Advanced Topics

XI

This section includes topics that are mostly beyond the scope of interviews but can come up on occasion. Interviewers shouldn't be surprised if you don't know these topics well. Feel free to dive into these topics if you want to. If you're pressed for time, they're low priority.

XI

Advanced Topics

When writing the 6th edition, I had a number of debates about what should and shouldn't be included. Red-black trees? Dijkstra's algorithm? Topological sort?

On one hand, I'd had a number of requests to include these topics. Some people insisted that these topics are asked "all the time" (in which case, they have a very different idea of what this phrase means!). There was clearly a desire—at least from some people—to include them. And learning more can't hurt, right?

On the other hand, I know these topics to be rarely asked. It happens, of course. Interviewers are individuals and might have their own ideas of what is "fair game" or "relevant" for an interview. But it's rare. When it does come up, if you don't know the topic, it's unlikely to be a big red flag.

> Admittedly, as an interviewer, I *have* asked candidates questions where the solution was essentially an application of one of these algorithms. On the rare occasions that a candidate already knew the algorithm, they did not benefit from this knowledge (nor were they hurt by it). I want to evaluate your ability to solve a problem you haven't seen before. So, I'll take into account whether you know the underlying algorithm in advance.

I believe in giving people a fair expectation of the interview, not scaring people into excess studying. I also have no interest in making the book more "advanced" so as to help book sales, at the expense of your time and energy. That's not fair or right to do to you.

(Additionally, I didn't want to give interviewers—who I know to be reading this—the impression that they can or should be covering these more advanced topics. Interviewers: If you ask about these topics, you're testing knowledge of algorithms. You're just going to wind up eliminating a lot of perfectly smart people.)

But there are many borderline "important" topics. They're not often asked, but sometimes they are.

Ultimately, I decided to leave the decision in your hands. After all, you know better than I do how thorough you want to be in your preparation. If you want to do an extra thorough job, read this. If you just love learning data structures and algorithms, read this. If you want to see new ways of approaching problems, read this.

But if you're pressed for time, this studying isn't a super high priority.

▸ Useful Math

Here's some math that can be useful in some questions. There are more formal proofs that you can look up online, but we'll focus here on giving you the intuition behind them. You can think of these as informal proofs.

Sum of Integers 1 through N

What is $1 + 2 + \ldots + n$? Let's figure it out by pairing up low values with high values.

If n is even, we pair 1 with n, 2 with n - 1, and so on. We will have $\frac{n}{2}$ pairs each with sum n + 1.

If n is odd, we pair 0 with n, 1 with n - 1, and so on. We will have $\frac{n+1}{2}$ pairs with sum n.

n is even			
pair #	**a**	**b**	**a + b**
1	1	n	n + 1
2	2	n - 1	n + 1
3	3	n - 2	n + 1
4	4	n - 3	n + 1
...	...	...	...
$\frac{n}{2}$	$\frac{n}{2}$	$\frac{n}{2}+1$	n + 1
total:	$\frac{n}{2}*(n+1)$		

n is odd			
pair #	**a**	**b**	**a + b**
1	0	n	n
2	1	n - 1	n
3	2	n - 2	n
4	3	n - 3	n
...	...	...	...
$\frac{n+1}{2}$	$\frac{n-1}{2}$	$\frac{n+1}{2}$	n
total:	$\frac{n+1}{2} * n$		

In either case, the sum is $\frac{n(n+1)}{2}$.

This reasoning comes up a lot in nested loops. For example, consider the following code:

```
for (int i = 0; i < n; i++) {
   for (int j = i + 1; j < n; j++) {
      System.out.println(i + j);
   }
}
```

On the first iteration of the outer for loop, the inner for loop iterates n - 1 times. On the second iteration of the outer for loop, the inner for loop iterates n-2 times. Next, n - 3, then n - 4, and so on. There are $\frac{n(n-1)}{2}$ total iterations of the inner for loop. Therefore, this code takes $O(n^2)$ time.

Sum of Powers of 2

Consider this sequence: $2^0 + 2^1 + 2^2 + \ldots + 2^n$. What is its result?

A nice way to see this is by looking at these values in binary.

	Power	Binary	Decimal
	2^0	00001	1
	2^1	00010	2
	2^2	00100	4
	2^3	01000	8
	2^4	10000	16
sum:	2^5-1	11111	32 - 1 = 31

Therefore, the sum of $2^0 + 2^1 + 2^2 + \ldots + 2^n$ would, in base 2, be a sequence of (n + 1) 1s. This is $2^{n+1} - 1$.

Takeaway: The sum of a sequence of powers of two is roughly equal to the *next* value in the sequence.

Bases of Logs

Suppose we have something in $\log_2$ (log base 2). How do we convert that to $\log_{10}$? That is, what's the relationship between $\log_b k$ and $\log_x k$?

Let's do some math. Assume $c = \log_b k$ and $y = \log_x k$.

```
log_b k = c --> b^c = k          // This is the definition of log.
log_x(b^c) = log_x k              // Take log of both sides of b^c = k.
c log_x b = log_x k               // Rules of logs. You can move out the exponents.
c = log_b k = log_x k / log_x b   // Dividing above expression and substituting c.
```

Therefore, if we want to convert $\log_2 p$ to $\log_{10}$, we just do this:

$$\log_{10} p = \frac{\log_2 p}{\log_2 10}$$

Takeaway: Logs of different bases are only off by a constant factor. For this reason, we largely ignore what the base of a log within a big O expression. It doesn't matter since we drop constants anyway.

Permutations

How many ways are there of rearranging a string of n unique characters? Well, you have n options for what to put in the first characters, then n - 1 options for what to put in the second slot (one option is taken), then n - 2 options for what to put in the third slot, and so on. Therefore, the total number of strings is n!.

$$n! = \underline{n} * \underline{n-1} * \underline{n-2} * \underline{n-3} * \ldots * \underline{1}$$

What if you were forming a k-length string (with all unique characters) from n total unique characters? You can follow similar logic, but you'd just stop your selection/multiplication earlier.

$$\frac{n!}{(n-k)!} = \underline{n} * \underline{n-1} * \underline{n-2} * \underline{n-3} * \ldots * \underline{n-k+1}$$

Combinations

Suppose you have a set of n distinct characters. How many ways are there of selecting k characters into a new set (where order doesn't matter)? That is, how many k-sized subsets are there out of n distinct elements? This is what the expression n-choose-k means, which is often written $\binom{n}{k}$.

Imagine we made a list of all the sets by first writing all k-length substrings and then taking out the duplicates.

From the above *Permutations* section, we'd have $n!/(n-k)!$ k-length substrings.

Since each k-sized subset can be rearranged k! unique ways into a string, each subset will be duplicated k! times in this list of substrings. Therefore, we need to divide by k! to take out these duplicates.

$$\binom{n}{k} = \frac{1}{k!} * \frac{n!}{(n-k)!} = \frac{n!}{k!(n-k)!}$$

Proof by Induction

Induction is a way of proving something to be true. It is closely related to recursion. It takes the following form.

Task: Prove statement P(k) is true for all k >= b.

- Base Case: Prove the statement is true for P(b). This is usually just a matter of plugging in numbers.
- Assumption: Assume the statement is true for P(n).
- Inductive Step: Prove that *if* the statement is true for P(n), then it's true for P(n+1).

This is like dominoes. If the first domino falls, and one domino always knocks over the next one, then all the dominoes must fall.

Let's use this to prove that there are 2^n subsets of an n-element set.

- Definitions: let $S = \{a_1, a_2, a_3, \ldots, a_n\}$ be the n-element set.

- Base case: Prove there are 2^0 subsets of `{}`. This is true, since the only subset of `{}` is `{}`.
- Assume that there are 2^n subsets of $\{a_1, a_2, a_3, \ldots, a_n\}$.
- Prove that there are 2^{n+1} subsets of $\{a_1, a_2, a_3, \ldots, a_{n+1}\}$.

  Consider the subsets of $\{a_1, a_2, a_3, \ldots, a_{n+1}\}$. Exactly half will contain a_{n+1} and half will not.

  The subsets that do not contain a_{n+1} are just the subsets of $\{a_1, a_2, a_3, \ldots, a_n\}$. We assumed there are 2^n of those.

  Since we have the same number of subsets with x as without x, there are 2^n subsets with a_{n+1}.

  Therefore, we have $2^n + 2^n$ subsets, which is 2^{n+1}.

Many recursive algorithms can be proved valid with induction.

Topological Sort

A topological sort of a directed graph is a way of ordering the list of nodes such that if `(a, b)` is an edge in the graph then a will appear before b in the list. If a graph has cycles or is not directed, then there is no topological sort.

There are a number of applications for this. For example, suppose the graph represents parts on an assembly line. The edge (Handle, Door) indicates that you need to assemble the handle before the door. The topological sort would offer a valid ordering for the assembly line.

We can construct a topological sort with the following approach.

1. Identify all nodes with no incoming edges and add those nodes to our topological sort.
   - We know those nodes are safe to add first since they have nothing that needs to come before them. Might as well get them over with!
   - We know that such a node must exist if there's no cycle. After all, if we picked an arbitrary node we could just walk edges backwards arbitrarily. We'll either stop at some point (in which case we've found a node with no incoming edges) or we'll return to a prior node (in which case there is a cycle).
2. When we do the above, remove each node's outbound edges from the graph.
   - Those nodes have already been added to the topological sort, so they're basically irrelevant. We can't violate those edges anymore.
3. Repeat the above, adding nodes with no incoming edges and removing their outbound edges. When all the nodes have been added to the topological sort, then we are done.

More formally, the algorithm is this:

1. Create a queue `order`, which will eventually store the valid topological sort. It is currently empty.
2. Create a queue `processNext`. This queue will store the next nodes to process.
3. Count the number of incoming edges of each node and set a class variable `node.inbound`. Nodes typically only store their outgoing edges. However, you can count the inbound edges by walking through each node n and, for each of its outgoing edges `(n, x)`, incrementing `x.inbound`.
4. Walk through the nodes again and add to `processNext` any node where `x.inbound == 0`.
5. While `processNext` is not empty, do the following:
   - Remove first node n from `processNext`.

  » For each edge `(n, x)`, decrement `x.inbound`. If `x.inbound == 0`, append x to `processNext`.

  » Append n to `order`.

6. If `order` contains all the nodes, then it has succeeded. Otherwise, the topological sort has failed due to a cycle.

This algorithm does sometimes come up in interview questions. Your interviewer probably wouldn't expect you to know it offhand. However, it would be reasonable to have you derive it even if you've never seen it before.

▸ Dijkstra's Algorithm

In some graphs, we might want to have edges with weights. If the graph represented cities, each edge might represent a road and its weight might represent the travel time. In this case, we might want to ask, just as your GPS mapping system does, what's the shortest path from your current location to another point p? This is where Dijksta's algorithm comes in.

Dijkstra's algorithm is a way to find the shortest path between two points in a weighted directed graph (which might have cycles). All edges must have positive values.

Rather than just stating what Dijkstra's algorithm is, let's try to derive it. Consider the earlier described graph. We could find the shortest path from `s` to `t` by literally taking all possible routes using actual time. (Oh, and we'll need a machine to clone ourselves.)

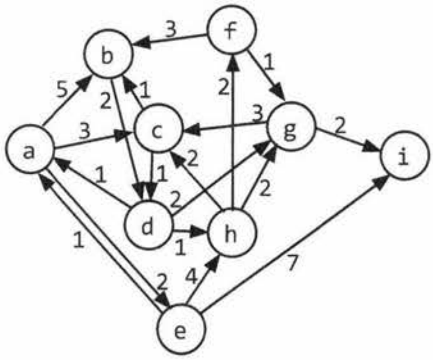
1. Start off at `s`.
2. For each of `s`'s outbound edges, clone ourselves and start walking. If the edge `(s, x)` has weight 5, we should actually take 5 minutes to get there.
3. Each time we get to a node, check if anyone's been there before. If so, then just stop. We're automatically not as fast as another path since someone beat us here from `s`. If no one has been here before, then clone ourselves and head out in all possible directions.
4. The first one to get to `t` wins.

This works just fine. But, of course, in the real algorithm we don't want to literally use a timer to find the shortest path.

Imagine that each clone could jump immediately from one node to its adjacent nodes (regardless of the edge weight), but it kept a `time_so_far` log of how long its path would have taken if it did walk at the "true" speed. Additionally, only one person moves at a time, and it's always the one with the lowest `time_so_far`. This is sort of how Dijkstra's algorithm works.

Dijkstra's algorithm finds the minimum weight path from a start node s to *every* node on the graph.

Consider the following graph.



Assume we are trying to find the shortest path from a to `i`. We'll use Dijkstra's algorithm to find the shortest path from a to all other nodes, from which we will clearly have the shortest path from a to `i`.

We first initialize several variables:

- `path_weight[node]`: maps from each node to the total weight of the shortest path. All values are initialized to infinity, except for `path_weight[a]` which is initialized to `0`.
- `previous[node]`: maps from each node to the previous node in the (current) shortest path.
- `remaining`: a priority queue of all nodes in the graph, where each node's priority is defined by its `path_weight`.

Once we've initialized these values, we can start adjusting the values of `path_weight`.

> A (min) **priority queue** is an abstract data type that—at least in this case—supports insertion of an object and key, removing the object with the minimum key, and decreasing a key. (Think of it like a typical queue, except that, instead of removing the oldest item, it removes the item with the lowest or highest priority.) It is an abstract data type because it is defined by its behavior (its operations). Its underlying implementation can vary. You could implement a priority queue with an array or a min (or max) heap (or many other data structures).

We iterate through the nodes in `remaining` (until `remaining` is empty), doing the following:

1. Select the node in `remaining` with the lowest value in `path_weight`. Call this node n.
2. For each adjacent node, compare `path_weight[x]` (which is the weight of the current shortest path from a to x) to `path_weight[n] + edge_weight[(n, x)]`. That is, could we get a path from a to x with lower weight by going through n instead of our current path? If so, update `path_weight` and `previous`.
3. Remove n from `remaining`.

When `remaining` is empty, then `path_weight` stores the weight of the current shortest path from a to each node. We can reconstruct this path by tracing through `previous`.

Let's walk through this on the above graph.

1. The first value of n is a. We look at its adjacent nodes (b, c, and e), update the values of `path_weight` (to 5, 3, and 2) and `previous` (to a) and then remove a from `remaining`.
2. Then, we go to the next smallest node, which is e. We previously updated `path_weight[e]` to be 2. Its adjacent nodes are h and `i`, so we update `path_weight` (to 6 and 9) and `previous` for both of those.

Observe that 6 is `path_weight[e]` (which is 2) + the weight of the edge `(e, h)` (which is 4).

3. The next smallest node is `c`, which has `path_weight` 3. Its adjacent nodes are b and d. The value of `path_weight[d]` is infinity, so we update it to 4 (which is `path_weight[c] + weight(edge c, d)`. The value of `path_weight[b]` has been previously set to 5. However, since `path_weight[c] + weight(edge c, b)` (which is 3 + 1 = 4) is less than 5, we update `path_weight[b]` to 4 and `previous` to `c`. This indicates that we would improve the path from a to b by going through `c`.

We continue doing this until `remaining` is empty. The following diagram shows the changes to the `path_weight` (left) and `previous` (right) at each step. The topmost row shows the current value for n (the node we are removing from `remaining`). We black out a row after it has been removed from `remaining`.

	INITIAL		n = a		n = e		n = c		n = b		n = d		n = h		n = g		n = f		FINAL	
	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr
a	0	-	removed																0	-
b	∞	-	5	a			4	c	removed										4	c
c	∞	-	3	a			removed												3	a
d	∞	-					4	c			removed								4	c
e	∞	-	2	a	removed														2	a
f	∞	-											7	h			removed		7	h
g	∞	-									6	d			removed				6	d
h	∞	-			6	e					5	d	removed						5	d
i	∞	-	∞	-	9	e									8	g			8	g

Once we're done, we can follow this chart backwards, starting at `i` to find the actual path. In this case, the smallest weight path has weight 8 and is `a -> c -> d -> g -> i`.

