the middle. I.e., when the current node is a left child, its parent must be the in-order successor.
- **Heaps**:
    - A natural implementation of the priority queue ADT (the other being a balanced binary tree).
        - The highest priority element (whether min or max) recursively sits at the top of the heap.
        - Works by maintaining a partial ordering, so less expensive than fully sorted, but more valuable than no ordering.
    - A *complete* tree where the highest (or lowest) priority element is always stored at the root -- hence a *heap*. A max heap is one where a node is always greater than or equal to its children; a min heap is one where a node is always lesser than or equal to its children.
        - Are used to implement priority queues -- queues where the ordering is based on priority, not on position in queue -- and to do heap sort.
        - There is no implied ordering within a level (that is, no ordering between siblings) and so a heap is not a sorted structure.
    - Insertion and deletion (get max / min element) are both O(log N) where N is the number of nodes.
    - (Binary) **Heaps are implemented in arrays**.
        - Note that *any* binary tree can be so implemented, but that it makes particular sense for heaps, because heaps are guaranteed to be complete trees and thus the array implementation will be compact.
        - If a non-complete tree is implemented in an array, then there will be empty slots because a node may have only 1 child and the tree may go on quite a bit deeper than that level.

    - Saves space because can use array position to implicitly maintain the ordering property, instead of storing pointers.
- In the array implementation:
    - Let n be the number of elements in the heap and i be an arbitrary valid index of the array storing the heap. If the tree root is at index 0, with valid indices 0 through n − 1, then each element a at index i has:
        - children at indices 2i + 1 and 2i + 2
        - and its parent at floor((i − 1)/2).
    - Alternatively, if the tree root is at index 1, with valid indices 1 through n, then each element a at index i has:
        - children at indices 2i and 2i +1
        - and its parent at index floor(i/2).
- **Operations**:
    - Both insert and remove -- remove only removes the root since that's all we care about -- are done to an end of the heap to maintain the compact shape. Then, to modify the ordering property, the heap is traversed (respectively, up-heap / sift-up and down-heap / sift-down). Both operations take O(log N) time.
    - Up-heap (or sift-up) is used, e.g., to restore the heap ordering when a new element is added to the end/bottom:
        - Compare the added element with its parent; if they are in the correct order, stop.
        - If not, swap the element with its parent and return to the previous step.
    - Down-heap (or sift-down) is used, e.g., to restore the heap ordering when the root is deleted/removed and replaced with the last element (on the last level):
        - Compare the new root with its children; if they are in the correct order, stop.
        - If not, swap the element with one of its children and return to the previous step (for the newly ordered subtree). (Swap with its smaller child in a min-heap and its larger child in a max-heap.
    - Heapify, given a complete binary tree, turns the data structure into a heap:

        - We want the heap property to obtain, so we need to ensure that each parent node is greater (or lesser, if min heap) than its children.
        - Starting at the last parent node (take the last node and use index arithmetic to figure out the last parent), call down-heap on it. This will make the subtree rooted at this parent node a heap.
        - Continue on for all the other parent nodes.
- **Heapsort** just entails copying elements to be sorted into an array, heapifying it, then taking the max (root) out each time and putting it at the end of the result array.
    - This can be done in-place (in the same array) where one section is for the heap and the other for the sorted list. Thus, each iteration of the algorithm will reduce the heap by 1 and increase the sorted list by 1.
    - Heapsort is similar to insertion sort in that each step of the algorithm takes the max element from the unsorted section and moves it to the sorted section.
- Note that heaps are *not* binary search trees, so we can't efficiently find a given key using binary search.

VII – Graphs

- The graph ADT comprises a set of vertices (or nodes) together with a set of pairs of vertices, the edges. A vertex can be any arbitrary abstract object.
- Graphs are an extremely powerful concept. Linked lists, trees, state transition diagrams, etc. are all just cases of graphs.
- **Terminology**:
    - **Direction**:
        - Directed graph (or digraph) = a graph where the edges have a direction associated with them. I.e., each edge is now an ordered pair of vertices.
        - In conceptual terms, direction implies that relations between vertices are not symmetric. For example, a graph of friends would be undirected, since X being friends with Y implies that Y is friends with X. And on the other hand, a graph of Twitter followers would be directed, since X being a follower of Y does *not* imply that Y is a follower of X.
    - **Multiple edges & loops**:
        - Multiple (or parallel) edges = 2 or more edges that connect the same 2 vertices.