Priority Queue and Runtime

As mentioned earlier, our algorithm used a priority queue, but this data structure can be implemented in different ways.

The runtime of this algorithm depends heavily on the implementation of the priority queue. Assume you have v vertices and e nodes.

- If you implemented the priority queue with an array, then you would call `remove_min` up to v times. Each operation would take $O(v)$ time, so you'd spend $O(v^2)$ time in the `remove_min` calls. Additionally, you would update the values of `path_weight` and `previous` at most once per edge, so that's $O(e)$ time doing those updates. Observe that e must be less than of equal to v^2 since you can't have more edges than there are pairs of vertices. Therefore, the total runtime is $O(v^2)$.
- If you implemented the priority queue with a min heap, then the `remove_min` calls will each take $O(\log v)$ time (as will inserting and updating a key). We will do one `remove_min` call for each vertex, so that's $O(v \log v)$ (v vertices at $O(\log v)$ time each). Additionally, on each edge, we might call one update key or insert operation, so that's $O(e \log v)$. The total runtime is $O((v + e) \log v)$.

Which one is better? Well, that depends. If the graph has a lot of edges, then v^2 will be close to e. In this case, you might be better off with the array implementation, as $O(v^2)$ is better than $O((v + v^2) \log v)$. However, if the graph is sparse, then e is much less than v^2. In this case, the min heap implementation may be better.

▸ Hash Table Collision Resolution

Essentially any hash table can have collisions. There are a number of ways of handling this.

Chaining with Linked Lists

With this approach (which is the most common), the hash table's array maps to a linked list of items. We just add items to this linked list. As long as the number of collisions is fairly small, this will be quite efficient.

In the worst case, lookup is `O(n)`, where `n` is the number of elements in the hash table. This would only happen with either some very strange data or a very poor hash function (or both).

Chaining with Binary Search Trees

Rather than storing collisions in a linked list, we could store collisions in a binary search tree. This will bring the worst-case runtime to `O(log n)`.

In practice, we would rarely take this approach unless we expected an extremely nonuniform distribution.

Open Addressing with Linear Probing

In this approach, when a collision occurs (there is already an item stored at the designated index), we just move on to the next index in the array until we find an open spot. (Or, sometimes, some other fixed distance, like the `index + 5`.)

If the number of collisions is low, this is a very fast and space-efficient solution.

One obvious drawback of this is that the total number of entries in the hash table is limited by the size of the array. This is not the case with chaining.

There's another issue here. Consider a hash table with an underlying array of size 100 where indexes 20 through 29 are filled (and nothing else). What are the odds of the next insertion going to index 30? The odds are 10% because an item mapped to any index between 20 and 30 will wind up at index 30. This causes an issue called *clustering*.

Quadratic Probing and Double Hashing

The distance between probes does not need to be linear. You could, for example, increase the probe distance quadratically. Or, you could use a second hash function to determine the probe distance.

▸ Rabin-Karp Substring Search

The brute force way to search for a substring S in a larger string B takes `O(s(b-s))` time, where `s` is the length of S and `b` is the length of B. We do this by searching through the first `b - s + 1` characters in B and, for each, checking if the next `s` characters match S.

The Rabin-Karp algorithm optimizes this with a little trick: if two strings are the same, they must have the same hash value. (The converse, however, is not true. Two different strings can have the same hash value.)

Therefore, if we efficiently precompute a hash value for each sequence of `s` characters within B, we can find the locations of S in `O(b)` time. We then just need to validate that those locations really do match S.

For example, imagine our hash function was simply the sum of each character (where space = 0, a = 1, b = 2, and so on). If S is `ear` and B = `doe are hearing me`, we'd then just be looking for sequences where the sum is 24 (e + a + r). This happens three times. For each of those locations, we'd check if the string really is `ear`.

char:	d	o	e		a	r	e		h	e	a	r	i	n	g		m	e
code:	4	15	5	0	1	18	5	0	8	5	1	18	9	14	7	0	13	5
sum of next 3:	24	20	6	19	24	23	13	13	14	24	28	41	30	21	20	18		

If we computed these sums by doing `hash('doe')`, then `hash('oe ')`, then `hash('e a')`, and so on, we would still be at `O(s(b-s))` time.

Instead, we compute the hash values by recognizing that `hash('oe ') = hash('doe') - code('d') + code(' ')`. This takes `O(b)` time to compute all the hashes.

You might argue that, still, in the worst case this will take `O(s(b-s))` time since many of the hash values could match. That's absolutely true—for this hash function.

In practice, we would use a better *rolling hash function*, such as the Rabin fingerprint. This essentially treats a string like `doe` as a base 128 (or however many characters are in our alphabet) number.

```
hash('doe') = code('d') * 128^2 + code('o') * 128^1 + code('e') * 128^0
```

This hash function will allow us to remove the `d`, shift the `o` and `e`, and then add in the space.

```
hash('oe ') = (hash('doe') - code('d') * 128^2) * 128 + code(' ')
```

This will considerably cut down on the number of false matches. Using a good hash function like this will give us expected time complexity of `O(s + b)`, although the worst case is `O(sb)`.

Usage of this algorithm comes up fairly frequently in interviews, so it's useful to know that you can identify substrings in linear time.

▸ AVL Trees

An AVL tree is one of two common ways to implement tree balancing. We will only discuss insertions here, but you can look up deletions separately if you're interested.

Properties

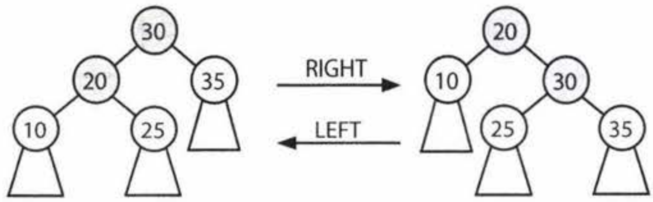
An AVL tree stores in each node the height of the subtrees rooted at this node. Then, for any node, we can check if it is height balanced: that the height of the left subtree and the height of the right subtree differ by no more than one. This prevents situations where the tree gets too lopsided.

```
balance(n) = n.left.height - n.right.height
        -1 <= balance(n) <= 1
```

Inserts

When you insert a node, the balance of some nodes might change to -2 or 2. Therefore, when we "unwind" the recursive stack, we check and fix the balance at each node. We do this through a series of rotations.

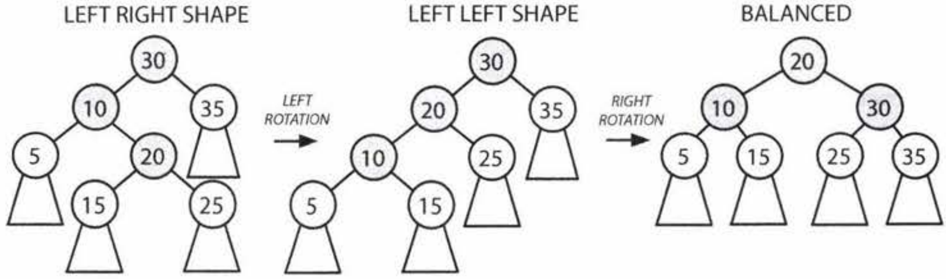
Rotations can be either left or right rotations. The right rotation is an inverse of the left rotation.



Depending on the balance and where the imbalance occurs, we fix it in a different way.

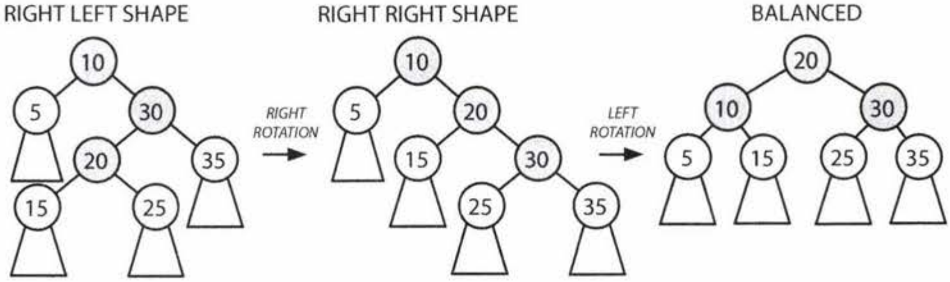
- *Case 1: Balance is 2.*

  In this case, the left's height is two bigger than the right's height. If the left side is larger, the left subtree's extra nodes must be hanging to the left (as in LEFT LEFT SHAPE) or hanging to the right (as in LEFT RIGHT SHAPE). If it looks like the LEFT RIGHT SHAPE, transform it with the rotations below into the LEFT LEFT SHAPE then into BALANCED. If it looks like the LEFT LEFT SHAPE already, just transform it into BALANCED.



- *Case 2: Balance is -2.*

  This case is the mirror image of the prior case. The tree will look like either the RIGHT LEFT SHAPE or the RIGHT RIGHT SHAPE. Perform the rotations below to transform it into BALANCED.



In both cases, "balanced" just means that the `balance` of the tree is between -1 and 1. It does not mean that the `balance` is 0.

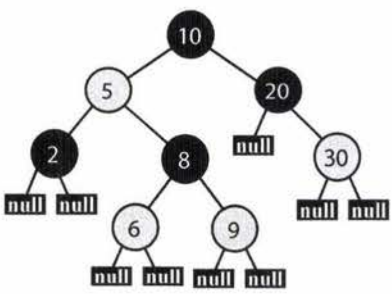
We recurse up the tree, fixing any imbalances. If we ever achieve a balance of 0 on a subtree, then we know that we have completed all the balances. This portion of the tree will not cause another, higher subtree to have a balance of -2 or 2. If we were doing this non-recursively, then we could break from the loop.

▸ Red-Black Trees

Red-black trees (a type of self-balancing binary search tree) do not ensure quite as strict balancing, but the balancing is still good enough to ensure O(log N) insertions, deletions, and retrievals. They require a bit less memory and can rebalance faster (which means faster insertions and removals), so they are often used in situations where the tree will be modified frequently.

Red-black trees operate by enforcing a quasi-alternating red and black coloring (under certain rules, described below) and then requiring every path from a node to its leaves to have the same number of black nodes. Doing so leads to a reasonably balanced tree.

The tree below is a red-black tree (where the red nodes are indicated with gray):



Properties

1. Every node is either red or black.
2. The root is black.
3. The leaves, which are NULL nodes, are considered black.
4. Every red node must have two black children. That is, a red node cannot have red children (although a black node can have black children).
5. Every path from a node to its leaves must have the same number of black children.

Why It Balances

Property #4 means that two red nodes cannot be adjacent in a path (e.g., parent and child). Therefore, no more than half the nodes in a path can be red.

Consider two paths from a node (say, the root) to its leaves. The paths must have the same number of black nodes (property #5), so let's assume that their red node counts are as different as possible: one path contains the minimum number of red nodes and the other one contains the maximum number.

- Path 1 (Min Red): The minimum number of red nodes is zero. Therefore, path 1 has b nodes total.
- Path 2 (Max Red): The maximum number of red nodes is b, since red nodes must have black children and there are b black nodes. Therefore, path 2 has 2b nodes total.

Therefore, even in the most extreme case, the lengths of paths cannot differ by more than a factor of two. That's good enough to ensure an O(log N) find and insert runtime.

If we can maintain these properties, we'll have a (sufficiently) balanced tree—good enough to ensure O(log N) insert and find, anyway. The question then is how to maintain these properties efficiently. We'll only discuss insertion here, but you can look up deletion on your own.

Insertion

Inserting a new node into a red-black tree starts off with a typical binary search tree insertion.

- New nodes are inserted at a leaf, which means that they replace a black node.
- New nodes are always colored red and are given two black leaf (NULL) nodes.

Once we've done that, we fix any resulting red-black property violations. We have two possible violations:

- Red violations: A red node has a red child (or the root is red).
- Black violations: One path has more blacks than another path.

The node inserted is red. We didn't change the number of black nodes on any path to a leaf, so we know that we won't have a black violation. However, we might have a red violation.

In the special case that where the root is red, we can always just turn it black to satisfy property 2, without violating the other constraints.

Otherwise, if there's a red violation, then this means that we have a red node under another red node. Oops!

Let's call N the current node. P is N's parent. G is N's grandparent. U is N's uncle and P's sibling. We know that:

- N is red and P is red, since we have a red violation.
- G is definitely black, since we didn't *previously* have a red violation.

The unknown parts are:

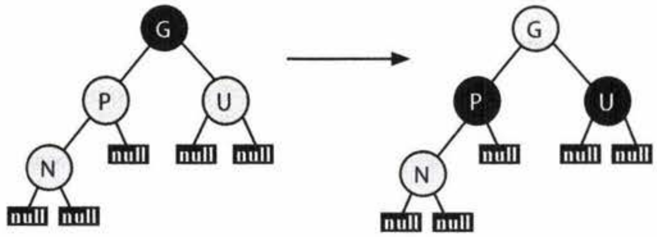
- U could be either red or black.
- U could be either a left or right child.
- N could be either a left or right child.

By simple combinatorics, that's eight cases to consider. Fortunately some of these cases will be equivalent.

- **Case 1: U is red.**

  It doesn't matter whether U is a left or right child, nor whether P is a left or right child. We can merge four of our eight cases into one.

  If U is red, we can just toggle the colors of P, U, and G. Flip G from black to red. Flip P and U from red to black. We haven't changed the number of black nodes in any path.

  However, by making G red, we might have created a red violation with G's parent. If so, we recursively apply the full logic to handle a red violation, where this G becomes the new N.

  Note that in the general recursive case, N, P, and U may also have subtrees in place of each black NULL (the leaves shown). In Case 1, these subtrees stay attached to the same parents, as the tree structure remains unchanged.

- **Case 2: U is black.**

  We'll need to consider the configurations (left vs. right child) of N and U. In each case, our goal is to fix up the red violation (red on top of red) without:

  » Messing up the ordering of the binary search tree.

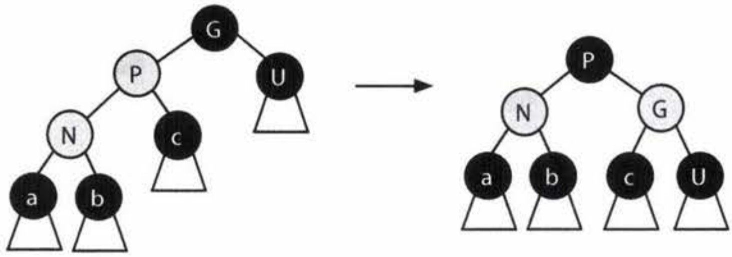
  » Introducing a black violation (more black nodes on one path than another).

  If we can do this, we're good. In each of the cases below, the red violation is fixed with rotations that maintain the node ordering.

  Further, the below rotations maintain the exact number of black nodes in each path through the affected portion of the tree that were in place beforehand. The children of the rotating section are either NULL leaves or subtrees that remain internally unchanged.

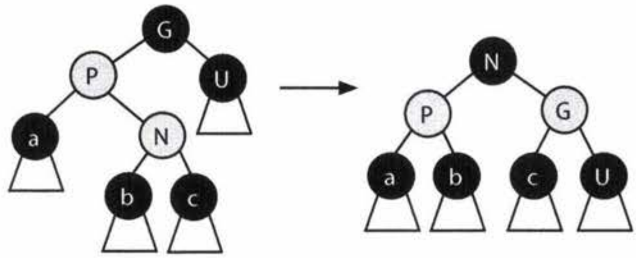
  Case A: N and P are both left children.

  We resolve the red violation with the rotation of N, P, and G and the associated recoloring shown below. If you picture the in-order traversal, you can see the rotation maintains the node ordering (a `<=` N `<=` b `<=` P `<=` c `<=` G `<=` U). The tree maintains the same, equal number of black nodes in the path down to each subtree a, b, c, and U (which may all be NULL) .



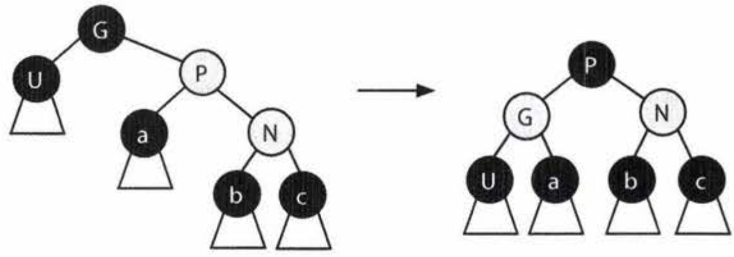
  Case B: P is a left child, and N is a right child.

  The rotations in Case B resolve the red violation and maintain the in-order property: a `<=` P `<=` b `<=` N `<=` c `<=` G `<=` U. Again, the count of the black nodes remains constant in each path down to the leaves (or subtrees).



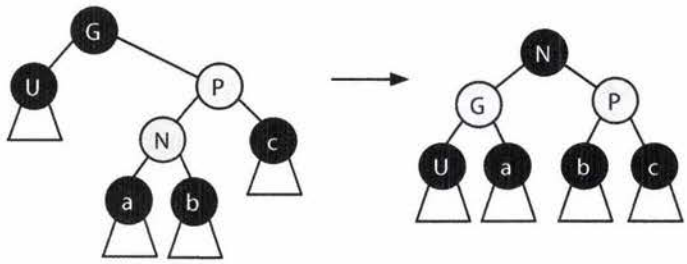
Case C: N and P are both right children.

This is a mirror image of case A.



Case D: N is a left child, and P is a right child.

This is a mirror image of case B.



In each of Case 2's subcases, the middle element by value of N, P, and G is rotated to become the root of what was G's subtree, and that element and G swap colors.

That said, do not try to just memorize these cases. Rather, study why they work. How does each one ensure no red violations, no black violations, and no violations of the binary search tree property?

▸ MapReduce

MapReduce is used widely in system design to process large amounts of data. As its name suggests, a MapReduce program requires you to write a Map step and a Reduce step. The rest is handled by the system.

1. The system splits up the data across different machines.
2. Each machine starts running the user-provided `Map` program.
3. The `Map` program takes some data and emits a `<key, value>` pair.
4. The system-provided `Shuffle` process reorganizes the data so that all <key, value> pairs associated with a given key go to the same machine, to be processed by `Reduce`.
5. The user-provided `Reduce` program takes a key and a set of associated values and "reduces" them in some way, emitting a new key and value. The results of this might be fed back into the `Reduce` program for more reducing.

The typical example of using MapReduce—basically the "Hello World" of MapReduce—is counting the frequency of words within a set of documents.

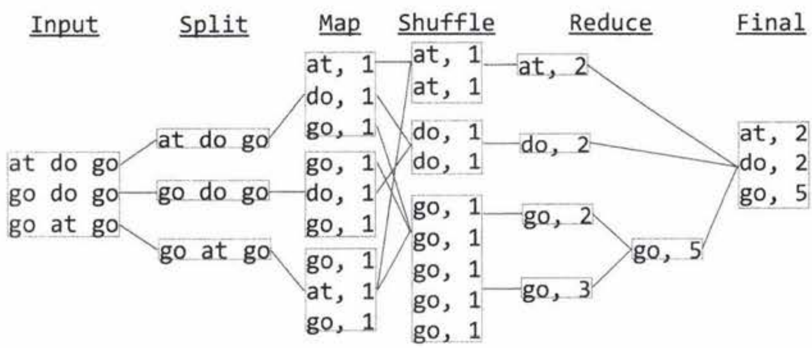
Of course, you could write this as a single function that reads in all the data, counts the number of times each word appears via a hash table, and then outputs the result.

MapReduce allows you to process the document in parallel. The Map function reads in a document and emits just each individual word and the count (which is always 1). The Reduce function reads in keys (words) and associated values (counts). It emits the sum of the counts. This sum could possibly wind up as input for another call to Reduce on the same key (as shown in the diagram).

```
void map(String name, String document):
  for each word w in document:
    emit(w, 1)

void reduce(String word, Iterator partialCounts):
  int sum = 0
  for each count in partialCounts:
    sum += count
  emit(word, sum)
```

The diagram below shows how this might work on this example.



Here's another example: You have a list of data in the form {City, Temperature, Date}. Calculate the average temperature in each city every year. For example {(2012, Philadelphia, 58.2), (2011, Philadelphia, 56.6), (2012, Seattle, 45.1)}.

- **Map**: The Map step outputs a key value pair where the key is `City_Year` and the value is `(Temperature, 1)`. The '1' reflects that this is the average temperature out of one data point. This will be important for the Reduce step.
- **Reduce**: The Reduce step will be given a list of temperatures that correspond with a particular city and year. It must use these to compute the average temperature for this input. You cannot simply add up the temperatures and divide by the number of values.

  To see this, imagine we have five data points for a particular city and year: 25, 100, 75, 85, 50. The Reduce step might only get some of this data at once. If you averaged {75, 85} you would get 80. This might end up being input for another Reduce step with 50, and it would be a mistake to just naively average 80 and 50. The 80 has more weight.

  Therefore, our Reduce step instead takes in {(80, 2), (50, 1)}, then sums the *weighted* temperatures. So it does 80 * 2 + 50 * 1 and then divides by (2 + 1) to get an average temperature of 70. It then emits (70, 3).

  Another Reduce step might reduce {(25, 1), (100, 1)} to get (62.5, 2). If we reduce this with (70, 3) we get the final answer: (67, 5). In other words, the average temperature in this city for this year was 67 degrees.

We could do this in other ways, too. We could have just the city as the key, and the value be (Year, Temperature, Count). The Reduce step would do essentially the same thing, but would have to group by Year itself.

In many cases, it's useful to think about what the Reduce step should do first, and then design the Map step around that. What data does Reduce need to have to do its job?

▸ Additional Studying

So, you've mastered this material and you want to learn even more? Okay. Here are some topics to get you started:

- **Bellman-Ford Algorithm**: Finds the shortest paths from a single node in a weighted directed graph with positive and negative edges.
- **Floyd-Warshall Algorithm**: Finds the shortest paths in a weighted graph with positive or negative weight edges (but no negative weight cycles).
- **Minimum Spanning Trees**: In a weighted, connected, undirected graph, a spanning tree is a tree that connects all the vertices. The minimum spanning tree is the spanning tree with minimum weight. There are various algorithms to do this.
- **B-Trees:** A self-balancing search tree (not a binary search tree) that is commonly used on disks or other storage devices. It is similar to a red-black tree, but uses fewer I/O operations.
- **A***: Find the least-cost path between a source node and a goal node (or one of several goal nodes). It extends Dijkstra's algorithm and achieves better performance by using heuristics.
- **Interval Trees**: An extension of a balanced binary search tree, but storing intervals (low -> high ranges) instead of simple values. A hotel could use this to store a list of all reservations and then efficiently detect who is staying at the hotel at a particular time.
- **Graph coloring**: A way of coloring the nodes in a graph such that no two adjacent vertices have the same color. There are various algorithms to do things like determine if a graph can be colored with only K colors.
- **P, NP, and NP-Complete**: P, NP, and NP-Complete refer to classes of problems. P problems are problems that can be quickly solved (where "quickly" means polynomial time). NP problems are those where, given a solution, the solution can be quickly verified. NP-Complete problems are a subset of NP problems that can all be reduced to each other (that is, if you found a solution to one problem, you could tweak the solution to solve other problems in the set in polynomial time).

  It is an open (and very famous) question whether P = NP, but the answer is generally believed to be no.
- **Combinatorics and Probability**: There are various things you can learn about here, such as random variables, expected value, and n-choose-k.
- **Bipartite Graph**: A bipartite graph is a graph where you can divide its nodes into two sets such that every edge stretches across the two sets (that is, there is never an edge between two nodes in the same set). There is an algorithm to check if a graph is a bipartite graph. Note that a bipartite graph is equivalent to a graph that can be colored with two colors.
- **Regular Expressions**: You should know that regular expressions exist and what they can be used for (roughly). You can also learn about how an algorithm to match regular expressions would work. Some of the basic syntax behind regular expressions could be useful as well.

There is of course a great deal more to data structures and algorithms. If you're interested in exploring these topics more deeply, I recommend picking up the hefty *Introduction to Algorithms* ("CLRS" by Cormen, Leiserson, Rivest and Stein) or *The Algorithm Design Manual* (by Steven Skiena).

Code Library

XII

Certain patterns came up while implementing the code for this book. We've tried to generally include the full code for a solution with the solution, but in some cases it got quite redundant.

This appendix provides the code for a few of the most useful chunks of code.

The complete compilable solutions can be downloaded from CrackingTheCodingInterview.com.

XI

Code Library

Certain patterns came up while implementing the code for this book. We've tried to generally include the full code for a solution with the solution, but in some cases it got quite redundant.

This appendix provides the code for a few of the most useful chunks of code.

All code for the book can be downloaded from CrackingTheCodingInterview.com.

▸ HashMapList<T, E>

The `HashMapList` class is essentially shorthand for `HashMap<T, ArrayList<E>>`. It allows us to map from an item of type of T to an `ArrayList` of type E.

For example, we might want a data structure that maps from an integer to a list of strings. Ordinarily, we'd have to write something like this:

```
HashMap<Integer, ArrayList<String>> maplist =
  new HashMap<Integer, ArrayList<String>>();
for (String s : strings) {
  int key = computeValue(s);
  if (!maplist.containsKey(key)) {
    maplist.put(key, new ArrayList<String>());
  }
  maplist.get(key).add(s);
}
```

Now, we can just write this:

```
HashMapList<Integer, String> maplist = new HashMapList<Integer, String>();
for (String s : strings) {
  int key = computeValue(s);
  maplist.put(key, s);
}
```

It's not a big change, but it makes our code a bit simpler.

```
public class HashMapList<T, E> {
  private HashMap<T, ArrayList<E>> map = new HashMap<T, ArrayList<E>>();

  /* Insert item into list at key. */
  public void put(T key, E item) {
    if (!map.containsKey(key)) {
      map.put(key, new ArrayList<E>());
    }
    map.get(key).add(item);
```

```
    }

    /* Insert list of items at key. */
    public void put(T key, ArrayList<E> items) {
      map.put(key, items);
    }

    /* Get list of items at key. */
    public ArrayList<E> get(T key) {
      return map.get(key);
    }

    /* Check if hashmaplist contains key. */
    public boolean containsKey(T key) {
      return map.containsKey(key);
    }

    /* Check if list at key contains value. */
    public boolean containsKeyValue(T key, E value) {
      ArrayList<E> list = get(key);
      if (list == null) return false;
      return list.contains(value);
    }

    /* Get the list of keys. */
    public Set<T> keySet() {
      return map.keySet();
    }

    @Override
    public String toString() {
      return map.toString();
    }
}
```

▸ TreeNode (Binary Search Tree)

While it's perfectly fine—even good—to use the built-in binary tree class when possible, it's not always possible. In many questions, we needed access to the internals of the node or tree class (or needed to tweak these) and thus couldn't use the built-in libraries.

The TreeNode class supports a variety of functionality, much of which we wouldn't necessarily want for every question/solution. For example, the TreeNode class tracks the parent of the node, even though we often don't use it (or specifically ban using it).

For simplicity, we'd implemented this tree as storing integers for data.

```
public class TreeNode {
    public int data;
    public TreeNode left, right, parent;
    private int size = 0;

    public TreeNode(int d) {
      data = d;
      size = 1;
    }
```

```
    public void insertInOrder(int d) {
        if (d <= data) {
            if (left == null) {
                setLeftChild(new TreeNode(d));
            } else {
                left.insertInOrder(d);
            }
        } else {
            if (right == null) {
                setRightChild(new TreeNode(d));
            } else {
                right.insertInOrder(d);
            }
        }
        size++;
    }

    public int size() {
        return size;
    }

    public TreeNode find(int d) {
        if (d == data) {
            return this;
        } else if (d <= data) {
            return left != null ? left.find(d) : null;
        } else if (d > data) {
            return right != null ? right.find(d) : null;
        }
        return null;
    }

    public void setLeftChild(TreeNode left) {
        this.left = left;
        if (left != null) {
            left.parent = this;
        }
    }

    public void setRightChild(TreeNode right) {
        this.right = right;
        if (right != null) {
            right.parent = this;
        }
    }

}
```

This tree is implemented to be a binary search tree. However, you can use it for other purposes. You would just need to use the `setLeftChild`/`setRightChild` methods, or the `left` and `right` child variables. For this reason, we have kept these methods and variables `public`. We need this sort of access for many problems.

▸ LinkedListNode (Linked List)

Like the TreeNode class, we often needed access to the internals of a linked list in a way that the built-in linked list class wouldn't support. For this reason, we implemented our own class and used it for many problems.

```
public class LinkedListNode {
   public LinkedListNode next, prev, last;
   public int data;
   public LinkedListNode(int d, LinkedListNode n, LinkedListNode p){
      data = d;
      setNext(n);
      setPrevious(p);
   }

   public LinkedListNode(int d) {
      data = d;
   }

   public LinkedListNode() { }

   public void setNext(LinkedListNode n) {
      next = n;
      if (this == last) {
         last = n;
      }
      if (n != null && n.prev != this) {
         n.setPrevious(this);
      }
   }

   public void setPrevious(LinkedListNode p) {
      prev = p;
      if (p != null && p.next != this) {
         p.setNext(this);
      }
   }

   public LinkedListNode clone() {
      LinkedListNode next2 = null;
      if (next != null) {
         next2 = next.clone();
      }
      LinkedListNode head2 = new LinkedListNode(data, next2, null);
      return head2;
   }
}
```

Again, we've kept the methods and variables public because we often needed this access. This would allow the user to "destroy" the linked list, but we actually needed this sort of functionality for our purposes.