    - Loop = an edge that connects a vertex to itself.
    - Multigraph = a graph that can have multiple edges and (depending on the convention) sometimes loops.
    - Simple graph = a graph that cannot have multiple edges or loops.
- **Connectivity**:
    - Connected graph = graph with no vertices by themselves; i.e., there is some path between every pair of vertices / every pair of the graph's vertices is connected.
    - A connected component is a maximal subgraph that is (1) connected and (2) not connected to any vertices in the rest of the subgraph.
        - Subgraph of a graph G = graph whose vertices are a subset of G's vertices and whose edges are a subset of G's edges. Note that this is subset, not proper subset.
        - Tracking the connected components of a changing graph is a straightforward partitioning problem, which can be naturally solved with the application of disjoint-set (or union-find) data structures.
        - Maximal such subgraph because otherwise there would be "overlapping" connected components. This constrains it such that each vertex and each edge belongs to exactly one connected component. Mathematically, connected components partition their subgraph.
    - A vertex cut (or separating set) = the set of vertices whose removal would make the graph disconnected.
    - Easy to compute connectivity: if the number of nodes visited through either BFS or DFS equals the number of vertices, then the graph is connected.
- **Weighted graphs** have weights (or costs or distances) associated with their edges. So 2 paths can have the same number of edges (same path length), but have different total weights (or distances) because their edges' weights differ.
- **Cut**:
    - A partition of a graph's vertices into two disjoint subsets.
    - Also defines a cut-set, the set of edges that have 1 endpoint in each subset of the partition. I.e., the edges that cross the cut (also called a crossing edge). Sometimes a cut is identified as its cut-set.

        - The size / weight of a cut is the (total) weight of the cut-set, the edges crossing the cut. If unweighted, the size is the number of such crossing edges.
- **Two *primary* implementation methods / data structures**:
    - **Adjacency list**:
        - Each vertex is associated with an unordered list that contains its neighbors.
        - More efficient than the adjacency matrix in returning a given vertex's neighbors, O(1); less efficient in testing whether two vertices are neighbors, O(|V|).
          |V| is the number of vertices.
        - Slow in removing a vertex / edge, because must find all vertices / edges.
        - To represent weights, each node in the neighbor list could also contain the weight.
    - **Adjacency matrix**:
        - A matrix where each non-diagonal entry Aij is the number of edges from vertex i to vertex j, and the diagonal entry Aii, depending on the convention, is either once or twice the number of edges (loops) from vertex i to itself.
        - Sometimes the matrix value is just a boolean, if there can be no parallel edges (i.e., the graph is a simple graph, not a multigraph).
        - In a graph without loops, the diagonal in the matrix will have all zero entries.
        - Adjacency matrices are space-efficient, because each matrix entry only requires one bit. However, for a sparse graph (few edges), adjacency lists use less space because they do not represent nonexistent edges.
        - More efficient than the adjacency list in determining whether two vertices are neighbors, O(1); less efficient in getting all of a given vertex's neighbors because the entire row must be scanned, O(|V|).
        - Slow to add or remove vertices, because the matrix must be resized / copied.
        - To represent weights, the matrix value could be the weight of that edge.
- **Graph traversals**:
    - **Depth-first search (DFS)**:
        - DFS visits the child nodes before visiting the sibling nodes; i.e., it traverses the depth of any particular path before exploring its breadth.
        - Pseudo-pseudocode:
            - Visit node r and then iterate through each of r's unvisited neighbors.

        - When visiting a neighbor n, immediately visit all of *his* neighbors. And so on (Thus depth-first. So neighbor n and its children are exhaustively searched before r moves on to its other adjacent nodes).
        - If no neighbors, i.e., a dead end, backtrack (pop the stack) until reach unvisited node.
    - Easy to iteratively implement with a stack. Just make sure to "mark" visited nodes to prevent infinite loops.
    - Also easy to recursively implement (also via a stack, i.e., the program's call stack)
- **Breadth-first search (BFS)**:
    - BFS explores the neighbor nodes first, before moving to the next "level."
    - Pseudo-pseudocode:
        - Visit node r and then iterate through each of r's unvisited neighbors.
        - When visiting a neighbor n, add its neighbors to queue, but *don't* visit them yet.
        - Visit *all* of node r's neighbors before visiting any of r's "grandchildren."
    - Implement iteratively with a queue and mark visited as always.
- DFS is the easiest if want to visit every node.
- But BFS can get shortest path(s) first.
- Both traversals (for the entire graph) take time linear to the size of the graph (number of vertices + number of edges), O(|V| + |E|) and space linear to the number of vertices.
- Note that just as trees are a special case of graphs, tree traversals are a special case of graph traversals.