▸ Trie & TrieNode

The trie data structure is used in a few problems to make it easier to look up if a word is a prefix of any other words in a dictionary (or list of valid words). This is often used when we're recursively building words so that we can short circuit when the word is not valid.

```java
public class Trie {
  // The root of this trie.
  private TrieNode root;

  /* Takes a list of strings as an argument, and constructs a trie that stores
   * these strings. */
  public Trie(ArrayList<String> list) {
    root = new TrieNode();
    for (String word : list) {
      root.addWord(word);
    }
  }


  /* Takes a list of strings as an argument, and constructs a trie that stores
   * these strings. */
  public Trie(String[] list) {
    root = new TrieNode();
    for (String word : list) {
      root.addWord(word);
    }
  }

  /* Checks whether this trie contains a string with the prefix passed in as
   * argument. */
  public boolean contains(String prefix, boolean exact) {
    TrieNode lastNode = root;
    int i = 0;
    for (i = 0; i < prefix.length(); i++) {
      lastNode = lastNode.getChild(prefix.charAt(i));
      if (lastNode == null) {
        return false;
      }
    }
    return !exact || lastNode.terminates();
  }

  public boolean contains(String prefix) {
    return contains(prefix, false);
  }

  public TrieNode getRoot() {
    return root;
  }
}
```

The Trie class uses the TrieNode class, which is implemented below.

```java
public class TrieNode {
  /* The children of this node in the trie.*/
  private HashMap<Character, TrieNode> children;
  private boolean terminates = false;

  /* The character stored in this node as data.*/
  private char character;

  /* Constructs an empty trie node and initializes the list of its children to an
   * empty hash map. Used only to construct the root node of the trie. */
```

```
    public TrieNode() {
      children = new HashMap<Character, TrieNode>();
    }

    /* Constructs a trie node and stores this character as the node's value.
     * Initializes the list of child nodes of this node to an empty hash map. */
    public TrieNode(char character) {
      this();
      this.character = character;
    }

    /* Returns the character data stored in this node. */
    public char getChar() {
      return character;
    }

    /* Add this word to the trie, and recursively create the child
     * nodes. */
    public void addWord(String word) {
      if (word == null || word.isEmpty()) {
        return;
      }

      char firstChar = word.charAt(0);

      TrieNode child = getChild(firstChar);
      if (child == null) {
        child = new TrieNode(firstChar);
        children.put(firstChar, child);
      }

      if (word.length() > 1) {
        child.addWord(word.substring(1));
      } else {
        child.setTerminates(true);
      }
    }

    /* Find a child node of this node that has the char argument as its data. Return
     * null if no such child node is present in the trie. */
    public TrieNode getChild(char c) {
      return children.get(c);
    }

    /* Returns whether this node represents the end of a complete word. */
    public boolean terminates() {
      return terminates;
    }

    /* Set whether this node is the end of a complete word.*/
    public void setTerminates(boolean t) {
      terminates = t;
    }
}
```

Hints

XIII

Interviewers usually don't just hand you a question and expect you to solve it. Rather, they will typically offer guidance when you're stuck, especially on the harder questions. It's impossible to totally simulate the interview experience in a book, but these hints are designed to get you closer.

Try to solve the questions independently when possible. But it's okay to look for some help when you are really struggling. Again, struggling is a normal part of the process.

I've organized the hints somewhat randomly here, such that all the hints for a problem aren't adjacent. This way you won't accidentally see the second hint when you're reading the first hint.

I

Hints for Data Structures

#1.	1.2	Describe what it means for two strings to be permutations of each other. Now, look at that definition you provided. Can you check the strings against that definition?
#2.	3.1	A stack is simply a data structure in which the most recently added elements are removed first. Can you simulate a single stack using an array? Remember that there are many possible solutions, and there are tradeoffs of each.
#3.	2.4	There are many solutions to this problem, most of which are equally optimal in runtime. Some have shorter, cleaner code than others. Can you brainstorm different solutions?
#4.	4.10	If T2 is a subtree of T1, how will its in-order traversal compare to T1's? What about its pre-order and post-order traversal?
#5.	2.6	A palindrome is something which is the same when written forwards and backwards. What if you reversed the linked list?
#6.	4.12	Try simplifying the problem. What if the path had to start at the root?
#7.	2.5	Of course, you could convert the linked lists to integers, compute the sum, and then convert it back to a new linked list. If you did this in an interview, your interviewer would likely accept the answer, and then see if you could do this without converting it to a number and back.
#8.	2.2	What if you knew the linked list size? What is the difference between finding the Kth-to-last element and finding the Xth element?
#9.	2.1	Have you tried a hash table? You should be able to do this in a single pass of the linked list.
#10.	4.8	If each node has a link to its parent, we could leverage the approach from question 2.7 on page 95. However, our interviewer might not let us make this assumption.
#11.	4.10	The in-order traversals won't tell us much. After all, every binary search tree with the same values (regardless of structure) will have the same in-order traversal. This is what in-order traversal means: contents are in-order. (And if it won't work in the specific case of a binary search tree, then it certainly won't work for a general binary tree.) The pre-order traversal, however, is much more indicative.
#12.	3.1	We could simulate three stacks in an array by just allocating the first third of the array to the first stack, the second third to the second stack, and the final third to the third stack. One might actually be much bigger than the others, though. Can we be more flexible with the divisions?

#13. 2.6 Try using a stack.

#14. 4.12 Don't forget that paths could overlap. For example, if you're looking for the sum 6, the paths `1->3->2` and `1->3->2->4->-6->2` are both valid.

#15. 3.5 One way of sorting an array is to iterate through the array and insert each element into a new array in sorted order. Can you do this with a stack?

#16. 4.8 The first common ancestor is the deepest node such that p and q are both descendants. Think about how you might identify this node.

#17. 1.8 If you just cleared the rows and columns as you found 0s, you'd likely wind up clearing the whole matrix. Try finding the cells with zeros first before making any changes to the matrix.

#18. 4.10 You may have concluded that if `T2.preorderTraversal()` is a substring of `T1.preorderTraversal()`, then T2 is a subtree of T1. This is almost true, except that the trees could have duplicate values. Suppose T1 and T2 have all duplicate values but different structures. The pre-order traversals will look the same even though T2 is not a subtree of T1. How can you handle situations like this?

#19. 4.2 A minimal binary tree has about the same number of nodes on the left of each node as on the right. Let's focus on just the root for now. How would you ensure that about the same number of nodes are on the left of the root as on the right?

#20. 2.7 You can do this in `O(A+B)` time and `O(1)` additional space. That is, you do not need a hash table (although you could do it with one).

#21. 4.4 Think about the definition of a balanced tree. Can you check that condition for a single node? Can you check it for every node?

#22. 3.6 We could consider keeping a single linked list for dogs and cats, and then iterating through it to find the first dog (or cat). What is the impact of doing this?

#23. 1.5 Start with the easy thing. Can you check each of the conditions separately?

#24. 2.4 Consider that the elements don't have to stay in the same relative order. We only need to ensure that elements less than the pivot must be before elements greater than the pivot. Does that help you come up with more solutions?

#25. 2.2 If you don't know the linked list size, can you compute it? How does this impact the runtime?

#26. 4.7 Build a directed graph representing the dependencies. Each node is a project and an edge exists from A to B if B depends on A (A must be built before B). You can also build it the other way if it's easier for you.

#27. 3.2 Observe that the minimum element doesn't change very often. It only changes when a smaller element is added, or when the smallest element is popped.

#28. 4.8 How would you figure out if p is a descendent of a node n?

#29. 2.6 Assume you have the length of the linked list. Can you implement this recursively?

#30. 2.5 Try recursion. Suppose you have two lists, `A = 1->5->9` (representing 951) and `B = 2->3->6->7` (representing 7632), and a function that operates on the remainder of the lists (`5->9` and `3->6->7`). Could you use this to create the `sum` method? What is the relationship between `sum(1->5->9, 2->3->6->7)` and `sum(5->9, 3->6->7)`?

#31. 4.10 Although the problem seems like it stems from duplicate values, it's really deeper than that. The issue is that the pre-order traversal is the same only because there are null nodes that we skipped over (because they're null). Consider inserting a placeholder value into the pre-order traversal string whenever you reach a null node. Register the null node as a "real" node so that you can distinguish between the different structures.

#32. 3.5 Imagine your secondary stack is sorted. Can you insert elements into it in sorted order? You might need some extra storage. What could you use for extra storage?

#33. 4.4 If you've developed a brute force solution, be careful about its runtime. If you are computing the height of the subtrees for each node, you could have a pretty inefficient algorithm.

#34. 1.9 If a string is a rotation of another, then it's a rotation at a particular point. For example, a rotation of `waterbottle` at character 3 means cutting `waterbottle` at character 3 and putting the right half (`erbottle`) before the left half (`wat`).

#35. 4.5 If you traversed the tree using an in-order traversal and the elements were truly in the right order, does this indicate that the tree is actually in order? What happens for duplicate elements? If duplicate elements are allowed, they must be on a specific side (usually the left).

#36. 4.8 Start with the root. Can you identify if root is the first common ancestor? If it is not, can you identify which side of root the first common ancestor is on?

#37. 4.10 Alternatively, we can handle this problem recursively. Given a specific node within `T1`, can we check to see if its subtree matches `T2`?

#38. 3.1 If you want to allow for flexible divisions, you can shift stacks around. Can you ensure that all available capacity is used?

#39. 4.9 What is the very first value that must be in each array?

#40. 2.1 Without extra space, you'll need $O(N^2)$ time. Try using two pointers, where the second one searches ahead of the first one.

#41. 2.2 Try implementing it recursively. If you could find the (`K-1`)th to last element, can you find the Kth element?

#42. 4.11 Be very careful in this problem to ensure that each node is equally likely and that your solution doesn't slow down the speed of standard binary search tree algorithms (like `insert`, `find`, and `delete`). Also, remember that even if you assume that it's a balanced binary search tree, this doesn't mean that the tree is full/complete/perfect.

#43. 3.5 Keep the secondary stack in sorted order, with the biggest elements on the top. Use the primary stack for additional storage.

#44. 1.1 Try a hash table.

#45. 2.7 Examples will help you. Draw a picture of intersecting linked lists and two equivalent linked lists (by value) that do not intersect.

#46. 4.8 Try a recursive approach. Check if p and q are descendants of the left subtree and the right subtree. If they are descendants of different subtrees, then the current node is the first common ancestor. If they are descendants of the same subtree, then that subtree holds the first common ancestor. Now, how do you implement this efficiently?

#47. 4.7 Look at this graph. Is there any node you can identify that will definitely be okay to build first?

#48. 4.9 The root is the very first value that must be in every array. What can you say about the order of the values in the left subtree as compared to the values in the right subtree? Do the left subtree values need to be inserted before the right subtree?

#49. 4.4 What if you could modify the binary tree node class to allow a node to store the height of its subtree?

#50. 2.8 There are really two parts to this problem. First, detect if the linked list has a loop. Second, figure out where the loop starts.

#51. 1.7 Try thinking about it layer by layer. Can you rotate a specific layer?

#52. 4.12 If each path had to start at the root, we could traverse all possible paths starting from the root. We can track the sum as we go, incrementing `totalPaths` each time we find a path with our target sum. Now, how do we extend this to paths that can start anywhere? Remember: Just get a brute-force algorithm done. You can optimize later.

#53. 1.3 It's often easiest to modify strings by going from the end of the string to the beginning.

#54. 4.11 This is your own binary search tree class, so you can maintain any information about the tree structure or nodes that you'd like (provided it doesn't have other negative implications, like making `insert` much slower). In fact, there's probably a reason the interview question specified that it was your own class. You probably need to store some additional information in order to implement this efficiently.

#55. 2.7 Focus first on just identifying if there's an intersection.

#56. 3.6 Let's suppose we kept separate lists for dogs and cats. How would we find the oldest animal of any type? Be creative!

#57. 4.5 To be a binary search tree, it's not sufficient that the `left.value <= current.value < right.value` for each node. Every node on the left must be less than the current node, which must be less than all the nodes on the right.

#58. 3.1 Try thinking about the array as circular, such that the end of the array "wraps around" to the start of the array.

#59. 3.2 What if we kept track of extra data at each stack node? What sort of data might make it easier to solve the problem?

#60. 4.7 If you identify a node without any incoming edges, then it can definitely be built. Find this node (there could be multiple) and add it to the build order. Then, what does this mean for its outgoing edges?

#61. 2.6 In the recursive approach (we have the length of the list), the middle is the base case: `isPermutation(middle)` is true. The node `x` to the immediate left of the middle: What can that node do to check if `x->middle->y` forms a palindrome? Now suppose that checks out. What about the previous node `a`? If `x->middle->y` is a palindrome, how can it check that `a->x->middle->y->b` is a palindrome?

#62. 4.11 As a naive "brute force" algorithm, can you use a tree traversal algorithm to implement this algorithm? What is the runtime of this?

#63. 3.6 Think about how you'd do it in real life. You have a list of dogs in chronological order and a list of cats in chronological order. What data would you need to find the oldest animal? How would you maintain this data?

#64. 3.3 You will need to keep track of the size of each substack. When one stack is full, you may need to create a new stack.

#65. 2.7 Observe that two intersecting linked lists will always have the same last node. Once they intersect, all the nodes after that will be equal.

#66. 4.9 The relationship between the left subtree values and the right subtree values is, essentially, anything. The left subtree values could be inserted before the right subtree, or the reverse (right values before left), or any other ordering.

#67. 2.2 You might find it useful to return multiple values. Some languages don't directly support this, but there are workarounds in essentially any language. What are some of those workarounds?

#68. 4.12 To extend this to paths that start anywhere, we can just repeat this process for all nodes.

#69. 2.8 To identify if there's a cycle, try the "runner" approach described on page 93. Have one pointer move faster than the other.

#70. 4.8 In the more naive algorithm, we had one method that indicated if `x` is a descendent of `n`, and another method that would recurse to find the first common ancestor. This is repeatedly searching the same elements in a subtree. We should merge this into one `firstCommonAncestor` function. What return values would give us the information we need?

#71. 2.5 Make sure you have considered linked lists that are not the same length.

#72. 2.3 Picture the list `1->5->9->12`. Removing 9 would make it look like `1->5->12`. You only have access to the 9 node. Can you make it look like the correct answer?

#73. 4.2 You could implement this by finding the "ideal" next element to add and repeatedly calling `insertValue`. This will be a bit inefficient, as you would have to repeatedly traverse the tree. Try recursion instead. Can you divide this problem into subproblems?

#74. 1.8 Can you use $O(N)$ additional space instead of $O(N^2)$? What information do you really need from the list of cells that are zero?

#75. 4.11 Alternatively, you could pick a random depth to traverse to and then randomly traverse, stopping when you get to that depth. Think this through, though. Does this work?

#76. 2.7 You can determine if two linked lists intersect by traversing to the end of each and comparing their tails.

#77. 4.12 If you've designed the algorithm as described thus far, you'll have an $O(N \log N)$ algorithm in a balanced tree. This is because there are N nodes, each of which is at depth $O(\log N)$ at worst. A node is touched once for each node above it. Therefore, the N nodes will be touched $O(\log N)$ time. There is an optimization that will give us an $O(N)$ algorithm.

#78. 3.2 Consider having each node know the minimum of its "substack" (all the elements beneath it, including itself).

#79. 4.6 Think about how an in-order traversal works and try to "reverse engineer" it.

#80. 4.8 The `firstCommonAncestor` function could return the first common ancestor (if p and q are both contained in the tree), p if p is in the tree and not q, q if q is in the tree and not p, and `null` otherwise.

#81. 3.3 Popping an element at a specific substack will mean that some stacks aren't at full capacity. Is this an issue? There's no right answer, but you should think about how to handle this.

#82. 4.9 Break this down into subproblems. Use recursion. If you had all possible sequences for the left subtree and the right subtree, how could you create all possible sequences for the entire tree?

#83. 2.8 You can use two pointers, one moving twice as fast as the other. If there is a cycle, the two pointers will collide. They will land at the same location at the same time. Where do they land? Why there?

#84. 1.2 There is one solution that is `O(N log N)` time. Another solution uses some space, but is `O(N)` time.

#85. 4.7 Once you decide to build a node, its outgoing edge can be deleted. After you've done this, can you find other nodes that are free and clear to build?

#86. 4.5 If every node on the left must be less than or equal to the current node, then this is really the same thing as saying that the biggest node on the left must be less than or equal to the current node.

#87. 4.12 What work is duplicated in the current brute-force algorithm?

#88. 1.9 We are essentially asking if there's a way of splitting the first string into two parts, x and y, such that the first string is `xy` and the second string is `yx`. For example, `x = wat` and `y = erbottle`. The first string is `xy = waterbottle`. The second string is `yx = erbottlewat`.

#89. 4.11 Picking a random depth won't help us much. First, there's more nodes at lower depths than higher depths. Second, even if we re-balanced these probabilities, we could hit a "dead end" where we meant to pick a node at depth 5 but hit a leaf at depth 3. Re-balancing the probabilities is an interesting , though.

#90. 2.8 If you haven't identified the pattern of where the two pointers start, try this: Use the linked list 1->2->3->4->5->6->7->8->9->?, where the ? links to another node. Try making the ? the first node (that is, the 9 points to the 1 such that the entire linked list is a loop). Then make the ? the node 2. Then the node 3. Then the node 4. What is the pattern? Can you explain why this happens?

#91. 4.6 Here's one step of the logic: The successor of a specific node is the leftmost node of the right subtree. What if there is no right subtree, though?

#92. 1.6 Do the easy thing first. Compress the string, then compare the lengths.

#93. 2.7 Now, you need to find where the linked lists intersect. Suppose the linked lists were the same length. How could you do this?

#94. 4.12 Consider each path that starts from the root (there are N such paths) as an array. What our brute-force algorithm is really doing is taking each array and finding all contiguous subsequences that have a particular sum. We're doing this by computing all subarrays and their sums. It might be useful to just focus on this little subproblem. Given an array, how would you find all contiguous subsequences with a particular sum? Again, think about the duplicated work in the brute-force algorithm.

#95. 2.5 Does your algorithm work on linked lists like 9->7->8 and 6->8->5? Double check that.

#96. 4.8 Careful! Does your algorithm handle the case where only one node exists? What will happen? You might need to tweak the return values a bit.

#97. 1.5 What is the relationship between the "insert character" option and the "remove character" option? Do these need to be two separate checks?

#98. 3.4 The major difference between a queue and a stack is the order of elements. A queue removes the oldest item and a stack removes the newest item. How could you remove the oldest item from a stack if you only had access to the newest item?

#99. 4.11 A naive approach that many people come up with is to pick a random number between 1 and 3. If it's 1, return the current node. If it's 2, branch left. If it's 3, branch right. This solution doesn't work. Why not? Is there a way you can adjust it to make it work?

#100. 1.7 Rotating a specific layer would just mean swapping the values in four arrays. If you were asked to swap the values in two arrays, could you do this? Can you then extend it to four arrays?

#101. 2.6 Go back to the previous hint. Remember: There are ways to return multiple values. You can do this with a new class.

#102. 1.8 You probably need some data storage to maintain a list of the rows and columns that need to be zeroed. Can you reduce the additional space usage to `O(1)` by using the matrix itself for data storage?

#103. 4.12 We are looking for subarrays with sum `targetSum`. Observe that we can track in constant time the value of $\texttt{runningSum}_i$, where this is the sum from element 0 through element `i`. For a subarray of element `i` through element `j` to have sum `targetSum`, $\texttt{runningSum}_{i-1}$ `+ targetSum` must equal $\texttt{runningSum}_j$ (try drawing a picture of an array or a number line). Given that we can track the `runningSum` as we go, how can we quickly look up the number of indices `i` where the previous equation is true?

#104. 1.9 Think about the earlier hint. Then think about what happens when you concatenate `erbottlewat` to itself. You get `erbottlewaterbottlewat`.

#105. 4.4 You don't need to modify the binary tree class to store the height of the subtree. Can your recursive function compute the height of each subtree while also checking if a node is balanced? Try having the function return multiple values.

#106. 1.4 You do not have to—and should not—generate all permutations. This would be very inefficient.

#107. 4.3 Try modifying a graph search algorithm to track the depth from the root.

#108. 4.12 Try using a hash table that maps from a `runningSum` value to the number of elements with this `runningSum`.

#109. 2.5 For the follow-up question: The issue is that when the linked lists aren't the same length, the head of one linked list might represent the 1000's place while the other represents the 10's place. What if you made them the same length? Is there a way to modify the linked list to do that, without changing the value it represents?

#110. 1.6 Be careful that you aren't repeatedly concatenating strings together. This can be very inefficient.

#111. 2.7 If the two linked lists were the same length, you could traverse forward in each until you found an element in common. Now, how do you adjust this for lists of different lengths?

#112. 4.11 The reason that the earlier solution (picking a random number between 1 and 3) doesn't work is that the probabilities for the nodes won't be equal. For example, the root will be returned with probability $\frac{1}{3}$, even if there are 50+ nodes in the tree. Clearly, not all the nodes have probability $\frac{1}{3}$, so these nodes won't have equal probability. We can resolve this one issue by picking a random number between 1 and `size_of_tree` instead. This only resolves the issue for the root, though. What about the rest of the nodes?

#113. 4.5 Rather than validating the current node's value against `leftTree.max` and `rightTree.min`, can we flip around the logic? Validate the left tree's nodes to ensure that they are smaller than `current.value`.

#114. 3.4 We can remove the oldest item from a stack by repeatedly removing the newest item (inserting those into the temporary stack) until we get down to one element. Then, after we've retrieved the newest item, putting all the elements back. The issue with this is that doing several pops in a row will require `O(N)` work each time. Can we optimize for scenarios where we might do several pops in a row?

#115. 4.12 Once you've solidified the algorithm to find all contiguous subarrays in an array with a given sum, try to apply this to a tree. Remember that as you're traversing and modifying the hash table, you may need to "reverse the damage" to the hash table as you traverse back up.

#116. 4.2 Imagine we had a `createMinimalTree` method that returns a minimal tree for a given array (but for some strange reason doesn't operate on the root of the tree). Could you use this to operate on the root of the tree? Could you write the base case for the function? Great! Then that's basically the entire function.

#117. 1.1 Could a bit vector be useful?

#118. 1.3 You might find you need to know the number of spaces. Can you just count them?

#119. 4.11 The issue with the earlier solution is that there could be more nodes on one side of a node than the other. So, we need to weight the probability of going left and right based on the number of nodes on each side. How does this work, exactly? How can we know the number of nodes?

#120. 2.7 Try using the difference between the lengths of the two linked lists.

#121. 1.4 What characteristics would a string that is a permutation of a palindrome have?

#122. 1.2 Could a hash table be useful?

#123. 4.3 A hash table or array that maps from level number to nodes at that level might also be useful.

#124.	4.4	Actually, you can just have a single `checkHeight` function that does both the height computation and the balance check. An integer return value can be used to indicate both.
#125.	4.7	As a totally different approach: Consider doing a depth-first search starting from an arbitrary node. What is the relationship between this depth-first search and a valid build order?
#126.	2.2	Can you do it iteratively? Imagine if you had two pointers pointing to adjacent nodes and they were moving at the same speed through the linked list. When one hits the end of the linked list, where will the other be?
#127.	4.1	Two well-known algorithms can do this. What are the tradeoffs between them?
#128.	4.5	Think about the `checkBST` function as a recursive function that ensures each node is within an allowable `(min, max)` range. At first, this range is infinite. When we traverse to the left, the min is negative infinity and the max is `root.value`. Can you implement this recursive function and properly adjust these ranges as you traverse the tree?
#129.	2.7	If you move a pointer in the longer linked list forward by the difference in lengths, you can then apply a similar approach to the scenario when the linked lists are equal.
#130.	1.5	Can you do all three checks in a single pass?
#131.	1.2	Two strings that are permutations should have the same characters, but in different orders. Can you make the orders the same?
#132.	1.1	Can you solve it in `O(N log N)` time? What might a solution like that look like?
#133.	4.7	Pick an arbitrary node and do a depth-first search on it. Once we get to the end of a path, we know that this node can be the last one built, since no nodes depend on it. What does this mean about the nodes right before it?
#134.	1.4	Have you tried a hash table? You should be able to get this down to `O(N)` time.
#135.	4.3	You should be able to come up with an algorithm involving both depth-first search and breadth-first search.
#136.	1.4	Can you reduce the space usage by using a bit vector?

II

Hints for Concepts and Algorithms

#137.	5.1	Break this into parts. Focus first on clearing the appropriate bits.
#138.	8.9	Try the Base Case and Build approach.
#139.	6.9	Given a specific door x, on which rounds will it be toggled (open or closed)?
#140.	11.5	What does the interviewer mean by a pen? There are a lot of different types of pens. Make a list of potential questions you would want to ask.
#141.	7.11	This is not as complicated as it sounds. Start by making a list of the key objects in the system, then think about how they interact.
#142.	9.6	First, start with making some assumptions. What do and don't you have to build?
#143.	5.2	To wrap your head around the problem, try thinking about how you'd do it for integers.
#144.	8.6	Try the Base Case and Build approach.
#145.	5.7	Swapping each pair means moving the even bits to the left and the odd bits to the right. Can you break this problem into parts?
#146.	6.10	Solution 1: Start with a simple approach. Can you just divide up the bottles into groups? Remember that you can't re-use a test strip once it is positive, but you can reuse it as long as it's negative.
#147.	5.4	Get Next: Start with a brute force solution for each.
#148.	8.14	Can we just try all possibilities? What would this look like?
#149.	6.5	Play around with the jugs of water, pouring water back and forth, and see if you can measure anything other than 3 quarts or 5 quarts. That's a start.
#150.	8.7	Approach 1: Suppose you had all permutations of `abc`. How can you use that to get all permutations of `abcd`?
#151.	5.5	Reverse engineer this, starting from the outermost layer to the innermost layer.
#152.	8.1	Approach this from the top down. What is the very last hop the child made?
#153.	7.1	Note that a "card deck" is very broad. You might want to think about a reasonable scope to the problem.
#154.	6.7	Observe that each family will have exactly one girl.
#155.	8.13	Will sorting the boxes help in any way?

#156. 6.8 This is really an algorithm problem, and you should approach it as such. Come up with a brute force, compute the worst-case number of drops, then try to optimize that.

#157. 6.4 In what cases will they not collide?

#158. 9.6 We've assumed that the rest of the eCommerce system is already handled, and we just need to deal with the analytics part of sales rank. We can get notified somehow when a purchase occurs.

#159. 5.3 Start with a brute force solution. Can you try all possibilities?

#160. 6.7 Think about writing each family as a sequence of Bs and Gs.

#161. 8.8 You could handle this by just checking to see if there are duplicates before printing them (or adding them to a list). You can do this with a hash table. In what case might this be okay? In what case might it not be a very good solution?

#162. 9.7 Will this application be write-heavy or read-heavy?

#163. 6.10 Solution 1: There is a relatively simple approach that works in 28 days, in the worst case. There are better approaches though.

#164. 11.5 Consider the scenario of a pen for children. What does this mean? What are the different use cases?

#165. 9.8 Scope the problem well. What will and won't you tackle as part of this system?

#166. 8.5 Think about multiplying 8 by 9 as counting the number of cells in a matrix with width 8 and height 9.

#167. 5.2 In a number like `.893` (in base 10), what does each digit signify? What then does each digit in `.10010` signify in base 2?

#168. 8.14 We can think about each possibility as each place where we can put parentheses. This means around each operator, such that the expression is split at the operator. What is the base case?

#169. 5.1 To clear the bits, create a "bit mask" that looks like a series of 1s, then 0s, then 1s.

#170. 8.3 Start with a brute force algorithm.

#171. 6.7 You can attempt this mathematically, although the math is pretty difficult. You might find it easier to estimate it up to families of, say, 6 children. This won't give you a good mathematical proof, but it might point you in the right direction of what the answer might be.

#172. 6.9 In which cases would a door be left open at the end of the process?

#173. 5.2 A number such as `.893` (in base 10) indicates $8 * 10^{-1} + 9 * 10^{-2} + 3 * 10^{-3}$. Translate this system into base 2.

#174. 8.9 Suppose we had all valid ways of writing two pairs of parentheses. How could we use this to get all valid ways of writing three pairs?

#175. 5.4 Get Next: Picture a binary number—something with a bunch of 1s and 0s spread out throughout the number. Suppose you flip a 1 to a 0 and a 0 to a 1. In what case will the number get bigger? In what case will it get smaller?

#176. 9.6 Think about what sort of expectations on freshness and accuracy of data is expected. Does the data always need to be 100% up to date? Is the accuracy of some products more important than others?

#177. 10.2 How do you check if two words are anagrams of each other? Think about what the definition of "anagram" is. Explain it in your own words.

#178. 8.1 If we knew the number of paths to each of the steps before step 100, could we compute the number of steps to 100?

#179. 7.8 Should white pieces and black pieces be the same class? What are the pros and cons of this?

#180. 9.7 Observe that there is a lot of data coming in, but people probably aren't reading the data very frequently.

#181. 6.2 Calculate the probability of winning the first game and winning the second game, then compare them.

#182. 10.2 Two words are anagrams if they contain the same characters but in different orders. How can you put characters in order?

#183. 6.10 Solution 2: Why do we have such a time lag between tests and results? There's a reason the question isn't phrased as just "minimize the number of rounds of testing." The time lag is there for a reason.

#184. 9.8 How evenly do you think traffic is distributed? Do all documents get roughly the same age of traffic? Or is it likely there are some very popular documents?

#185. 8.7 Approach 1: The permutations of `abc` represent all ways of ordering `abc`. Now, we want to create all orderings of `abcd`. Take a specific ordering of `abcd`, such as `bdca`. This `bdca` string represents an ordering of `abc`, too: Remove the d and you get `bca`. Given the string `bca`, can you create all the "related" orderings that include d, too?

#186. 6.1 You can only use the scale once. This means that all, or almost all, of the bottles must be used. They also must be handled in different ways or else you couldn't distinguish between them.

#187. 8.9 We could try generating the solution for three pairs by taking the list of two pairs of parentheses and adding a third pair. We'd have to add the third paren before, around, and after. That is: `()<SOLUTION>`, `(<SOLUTION>)`, `<SOLUTION>()`. Will this work?

#188. 6.7 Logic might be easier than math. Imagine we wrote every birth into a giant string of Bs and Gs. Note that the groupings of families are irrelevant for this problem. What is the probability of the next character added to the string being a B versus a G?

#189. 9.6 Purchases will occur very frequently. You probably want to limit database writes.

#190. 8.8 If you haven't solved 8.7 yet, do that one first.

#191. 6.10 Solution 2: Consider running multiple tests at once.

#192. 7.6 A common trick when solving a jigsaw puzzle is to separate edge and non-edge pieces. How will you represent this in an object-oriented manner?

#193. 10.9 Start with a naive solution. (But hopefully not too naive. You should be able to use the fact that the matrix is sorted.)

#194.	8.13	We can sort the boxes by any dimension in descending order. This will give us a partial order for the boxes, in that boxes later in the array must appear before boxes earlier in the array.
#195.	6.4	The only way they won't collide is if all three are walking in the same direction. What's the probability of all three walking clockwise?
#196.	10.11	Imagine the array were sorted in ascending order. Is there any way you could "fix it" to be sorted into alternating peaks and valleys?
#197.	8.14	The base case is when we have a single value, 1 or 0.
#198.	7.3	Scope the problem first and make a list of your assumptions. It's often okay to make reasonable assumptions, but you need to make them explicit.
#199.	9.7	The system will be write-heavy: Lots of data being imported, but it's rarely being read.
#200.	8.7	Approach 1: Given a string such as `bca`, you can create all permutations of `abcd` that have `{a, b, c}` in the order `bca` by inserting `d` into each possible location: `dbca`, `bdca`, `bcda`, `bcad`. Given all permutations of `abc`, can you then create all permutations of `abcd`?
#201.	6.7	Observe that biology hasn't changed; only the conditions under which a family stops having kids has changed. Each pregnancy has a 50% odds of being a boy and a 50% odds of being a girl.
#202.	5.5	What does it mean if `A & B == 0`?
#203.	8.5	If you wanted to count the cells in an 8x9 matrix, you could count the cells in a 4x9 matrix and then double it.
#204.	8.3	Your brute force algorithm probably ran in `O(N)` time. If you're trying to beat that runtime, what runtime do you think you will get to? What sorts of algorithms have that runtime?
#205.	6.10	Solution 2: Think about trying to figure out the bottle, digit by digit. How can you detect the first digit in the poisoned bottle? What about the second digit? The third digit?
#206.	9.8	How will you handle generating URLs?
#207.	10.6	Think about merge sort versus quick sort. Would one of them work well for this purpose?
#208.	9.6	You also want to limit joins because they can be very expensive.
#209.	8.9	The problem with the solution suggested by the earlier hint is that it might have duplicate values. We could eliminate this by using a hash table.
#210.	11.6	Be careful about your assumptions. Who are the users? Where are they using this? It might seem obvious, but the real answer might be different.
#211.	10.9	We can do a binary search in each row. How long will this take? How can we do better?
#212.	9.7	Think about things like how you're going to get the bank data (will it be pulled or pushed?), what features the system will support, etc.
#213.	7.7	As always, scope the problem. Are "friendships" mutual? Do status messages exist? Do you support group chat?
#214.	8.13	Try to break it down into subproblems.