- **DAGs and topological sorts** are important concepts to know:
    - Topological sort is relatively easy to implement, just remember the definition (for vertices *u*, *v* where *u* has a directed edge to *v*, *u* has to come before *v* in the topological ordering) and use a stack.
    - Start at a given vertex, add to visited set, recursively (depth-first) visit and explore unvisited neighbors, when have fully explored *v*'s neighbors add *v* to stack, and pop all at end for topologically sorted ordering.

VIII – Sorting and Searching

- Bubble sort and selection sort suck. Both O(n^2).

- Merge sort and quick sort are both good and, in the average case, O(n Log(n)).
- **Merge sort**:
  - Worse case is also O(n Log(n)).
  - From Algorithms class:
    - like quicksort in that it recursively splits array and builds it back in right order (both also divide-and-conquer)
    - divide array into 2 halves, recursively sort halves, merge results
    - recursive calls then sorting action (contra quicksort); "on the way up"; only way sorting occurs is through these "on the way up" merges
    - strength: guaranteed N log N performance
    - weakness: extra space proportional to N (aux array for merging)
    - time efficiency: average, best, & worst -- O(N log N), requires some aux space
  - All the work happens in the merging where consider both arrays and pick the smallest element to copy to main array each time. Base case is merging two 1-element arrays.
  - Usually stable.
- **Quicksort**:
  - Worse case is O(n^2) when array already sorted, but small constant factor and can be in-place.
  - From Algorithms class:
    - array rearranged such that, when two subarrays sorted, the whole array is ordered (contra merge where array is broken into 2 subarrays to be sorted and then combined to make whole ordered array)
    - recursive calls happen after working on whole array
    - partition/pivot not necessarily in middle (Or necessarily the median value, leading to the worst case performance).
    - improvements: insertion sort for tiny arrays, median of 3, randomize beforehand
    - strength: average case N log N, in-place so small usage of memory (small aux stack), shorter inner loop so fast in practice as well
    - weakness: worst case is quadratic, happens with already sorted array (where pivot = first item)

        - time efficiency: average & best -- O(N log N), worst -- O(N^2), small constant factor
    - To partition, have two pointers at each end working toward each other. Stop pointers when each reaches a value out of place. Swap them.
    - Usually not stable.
- Difference is that in merge sort, the sorting action happens on the way up (when ordered subarrays are merged), whereas in quicksort, the sorting happens on the way down when the (sub)array is split and the low and high elements are separated and the pivot element is placed in its correct final position.
    - "The difference between the two is basically which linear operation they leverage. Quick Sort is efficient because you can partition a list into items that come before and items that come after a given item in linear time. Merge Sort is efficient because given two already sorted lists you can combine the two, and maintain sorted order, in linear time."

IX – Dynamic Programming and Memoization

- When we **memoize** (cache a subproblem's answer / recursive function's output so we don't need to recompute it again), the memoization dictionary probably needs to be outside the function to save/share state. I.e., make a class.
- Top-down recursion can be memory-intensive because of building up the call stack. Can avoid by going bottom-up and using DP.
- DP often involves making a binary decision at each (bottom-up) step, e.g., do I include this coin / item in my knapsack / element, etc. If I do include it, is that more optimal (value or stock market profit or so on) than if I don't, where either calculation usually makes use of previously calculated optima.
    - To get those previous locally optimal calculations, we generally use a matrix or list to store the previous solutions. But sometimes that can store more state than we really need, see e.g. the bottom-up Fibonacci implementation where we can store the answers to the previous sub-problems we need in just 2 variables.
- **Interview Cake**:
    - "Dynamic programming is kind of like the next step up from greedy. You're taking that idea of "keeping track of what we need in order to update the best answer so far," and

applying it to situations where the new best answer so far might not just have to do with the previous answer, but some other earlier answer as well.