#215.	5.1	It's easy to create a bit mask of 0s at the beginning or end. But how do you create a bit mask with a bunch of zeroes in the middle? Do it the easy way: Create a bit mask for the left side and then another one for the right side. Then you can merge those.
#216.	7.11	What is the relationship between files and directories?
#217.	8.1	We can compute the number of steps to 100 by the number of steps to 99, 98, and 97. This corresponds to the child hopping 1, 2, or 3 steps at the end. Do we add those or multiply them? That is: Is it `f(100) = f(99) + f(98) + f(97)` *or* `f(100) = f(99) * f(98) * f(97)`?
#218.	6.6	This is a logic problem, not a clever word problem. Use logic/math/algorithms to solve it.
#219.	10.11	Try walking through a sorted array. Can you just swap elements until you have fixed the array?
#220.	11.5	Have you considered both intended uses (writing, etc.) and unintended use? What about safety? You would not want a pen for children to be dangerous.
#221.	6.10	Solution 2: Be very careful about edge cases. What if the third digit in the bottle number matches the first or second digit?
#222.	8.8	Try getting the count of each character. For example, ABCAAC has 3 As, 2 Cs, and 1 B.
#223.	9.6	Don't forget that a product can be listed under multiple categories.
#224.	8.6	You can easily move the smallest disk from one tower to another. It's also pretty easy to move the smallest two disks from one tower to another. Can you move the smallest three disks?
#225.	11.6	In a real interview, you would also want to discuss what sorts of test tools we have available.
#226.	5.3	Flipping a 0 to a 1 can merge two sequences of 1s—but only if the two sequences are separated by only one 0.
#227.	8.5	Think about how you might handle this for odd numbers.
#228.	7.8	What class should maintain the score?
#229.	10.9	If you're considering a particular column, is there a way to quickly eliminate it (in some cases at least)?
#230.	6.10	Solution 2: You can run an additional day of testing to check digit 3 in a different way. But again, be very careful about edge cases here.
#231.	10.11	Note that if you ensure the peaks are in place, the valleys will be, too. Therefore, your iteration to fix the array can skip over every other element.
#232.	9.8	If you generate URLs randomly, do you need to worry about collisions (two documents with the same URL)? If so, how can you handle this?
#233.	6.8	As a first approach, you might try something like binary search. Drop it from the 50th floor, then the 75th, then the 88th, and so on. The problem is that if the first egg drops at the 50th floor, then you'll need to start dropping the second egg starting from the 1st floor and going up. This could take, at worst, 50 drops (the 50th floor drop, the 1st floor drop, the 2nd floor drop, and up through the 49th floor drop). Can you beat this?

#234. 8.5 If there's duplicated work across different recursive calls, can you cache it?

#235. 10.7 Would a bit vector help?

#236. 9.6 Where would it be appropriate to cache data or queue up tasks?

#237. 8.1 We multiply the values when it's "we do this then this." We add them when it's "we do this or this."

#238. 7.6 Think about how you might record the position of a piece when you find it. Should it be stored by row and location?

#239. 6.2 To calculate the probability of winning the second game, start with calculating the probability of making the first hoop, the second hoop, and not the third hoop.

#240. 8.3 Can you solve the problem in O(log N)?

#241. 6.10 Solution 3: Think about each test strip as being a binary indicator for poisoned vs. non-poisoned.

#242. 5.4 Get Next: If you flip a 1 to a 0 and a 0 to a 1, it will get bigger if the 0->1 bit is more significant than the 1->0 bit. How can you use this to create the next biggest number (with the same number of 1s)?

#243. 8.9 Alternatively, we could think about doing this by moving through the string and adding left and right parens at each step. Will this eliminate duplicates? How do we know if we can add a left or right paren?

#244. 9.6 Depending on what assumptions you made, you might even be able to do without a database at all. What would this mean? Would it be a good idea?

#245. 7.7 This is a good problem to think about the major system components or technologies that would be useful.

#246. 8.5 If you're doing `9*7` (both odd numbers), then you could do 4*7 and 5*7.

#247. 9.7 Try to reduce unnecessary database queries. If you don't need to permanently store the data in the database, you might not need it in the database at all.

#248. 5.7 Can you create a number that represents just the even bits? Then can you shift the even bits over by one?

#249. 6.10 Solution 3: If each test strip is a binary indicator, can we map , integer keys to a set of 10 binary indicators such that each key has a unique configuration (mapping)?

#250. 8.6 Think about moving the smallest disk from tower `X=0` to tower `Y=2` using tower `Z=1` as a temporary holding spot as having a solution for `f(1, X=0, Y=2, Z=1)`. Moving the smallest two disks is `f(2, X=0, Y=2, Z=1)`. Given that you have a solution for `f(1, X=0, Y=2, Z=1)` and `f(2, X=0, Y=2, Z=1)`, can you solve `f(3, X=0, Y=2, Z=1)`?

#251. 10.9 Since each column is sorted, you know that the value can't be in this column if it's smaller than the min value in this column. What else does this tell you?

#252. 6.1 What happens if you put one pill from each bottle on the scale? What if you put two pills from each bottle on the scale?

#253. 10.11 Do you necessarily need the arrays to be sorted? Can you do it with an unsorted array?

#254. 10.7 To do it with less memory, can you try multiple passes?

#255. 8.8 To get all permutations with 3 As, 2 Cs, and 1 B, you need to first pick a starting character: A, B, or C. If it's an A, then you need all permutations with 2 As, 2 Cs, and 1 B.

#256. 10.5 Try modifying binary search to handle this.

#257. 11.1 There are two mistakes in this code.

#258. 7.4 Does the parking lot have multiple levels? What "features" does it support? Is it paid? What types of vehicles?

#259. 9.5 You may need to make some assumptions (in part because you don't have an interviewer here). That's okay. Make those assumptions explicit.

#260. 8.13 Think about the first decision you have to make. The first decision is which box will be at the bottom.

#261. 5.5 If `A & B == 0`, then it means that A and B never have a 1 at the same spot. Apply this to the equation in the problem.

#262. 8.1 What is the runtime of this method? Think carefully. Can you optimize it?

#263. 10.2 Can you leverage a standard sorting algorithm?

#264. 6.9 Note: If an integer x is divisible by a, and `b = x / a`, then x is also divisible by b. Does this mean that all numbers have an even number of factors?

#265. 8.9 Adding a left or right paren at each step will eliminate duplicates. Each substring will be unique at each step. Therefore, the total string will be unique.

#266. 10.9 If the value x is smaller than the start of the column, then it also can't be in any columns to the right.

#267. 8.7 Approach 1: You can create all permutations of `abcd` by computing all permutations of `abc` and then inserting d into each possible location within those.

#268. 11.6 What are the different features and uses we would want to test?

#269. 5.2 How would you get the first digit in `.893`? If you multiplied by 10, you'd shift the values over to get `8.93`. What happens if you multiply by 2?

#270. 9.2 To find the connection between two nodes, would it be better to do a breadth-first search or depth-first search? Why?

#271. 7.7 How will you know if a user signs offline?

#272. 8.6 Observe that it doesn't really matter which tower is the source, destination, or buffer. You can do `f(3, X=0, Y=2, Z=1)` by first doing `f(2, X=0, Y=1, Z=2)` (moving two disks from tower 0 to tower 1, using tower 2 as a buffer), then moving disk 3 from tower 0 to tower 2, then doing `f(2, X=1, Y=2, Z=0)` (moving two disks from tower 1 to tower 2, using tower 0 as a buffer). How does this process repeat?

#273. 8.4 How can you build all subsets of `{a, b, c}` from the subsets of `{a, b}`?

#274. 9.5 Think about how you could design this for a single machine. Would you want a hash table? How would that work?

#275. 7.1 How, if at all, will you handle aces?

#276.	9.7	As much work as possible should be done asynchronously.
#277.	10.11	Suppose you had a sequence of three elements ({0, 1, 2}, in any order. Write out all possible sequences for those elements and how you can fix them to make 1 the peak.
#278.	8.7	Approach 2: If you had all permutations of two-character substrings, could you generate all permutations of three-character substrings?
#279.	10.9	Think about the previous hint in the context of rows.
#280.	8.5	Alternatively, if you're doing 9 * 7, you could do 4*7, double that, and then add 7.
#281.	10.7	Try using one pass to get it down to a range of values, and then a second pass to find a specific value.
#282.	6.6	Suppose there were exactly one blue-eyed person. What would that person see? When would they leave?
#283.	7.6	Which will be the easiest pieces to match first? Can you start with those? Which will be the next easiest, once you've nailed those down?
#284.	6.2	If two events are mutually exclusive (they can never occur simultaneously), you can add their probabilities together. Can you find a set of mutually exclusive events that represent making two out of three hoops?
#285.	9.2	A breadth-first search is probably better. A depth-first search can wind up going on a long path, even though the shortest path is actually very short. Is there a modification to a breadth-first search that might be even faster?
#286.	8.3	Binary search has a runtime of O(log N). Can you apply a form of binary search to the problem?
#287.	7.12	In order to handle collisions, the hash table should be an array of linked lists.
#288.	10.9	What would happen if we tried to keep track of this using an array? What are the pros and cons of this?
#289.	10.8	Can you use a bit vector?
#290.	8.4	Anything that is a subset of {a, b} is also a subset of {a, b, c}. Which sets are subsets of {a, b, c} but not {a, b}?
#291.	10.9	Can we use the previous hints to move up, down, left, and right around the rows and columns?
#292.	10.11	Revisit the set of sequences for {0, 1, 2} that you just wrote out. Imagine there are elements before the leftmost element. Are you sure that the way you swap the elements won't invalidate the previous part of the array?
#293.	9.5	Can you combine a hash table and a linked list to get the best of both worlds?
#294.	6.8	It's actually better for the first drop to be a bit lower. For example, you could drop at the 10th floor, then the 20th floor, then the 30th floor, and so on. The worst case here will be 19 drops (10, 20, ..., 100, 91, 92, ..., 99). Can you beat that? Try not randomly guessing at different solutions. Rather, think deeper. How is the worst case defined? How does the number of drops of each egg factor into that?

#295. 8.9 We can ensure that this string is valid by counting the number of left and right parens. It is always valid to add a left paren, up until the total number of pairs of parens. We can add a right paren as long as `count(left parens) <= count(right parens)`.

#296. 6.4 You can think about this either as the probability(3 ants walking clockwise) + probability(3 ants walking counter-clockwise). Or, you can think about it as: The first ant picks a direction. What's the probability of the other ants picking the same direction?

#297. 5.2 Think about what happens for values that can't be represented accurately in binary.

#298. 10.3 Can you modify binary search for this purpose?

#299. 11.1 What will happen to the `unsigned int`?

#300. 8.11 Try breaking it down into subproblems. If you were making change, what is the first choice you would make?

#301. 10.10 The problem with using an array is that it will be slow to insert a number. What other data structures could we use?

#302. 5.5 If `(n & (n-1)) == 0`, then this means that n and `n - 1` never have a 1 in the same spot. Why would that happen?

#303. 10.9 Another way to think about this is that if you drew a rectangle around a cell extending to the bottom, right coordinate of the matrix, the cell would be bigger than all the items in this square.

#304. 9.2 Is there any way to search from both the source and destination? For what reason or in what case might this be faster?

#305. 8.14 If your code looks really lengthy, with a lot of if's (for each possible operator, "target" boolean result, and left/right side), think about the relationship between the different parts. Try to simplify your code. It should not need a ton of complicated if-statements. For example, consider expressions of the form <LEFT>OR<RIGHT> versus <LEFT>AND<RIGHT>. Both may need to know the number of ways that the <LEFT> evaluates to true. See what code you can reuse.

#306. 6.9 The number 3 has an even number of factors (1 and 3). The number 12 has an even number of factors (1, 2, 3, 4, 6, 12). What numbers do not? What does this tell you about the doors?

#307. 7.12 Think carefully about what information the linked list node needs to contain.

#308. 8.12 We know that each row must have a queen. Can you try all possibilities?

#309. 8.7 Approach 2: To generate a permutation of `abcd`, you need to pick an initial character. It can be a, b, c, or d. You can then permute the remaining characters. How can you use this approach to generate all permutations of the full string?

#310. 10.3 What is the runtime of your algorithm? What will happen if the array has duplicates?

#311. 9.5 How would you scale this to a larger system?

#312. 5.4 Get Next: Can you flip a 0 to a 1 to create the next biggest number?

#313. 11.4 Think about what load testing is designed to test. What are the factors in the load of a webpage? What criteria would be used to judge if a webpage performs satisfactorily under heavy load?

#314.	5.3	Each sequence can be lengthened by merging it with an adjacent sequence (if any) or just flipping the immediate neighboring zero. You just need to find the best choice.
#315.	10.8	Consider implementing your own bit vector class. It's a good exercise and an important part of this problem.
#316.	10.11	You should be able to design an O(n) algorithm.
#317.	10.9	A cell will be larger than all the items below it and to the right. It will be smaller than all cells above it and to the left. If we wanted to eliminate the most elements first, which element should we compare the value x to?
#318.	8.6	If you're having trouble with recursion, then try trusting the recursive process more. Once you've figured out how to move the top two disks from tower 0 to tower 2, trust that you have this working. When you need to move three disks, trust that you can move two disks from one tower to another. Now, two disks have been moved. What do you do about the third?
#319.	6.1	Imagine there were just three bottles and one had heavier pills. Suppose you put different numbers of pills from each bottle on the scale (for example, bottle 1 has 5 pills, bottle 2 has 2 pills, and bottle 3 has 9 pills). What would the scale show?
#320.	10.4	Think about how binary search works. What will be the issue with just implementing binary search?
#321.	9.2	Discuss how you might implement these algorithms and this system in the real world. What sort of optimizations might you make?
#322.	8.13	Once we pick the box on the bottom, we need to pick the second box. Then the third box.
#323.	6.2	The probability of making two out of three shots is probability(make shot 1, make shot 2, miss shot 3) + probability(make shot 1, miss shot 2, make shot 3) + probability(miss shot 1, make shot 2, make shot 3) + probability(make shot 1, make shot 2, make shot 3).
#324.	8.11	If you were making change, the first choice you might make is how many quarters you need to use.
#325.	11.2	Think about issues both within the program and outside of the program (the rest of the system).
#326.	9.4	Estimate how much space is needed for this.
#327.	8.14	Look at your recursion. Do you have repeated calls anywhere? Can you memoize it?
#328.	5.7	The value 1010 in binary is 10 in decimal or 0xA in hex. What will a sequence of 101010... be in hex? That is, how do you represent an alternating sequence of 1s and 0s with 1s in the odd places? How do you do this for the reverse (1s in the even spots)?
#329.	11.3	Consider both extreme cases and more general cases.
#330.	10.9	If we compare x to the center element in the matrix, we can eliminate roughly one quarter of the elements in the matrix.
#331.	8.2	For the robot to reach the last cell, it must find a path to the second-to-last cells. For it to find a path to the second-to-last cells, it must find a path to the third-to-last cells.
#332.	10.1	Try moving from the end of the array to the beginning.

#333. 6.8 If we drop Egg 1 at fixed intervals (e.g., every 10 floors), then the worst case is the worst case for Egg 1 + the worst case for Egg 2. The problem with our earlier solutions is that as Egg 1 does more work, Egg 2 doesn't do any less work. Ideally, we'd like to balance this a bit. As Egg 1 does more work (has survived more drops), Egg 2 should have less work to do. What might this mean?

#334. 9.3 Think about how infinite loops might occur.

#335. 8.7 Approach 2: To generate all permutations of `abcd`, pick each character (a, b, c, or d) as a starting character. Permute the remaining characters and prepend the starting character. How do you permute the remaining characters? With a recursive process that follows the same logic.

#336. 5.6 How would you figure out how many bits are different between two numbers?

#337. 10.4 Binary search requires comparing an element to the midpoint. Getting the midpoint requires knowing the length. We don't know the length. Can we find it?

#338. 8.4 Subsets that contain c will be subsets `{a, b, c}` but not `{a, b}`. Can you build these subsets from the subsets of `{a, b}`?

#339. 5.4 Get Next: Flipping a 0 to a 1 will create a bigger number. The farther right the index is the smaller the bigger number is. If we have a number like 1001, we want to flip the rightmost 0 (to create 1011). But if we have a number like 1010, we should not flip the rightmost 1.

#340. 8.3 Given a specific index and value, can you identify if the magic index would be before or after it?

#341. 6.6 Now suppose there were two blue-eyed people. What would they see? What would they know? When would they leave? Remember your answer from the prior hint. Assume they know the answer to the earlier hint.

#342. 10.2 Do you even need to truly "sort"? Or is just reorganizing the list sufficient?

#343. 8.11 Once you've decided to use two quarters to make change for 98 cents, you now need to figure out how many ways to make change for 48 cents using nickels, dimes, and pennies.

#344. 7.5 Think about all the different functionality a system to read books online would have to support. You don't have to do everything, but you should think about making your assumptions explicit.

#345. 11.4 Could you build your own? What might that look like?

#346. 5.5 What is the relationship between how n looks and how `n - 1` looks? Walk through a binary subtraction.

#347. 9.4 Will you need multiple passes? Multiple machines?

#348. 10.4 We can find the length by using an exponential backoff. First check index 2, then 4, then 8, then 16, and so on. What will be the runtime of this algorithm?

#349. 11.6 What can we automate?

#350. 8.12 Each row must have a queen. Start with the last row. There are eight different columns on which you can put a queen. Can you try each of these?

#351.	7.10	Should number cells, blank cells, and bomb cells be separate classes?
#352.	5.3	Try to do it in linear time, a single pass, and `O(1)` space.
#353.	9.3	How would you detect the same page? What does this mean?
#354.	8.4	You can build the remaining subsets by adding c to all the subsets of `{a, b}`.
#355.	5.7	Try masks `0xaaaaaaaa` and `0x55555555` to select the even and odd bits. Then try shifting the even and odd bits around to create the right number.
#356.	8.7	Approach 2: You can implement this approach by having the recursive function pass back the list of the strings, and then you prepend the starting character to it. Or, you can push down a prefix to the recursive calls.
#357.	6.8	Try dropping Egg 1 at bigger intervals at the beginning and then at smaller and smaller intervals. The idea is to keep the sum of Egg 1 and Egg 2's drops as constant as possible. For each additional drop that Egg 1 takes, Egg 2 takes one fewer drop. What is the right interval?
#358.	5.4	Get Next: We should flip the rightmost non-trailing 0. The number 1010 would become 1110. Once we've done that, we need to flip a 1 to a 0 to make the number as small as possible, but bigger than the original number (1010). What do we do? How can we shrink the number?
#359.	8.1	Try memoization as a way to optimize an inefficient recursive program.
#360.	8.2	Simplify this problem a bit by first figuring out if there's a path. Then, modify your algorithm to track the path.
#361.	7.10	What is the algorithm to place the bombs around the board?
#362.	11.1	Look at the parameters for `printf`.
#363.	7.2	Before coding, make a list of the objects you need and walk through the common algorithms. Picture the code. Do you have everything you need?
#364.	8.10	Think about this as a graph.
#365.	9.3	How do you define if two pages are the same? Is it the URLs? Is it the content? Both of these can be flawed. Why?
#366.	5.8	First try the naive approach. Can you set a particular "pixel"?
#367.	6.3	Picture a domino laying down on the board. How many black squares does it cover? How many white squares?
#368.	8.13	Once you have a basic recursive algorithm implemented, think about if you can optimize it. Are there any repeated subproblems?
#369.	5.6	Think about what an XOR indicates. If you do a XOR b, where does the result have 1s? Where does it have 0s?
#370.	6.6	Build up from this. What if there were three blue-eyed people? What if there were four blue-eyed people?
#371.	8.12	Break this down into smaller subproblems. The queen at row 8 must be at column 1, 2, 3, 4, 5, 6, 7, or 8. Can you print all ways of placing eight queens where a queen is at row 8 and column 3? You then need to check all the ways of placing a queen on row 7.

#372.	5.5	When you do a binary subtraction, you flip the rightmost 0s to a 1, stopping when you get to a 1 (which is also flipped). Everything (all the 1s and 0s) on the left will stay put.
#373.	8.4	You can also do this by mapping each subset to a binary number. The `i`th bit could represent a "boolean" flag for whether an element is in the set.
#374.	6.8	Let X be the first drop of Egg 1. This means that Egg 2 would do `X - 1` drops if Egg 1 broke. We want to try to keep the sum of Egg 1 and Egg 2's drops as constant as possible. If Egg 1 breaks on the second drop, then we want Egg 2 to do `X - 2` drops. If Egg 1 breaks on the third drop, then we want Egg 2 to do `X - 3` drops. This keeps the sum of Egg 1 and Egg 2 fairly constant. What is X?
#375.	5.4	Get Next: We can shrink the number by moving all the 1s to the right of the flipped bit as far right as possible (removing a 1 in the process).
#376.	10.10	Would it work well to use a binary search tree?
#377.	7.10	To place the bombs randomly on the board: Think about the algorithm to shuffle a deck of cards. Can you apply a similar technique?
#378.	8.13	Alternatively, we can think about the repeated choices as: Does the first box go on the stack? Does the second box go on the stack? And so on.
#379.	6.5	If you fill the 5-quart jug and then use it to fill the 3-quart jug, you'll have two quarts left in the 5-quart jug. You can either keep those two quarts where they are, or you can dump the contents of the smaller jug and pour the two quarts in there.
#380.	8.11	Analyze your algorithm. Is there any repeated work? Can you optimize this?
#381.	5.8	When you're drawing a long line, you'll have entire bytes that will become a sequence of 1s. Can you set this all at once?
#382.	8.10	You can implement this using depth-first search (or breadth-first search). Each adjacent pixel of the "right" color is a connected edge.
#383.	5.5	Picture `n` and `n-1`. To subtract 1 from `n`, you flipped the rightmost 1 to a 0 and all the 0s on its right to 1s. If `n & n-1 == 0`, then there are no 1s to the left of the first 1. What does that mean about `n`?
#384.	5.8	What about the start and end of the line? Do you need to set those pixels individually, or can you set them all at once?
#385.	9.1	Think about this as a real-world application. What are the different factors you would need to consider?
#386.	7.10	How do you count the number of bombs neighboring a cell? Will you iterate through all cells?
#387.	6.1	You should be able to have an equation that tells you the heavy bottle based on the weight.
#388.	8.2	Think again about the efficiency of your algorithm. Can you optimize it?
#389.	7.9	The `rotate()` method should be able to run in `O(1)` time.
#390.	5.4	Get Previous: Once you've solved Get Next, try to invert the logic for Get Previous.
#391.	5.8	Does your code handle the case when `x1` and `x2` are in the same byte?
#392.	10.10	Consider a binary search tree where each node stores some additional data.

#393.	11.6	Have you thought about security and reliability?
#394.	8.11	Try using memoization.
#395.	6.8	I got 14 drops in the worst case. What did you get?
#396.	9.1	There's no one right answer here. Discuss several different technical implementations.
#397.	6.3	How many black squares are there on the board? How many white squares?
#398.	5.5	We know that n must have only one 1 if `n & (n-1) == 0`. What sorts of numbers have only one 1?
#399.	7.10	When you click on a blank cell, what is the algorithm to expand the neighboring cells?
#400.	6.5	Once you've developed a way to solve this problem, think about it more broadly. If you are given a jug of size X and another jug of size Y, can you always use it to measure Z?
#401.	11.3	Is it possible to test everything? How will you prioritize testing?

III

Hints for Knowledge-Based Questions

#402.	12.9	Focus on the concept firsts, then worry about the exact implementation. How should `SmartPointer` look?
#403.	15.2	A context switch is the time spent switching between two processes. This happens when you bring one process into execution and swap out the existing process.
#404.	13.1	Think about who can access private methods.
#405.	15.1	How do these differ in terms of memory?
#406.	12.11	Recall that a two dimensional array is essentially an array of arrays.
#407.	15.2	Ideally, we would like to record the timestamp when one process "stops" and the timestamp when another process "starts." But how do we know when this swapping will occur?
#408.	14.1	A GROUP BY clause might be useful.
#409.	13.2	When does a finally block get executed? Are there any cases where it won't get executed?
#410.	12.2	Can we do this in place?
#411.	14.2	It might be helpful to break the approach into two pieces. The first piece is to get each building ID and the number of open requests. Then, we can get the building names.
#412.	13.3	Consider that some of these might have different meanings depending on where they are applied.
#413.	12.10	Typically, `malloc` will just give us an arbitrary block of memory. If we can't override this behavior, can we work with it to do what we need?
#414.	15.7	First implement the single-threaded FizzBuzz problem.
#415.	15.2	Try setting up two processes and have them pass a small amount of data back and forth. This will encourage the system to stop one process and bring the other one in.
#416.	13.4	The purpose of these might be somewhat similar, but how does the implementation differ?
#417.	15.5	How can we ensure that `first()` has terminated before calling `second()`?
#418.	12.11	One approach is to call `malloc` for each array. How would we free the memory here?
#419.	15.3	A deadlock can happen when there's a "cycle" in the order of who is waiting for whom. How can we break or prevent this cycle?

#420.	13.5	Think about the underlying data structure.
#421.	12.7	Think about why we use virtual methods.
#422.	15.4	If every thread had to declare upfront what processes it might need, could we detect possible deadlocks in advance?
#423.	12.3	What is the underlying data structure behind each? What are the implications of this?
#424.	13.5	`HashMap` uses an array of linked lists. `TreeMap` uses a red-black tree. `LinkedHashMap` uses doubly-linked buckets. What is the implication of this?
#425.	13.4	Consider the usage of primitive types. How else might they differ in terms of how you can use the types?
#426.	12.11	Can we allocate this instead as a contiguous block of memory?
#427.	12.8	This data structure can be pictured as a binary tree, but it's not necessarily. What if there's a loop in the structure?
#428.	14.7	You probably need a list of students, their courses, and another table building a relationship between students and courses. Note that this is a many-to-many relationship.
#429.	15.6	The keyword `synchronized` ensures that two threads cannot execute synchronized methods on the same instance at the same time.
#430.	13.5	Consider how they might differ in terms of the order of iteration through the keys. Why might you want one option instead of the others?
#431.	14.3	First try to get a list of the IDs (just the IDs) of all the relevant apartments.
#432.	12.10	Imagine we have a sequential set of integers (3, 4, 5, ...). How big does this set need to be to ensure that one of the numbers is divisible by 16?
#433.	15.5	Why would using `boolean` flags to do this be a bad idea?
#434.	15.4	Think about the order of requests as a graph. What does a deadlock look like within this graph?
#435.	13.6	Object reflection allows you to get information about methods and fields in an object. Why might this be useful?
#436.	14.6	Be particularly careful about which relationships are one-to-one vs. one-to-many vs. many-to-many.
#437.	15.3	One idea is to just not let a philosopher hold onto a chopstick if he can't get the other one.
#438.	12.9	Think about tracking the number of references. What will this tell us?
#439.	15.7	Don't try to do anything fancy on the single-threaded problem. Just get something that is simple and easily readable.
#440.	12.10	How will we free the memory?
#441.	15.2	It's okay if your solution isn't totally perfect. That might not be possible. Discuss the tradeoffs of your approach.
#442.	14.7	Think carefully about how you handle ties when selecting the top 10%.

#443.	13.8	A naive approach is to pick a random subset size z and then iterate through the elements, putting it in the set with probability `z/list_size`. Why would this not work?
#444.	14.5	Denormalization means adding redundant data to a table. It's typically used in very large systems. Why might this be useful?
#445.	12.5	A shallow copy copies just the initial data structure. A deep copy does this, and also copies any underlying data. Given this, why might you use one versus the other?
#446.	15.5	Would semaphores be useful here?
#447.	15.7	Outline the structure for the threads without worrying about synchronizing anything.
#448.	13.7	Consider how you'd implement this first without lambda expressions.
#449.	12.1	If we already had the number of lines in the file, how would we do this?
#450.	13.8	Pick the list of all the subsets of an n-element set. For any given item `x`, half of the subsets contain x and half do not.
#451.	14.4	Describe INNER JOINs and OUTER JOINs. OUTER JOINs can have multiple types: left, right, and full.
#452.	12.2	Be careful about the null character.
#453.	12.9	What are all the different methods/operators we might want to override?
#454.	13.5	What would the runtime of the common operations be?
#455.	14.5	Think about the cost of joins on a large system.
#456.	12.6	The keyword `volatile` signals that a variable might be changed from outside of the program, such as by another process. Why might this be necessary?
#457.	13.8	Do not pick the length of the subset in advance. You don't need to. Instead, think about this as picking whether each element will be put into the set.
#458.	15.7	Once you get the structure of each thread done, think about what you need to synchronize.
#459.	12.1	Suppose we didn't have the number of lines in the file. Is there a way we could do this without first counting the number of lines?
#460.	12.7	What would happen if the destructor were not virtual?
#461.	13.7	Break this up into two parts: filtering the countries and then getting a sum.
#462.	12.8	Consider using a hash table.
#463.	12.4	You should discuss vtables here.
#464.	13.7	Can you do this without a `filter` operation?

IV

Hints for Additional Review Problems

#465. 16.3 Think about what you're going to design for.

#466. 16.12 Consider a recursive or tree-like approach.

#467. 17.1 Walk through binary addition by hand (slowly!) and try to really understand what is happening.

#468. 16.13 Draw a square and a bunch of lines that cut it in half. Where are those lines located?

#469. 17.24 Start with a brute force solution.

#470. 17.14 There are actually several approaches. Brainstorm these. It's okay to start off with a naive approach.

#471. 16.20 Consider recursion.

#472. 16.3 Will all lines intercept? What determines if two lines intercept?

#473. 16.7 Let k be 1 if `a > b` and 0 otherwise. If you were given k, could you return the max (without a comparison or if-else logic)?

#474. 16.22 The tricky bit is handling an infinite grid. What are your options?

#475. 17.15 Try simplifying this problem: What if you just needed to know the longest word made up of two other words in the list?

#476. 16.10 Solution 1: Can you count the number of people alive in each year?

#477. 17.25 Start by grouping the dictionary by the word lengths, since you know each column has to be the same length and each row has to be the same length.

#478. 17.7 Discuss the naive approach: merging names together when they are synonyms. How would you identify transitive relationships? `A == B`, `A == C`, and `C == D` implies `A == D == B == C`.

#479. 16.13 Any straight line that cuts a square in half goes through the center of the square. How then can you find a line that cuts two squares in half?

#480. 17.17 Start with a brute force solution. What is the runtime?

#481. 16.22 Option #1: Do you actually need an infinite grid? Read the problem again. Do you know the max size of the grid?