- So as you can see in this problem, we kept track of all of our previous answers to smaller versions of the problem (called "sub-problems") in a
  big list called ways_of_doing_n_cents.
- Again, same idea of keeping track of what we need in order to update the answer as we go, like we did when storing the max product of 2, min product of 2, etc in the highest product of 3 question. Except now the thing we need to keep track of is all our previous answers, which we're keeping in a list.
- We built that list bottom-up, but we also talked about how we could do it top-down and memoize. Going bottom-up is cleaner and usually more efficient, but often it's easier to think of the top-down version first and try to adapt from there."

X – Time and Space Complexity

- http://bigocheatsheet.com/
- O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)
- Time complexity of recursive algorithms that make multiple calls per call is often exponential, think of the fib algorithm: f(n) = f(n-1) + f(n-2).
- On a similar note, even if a function doesn't take any extra space with local variables, if it makes a recursive call, that state has to be added to the call stack. (The caller function "pauses" and the callee is pushed to the call stack. When the callee is done executing, it's popped and control is returned to the caller.)
    - If the recursion is tail recursive (last call of function is the recursive call), however, then can be tail call optimized so that the last call acts as a GOTO and there's no space overhead—i.e., don't need to push frame to call stack.
- Space used for a recursive function is *proportional to the maximum depth of its recursion tree*, in other words, the length of the longest path in the tree.
    - It's not the number of function calls since the call stack is popping function stack frames as it finishes executing each.

XI – Behavioral

- **Haseeb**:
  http://haseebq.com/how-to-break-into-tech-job-hunting-and-interviews/

- Almost every question you'll be asked will be a permutation of one of these four:
    - What's your story / walk me through your resume / why'd you leave your last job? (these are essentially the same question)
    - Why us?
    - Tell me about a challenging bug you faced and how you solved it.
    - Tell me about an interesting project you worked on.
- The first question is particularly important. Essentially, they want to hear your personal narrative. Your answer will strongly influence their perception of you.
- This really just comes down to storytelling. Consider yourself a character in a story, and structure the story with a beginning, middle, and end. There should be inflection points, characterization, and easy to understand motivations. Keep it as short as possible, while preserving color and what makes you interesting. Try not to be negative. Frame your story around seeking challenge and wanting to better yourself, rather than rejecting or disliking things.
- You will tell this narrative again and again. If you interview enough, eventually it will congeal into a script. The best way to improve at it is to literally practice it out loud and listen to a recording of it. Also try to get feedback from someone whose judgment you trust, and ask them to be as ruthlessly critical as possible.
- For the remaining three questions, you should have have pre-crafted answers. If you're at a loss for stories, it may help to sit down and just brainstorm every relevant story you can remember (for example, every bug you remember working on), and then narrow it down from a larger list.
- It's hard to practice this effectively in a vacuum, so this is a good thing to work with someone else on.

- **Interview Cake**: https://www.interviewcake.com/coding-interview-tips#chitchat
    - Chitchat like a pro.
    - Before diving into code, most interviewers like to chitchat about your background. They're looking for:
        - Metacognition about coding. Do you think about how to code well?

        - Ownership/leadership. Do you see your work through to completion? Do you fix things that aren't quite right, even if you don't have to?
        - Communication. Would chatting with you about a technical problem be useful or painful?
    - You should have at least one:
        - example of an interesting technical problem you solved
        - example of an interpersonal conflict you overcame
        - example of leadership or ownership
        - story about what you should have done differently in a past project
        - piece of trivia about your favorite language, and something you do and don't like about said language
        - question about the company's product/business
        - question about the company's engineering strategy (testing, Scrum, etc)
    - Nerd out about stuff. Show you're proud of what you've done, you're amped about what they're doing, and you have opinions about languages and workflows.
- **Questions to ask**:
    - https://www.reddit.com/r/cscareerquestions/comments/4ce2s3/resource_interview_questions_my_massive/
    - What does success look like for this position? How will I know if I am accomplishing what is expected of me?
    - What is the last project you shipped? What was the goal, how long did it take, what were the stumbling blocks, what tools did you use?
    - What will my first 90 days in this role look like? First 180 days?
    - Who will I report to and how many people report to that person? Do they have regular 1:1 with their team members?
    - Why did the last person who quit this team leave? The company?
    - If a startup, how long is your runway? How are financial decisions made?
    - What would be my first project here? Has someone already been working on this or is this in the aspirational stage?
    - What is the current state of the data infrastructure? How much work needs to be done on getting the infrastructure and pipeline into shape before we start analyzing that data?

- Emphasize certain traits in my elevator speech story:
  - passion
  - willingness to learn / intellectual curiosity
  - seeking challenges
  - into data