#482. 16.16 Would it help to know the longest sorted sequences at the beginning and end?

#483. 17.2 Try approaching this problem recursively.

#484.	17.26	Solution 1: Start with just a simple algorithm comparing all documents to all other documents. How would you compute the similarity of two documents as fast as possible?
#485.	17.5	It doesn't really matter which letter or number it is. You can simplify this problem to just having an array of As and Bs. You would then be looking for the longest subarray with an equal number of As and Bs.
#486.	17.11	Consider first the algorithm for finding the closest distance if you will run the algorithm only once. You should be able to do this in O(N) time, where N is the number of words in the document.
#487.	16.20	Can you recursively try all possibilities?
#488.	17.9	Be clear about what this problem is asking for. It's asking for the kth smallest number in the form $3^a * 5^b * 7^c$.
#489.	16.2	Think about what the best conceivable runtime is for this problem. If your solution matches the best conceivable runtime, then you probably can't do any better.
#490.	16.10	Solution 1: Try using a hash table, or an array that maps from a birth year to how many people are alive in that year.
#491.	16.14	Sometimes, a brute force is a pretty good solution. Can you try all possible lines?
#492.	16.1	Try picturing the two numbers, a and b, on a number line.
#493.	17.7	The core part of the problem is to group names into the various spellings. From there, figuring out the frequencies is relatively easy.
#494.	17.3	If you haven't already, solve 17.2 on page 186.
#495.	17.16	There are recursive and iterative solutions to this problem, but it's probably easier to start with the recursive solution.
#496.	17.13	Try a recursive approach.
#497.	16.3	Infinite lines will almost always intersect—unless they're parallel. Parallel lines might still "intercept"—if they're the same lines. What does this mean for line segments?
#498.	17.26	Solution 1: To compute the similarity of two documents, try reorganizing the data in some way. Sorting? Using another data structure?
#499.	17.15	If we wanted to know just the longest word made up of other words in the list, then we could iterate over all words, from longest to shortest, checking if each could be made up of other words. To check this, we split the string in all possible locations.
#500.	17.25	Can you find a word rectangle of a specific length and width? What if you just tried all options?
#501.	17.11	Adapt your algorithm for one execution of the algorithm for repeated executions. What is the slow part? Can you optimize it?
#502.	16.8	Try thinking about the number in terms of chunks of three digits.
#503.	17.19	Start with the first part: Finding the missing number if only one number is missing.

#504. 17.16 Recursive solution: You have two choices at each appointment (take the appointment or reject the appointment). As a brute force approach, you can recurse through all possibilities. Note, though, that if you take request `i`, your recursive algorithm should skip request `i + 1`.

#505. 16.23 Be very careful that your solution actually returns each value from 0 through 6 with equal probability.

#506. 17.22 Start with a brute force, recursive solution. Just create all words that are one edit away, check if they are in the dictionary, and then attempt that path.

#507. 16.10 Solution 2: What if you sorted the years? What would you sort by?

#508. 17.9 What does a brute force solution to get the kth smallest value for $3^a * 5^b * 7^c$ look like?

#509. 17.12 Try a recursive approach.

#510. 17.26 Solution 1: You should be able to get an `O(A+B)` algorithm to compute the similarity of two documents.

#511. 17.24 The brute force solution requires us to continuously compute the sums of each matrix. Can we optimize this?

#512. 17.7 One thing to try is maintaining a mapping of each name to its "true" spelling. You would also need to map from a true spelling to all the synonyms. Sometimes, you might need to merge two different groups of names. Play around with this algorithm to see if you can get it to work. Then see if you can simplify/optimize it.

#513. 16.7 If k were 1 when `a > b` and 0 otherwise, then you could return `a*k + b*(not k)`. But how do you create k?

#514. 16.10 Solution 2: Do you actually need to match the birth years and death years? Does it matter when a specific person died, or do you just need a list of the years of deaths?

#515. 17.5 Start with a brute force solution.

#516. 17.16 Recursive solution: You can optimize this approach through memoization. What is the runtime of this approach?

#517. 16.3 How can we find the intersection between two lines? If two line segments intercept, then this must be at the same point as their "infinite" extensions. Is this intersection point within both lines?

#518. 17.26 Solution 1: What is the relationship between the intersection and the union? Can you compute one from the other?

#519. 17.20 Recall that the median means the number for which half the numbers are larger and half the numbers are smaller.

#520. 16.14 You can't truly try all possible lines in the world—that's infinite. But you know that a "best" line must intersect at least two points. Can you connect each pair of points? Can you check if each line is indeed the best line?

#521. 16.26 Can we just process the expression from left to right? Why might this fail?

#522. 17.10 Start with a brute force solution. Can you just check each value to see if it's the majority element?

#523. 16.10 Solution 2: Observe that people are "fungible." It doesn't matter who was born and when they died. All you need is a list of birth years and death years. This might make the question of how you sort the list of people easier.

#524. 16.25 First scope the problem. What are the features you would want?

#525. 17.24 Can you do any sort of precomputation to make computing the sum of a submatrix `O(1)`?

#526. 17.16 Recursive solution: The runtime of your memoization approach should be `O(N)`, with `O(N)` space.

#527. 16.3 Think carefully about how to handle the case of line segments that have the same slope and y-intercept.

#528. 16.13 To cut two squares in half, a line must go through the middle of both squares.

#529. 16.14 You should be able to get to an $O(N^2)$ solution.

#530. 17.14 Consider thinking about reorganizing the data in some way or using additional data structures.

#531. 16.17 Picture the array as alternating sequences of positive and negative numbers. Observe that we would never include just part of a positive sequence or part of a negative sequence.

#532. 16.10 Solution 2: Try creating a sorted list of births and a sorted list of deaths. Can you iterate through both, tracking the number of people alive at any one time?

#533. 16.22 Option #2: Think about how an `ArrayList` works. Can you use an `ArrayList` for this?

#534. 17.26 Solution 1: To understand the relationship between the union and the intersection of two sets, consider a Venn diagram (a diagram where one circle overlaps another circle).

#535. 17.22 Once you have a brute force solution, try to find a faster way of getting all valid words that are one edit away. You don't want to create all strings that are one edit away when the vast majority of them are not valid dictionary words.

#536. 16.2 Can you use a hash table to optimize the repeated case?

#537. 17.7 An easier way of taking the above approach is to have each name map to a list of alternate spellings. What should happen when a name in one group is set equal to a name in another group?

#538. 17.11 You could build a lookup table that maps from a word to a list of the locations where each word appears. How then could you find the closest two locations?

#539. 17.24 What if you precomputed the sum of the submatrix starting at the top left corner and continuing to each cell? How long would it take you to compute this? If you did this, could you then get the sum of an arbitrary submatrix in `O(1)` time?

#540. 16.22 Option #2: It's not impossible to use an `ArrayList`, but it would be tedious. Perhaps it would be easier to build your own, but specialized for matrices.

#541. 16.10 Solution 3: Each birth adds one person and each death removes a person. Try writing an example of a list of people (with birth and death years) and then re-formatting this into a list of each year and a +1 for a birth and a -1 for a death.

#542. 17.16 Iterative solution: Take the recursive solution and investigate it more. Can you implement a similar strategy iteratively?

#543. 17.15 Extend the earlier idea to multiple words. Can we just break each word up in all possible ways?

#544. 17.1 You can think about binary addition as iterating through the number, bit by bit, adding two bits, and then carrying over the one if necessary. You could also think about it as grouping the operations. What if you first added each of the bits (without carrying any overflow)? After that, you can handle the overflow.

#545. 16.21 Do some math here or play around with some examples. What does this pair need to look like? What can you say about their values?

#546. 17.20 Note that you have to store all the elements you've seen. Even the smallest of the first 100 elements could become the median. You can't just toss very low or very high elements.

#547. 17.26 Solution 2: It's tempting to try to think of minor optimizations—for example, keeping track of the min and max elements in each array. You could then figure out quickly, in specific cases, if two arrays don't overlap. The problem with that (and other optimizations along these lines) is that you still need to compare all documents to all other documents. It doesn't leverage the fact that the similarity is sparse. Given that we have a lot of documents, we really need to not compare all documents to all other documents (even if that comparison is very fast). All such solutions will be $O(D^2)$, where D is the number of documents. We shouldn't compare all documents to all other documents.

#548. 16.24 Start with a brute force solution. What is the runtime? What is the best conceivable runtime for this problem?

#549. 16.10 Solution 3: What if you created an array of years and how the population changed in each year? Could you then find the year with the highest population?

#550. 17.9 In looking for the kth smallest value of $3^a * 5^b * 7^c$, we know that a, b, and c will be less than or equal to k. Can you generate all such numbers?

#551. 16.17 Observe that if you have a sequence of values which have a negative sum, those will never start or end a sequence. (They could be present in a sequence if they connected two other sequences.)

#552. 17.14 Can you sort the numbers?

#553. 16.16 We can think about the array as divided into three subarrays: `LEFT`, `MIDDLE`, `RIGHT`. `LEFT` and `RIGHT` are both sorted. The `MIDDLE` elements are in an arbitrary order. We need to expand `MIDDLE` until we could sort those elements and then have the entire array sorted.

#554. 17.16 Iterative solution: It's probably easiest to start with the end of the array and work backwards.

#555. 17.26 Solution 2: If we can't compare all documents to all other documents, then we need to dive down and start looking at things at the element level. Consider a naive solution and see if you can extend that to multiple documents.

#556. 17.22 To quickly get the valid words that are one edit away, try to group the words in the dictionary in a useful way. Observe that all words in the form b_ll (such as `bill`, `ball`, `bell`, and `bull`) will be one edit away. However, those aren't the only words that are one edit away from `bill`.

#557. 16.21 When you move a value a from array A to array B, then A's sum decreases by a and B's sum increases by a. What happens when you swap two values? What would be needed to swap two values and get the same sum?

#558. 17.11 If you had a list of the occurrences of each word, then you are really looking for a pair of values within two arrays (one value for each array) with the smallest difference. This could be a fairly similar algorithm to your initial algorithm.

#559. 16.22 Option #2: One approach is to just double the size of the array when the ant wanders to an edge. How will you handle the ant wandering into negative coordinates, though? Arrays can't have negative indices.

#560. 16.13 Given a line (slope and y-intercept), can you find where it intersects another line?

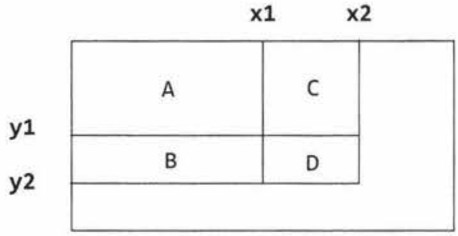
#561. 17.26 Solution 2: One way to think about this is that we need to be able to very quickly pull a list of all documents with some similarity to a specific document. (Again, we should not do this by saying "look at all documents and quickly eliminate the dissimilar documents." That will be at least $O(D^2)$.)

#562. 17.16 Iterative solution: Observe that you would never skip three appointments in a row. Why would you? You would always be able to take the middle booking.

#563. 16.14 Have you tried using a hash table?

#564. 16.21 If you swap two values, a and b, then the sum of A becomes `sumA - a + b` and the sum of B becomes `sumB - b + a`. These sums need to be equal.

#565. 17.24 If you can precompute the sum from the top left corner to each cell, you can use this to compute the sum of an arbitrary submatrix in $O(1)$ time. Picture a particular submatrix. The full, precomputed sum will include this submatrix, an array immediately above it (C), and array to the left (B), and an area to the top and left (A). How can you compute the sum of just D?



#566. 17.10 Consider the brute force solution. We pick an element and then validate if it's the majority element by counting the number of matching and non-matching elements. Suppose, for the first element, the first few checks reveal seven non-matching elements and three matching elements. Is it necessary to keep checking this element?

#567. 16.17 Start from the beginning of the array. As that subsequence gets larger, it stays as the best subsequence. Once it becomes negative, though, it's useless.

#568. 17.16 Iterative solution: If you take appointment `i`, you will never take appointment `i + 1`, but you will always take appointment `i + 2` or `i + 3`.

#569. 17.26 Solution 2: Building off the earlier hint, we can ask what defines the list of documents with some similarity to a document like {13, 16, 21, 3}. What attributes does that list have? How would we gather all documents like that?

#570. 16.22 Option #2: Observe that nothing in the problem stipulates that the label for the coordinates must remain the same. Can you move the ant and all cells into positive coordinates? In other words, what would happen if, whenever you needed to grow the array in a negative direction, you relabeled all the indices such that they were still positive?

#571. 16.21 You are looking for values a and b where `sumA - a + b = sumB - b + a`. Do the math to work out what this means for a and b's values.

#572. 16.9 Approach these one by one, starting with subtraction. Once you've completed one function, you can use it to implement the others.

#573. 17.6 Start with a brute force solution.

#574. 16.23 Start with a brute force solution. How many times does it call `rand5()` in the worst case?

#575. 17.20 Another way to think about this is: Can you maintain the bottom half of elements and the top half of elements?

#576. 16.10 Solution 3: Be careful with the little details in this problem. Does your algorithm/code handle a person who dies in the same year that they are born? This person should be counted as one person in the population count.

#577. 17.26 Solution 2: The list of documents similar to {13, 16, 21, 3} includes all documents with a 13, 16, 21, and 3. How can we efficiently find this list? Remember that we'll be doing this for many documents, so some precomputing can make sense.

#578. 17.16 Iterative solution: Use an example and work backwards. You can easily find the optimal solution for the subarrays $\{r_n\}$, $\{r_{n-1}, r_n\}$, $\{r_{n-2}, ..., r_n\}$. How would you use those to quickly find the optimal solution for $\{r_{n-3}, ..., r_n\}$?

#579. 17.2 Suppose you had a method `shuffle` that worked on decks up to `n - 1` elements. Could you use this method to implement a new `shuffle` method that works on decks up to n elements?

#580. 17.22 Create a mapping from a wildcard form (like `b_ll`) to all words in that form. Then, when you want to find all words that are one edit away from `bill`, you can look up `_ill`, `b_ll`, `bi_l`, and `bil_` in the mapping.

#581. 17.24 The sum of just D will be `sum(A&B&C&D) - sum(A&B) - sum(A&C) + sum(A)`.

#582. 17.17 Can you use a trie?

#583. 16.21 If we do the math, we are looking for a pair of values such that `a - b = (sumA - sumB) / 2`. The problem then reduces to looking for a pair of values with a particular difference.

#584. 17.26 Solution 2: Try building a hash table from each word to the documents that contain this word. This will allow us to easily find all documents with some similarity to {13, 16, 21, 3}.

#585. 16.5 How does a zero get into the result of `n!`? What does it mean?

#586. 17.7 If each name maps to a list of its alternate spellings, you might have to update a lot of lists when you set X and Y as synonyms. If X is a synonym of `{A, B, C}`, and Y is a synonym of `{D, E, F}` then you would need to add `{Y, D, E, F}` to A's synonym list, B's synonym list, C's synonym list, and X's synonym list. Ditto for `{Y, D, E, F}`. Can we make this faster?

#587. 17.16 Iterative solution: If you take an appointment, you can't take the next appointment, but you can take anything after that. Therefore, $\texttt{optimal}(r_i, \ldots, r_n) = \max(r_i + \texttt{optimal}(r_{i+2}, \ldots, r_n), \texttt{optimal}(r_{i+1}, \ldots, r_n))$. You can solve this iteratively by working backwards.

#588. 16.8 Have you considered negative numbers? Does your solution work for values like 100,030,000?

#589. 17.15 When you get recursive algorithms that are very inefficient, try looking for repeated subproblems.

#590. 17.19 Part 1: If you have to find the missing number in `O(1)` space and `O(N)` time, then you can do a only constant number of passes through the array and can store only a few variables.

#591. 17.9 Look at the list of all values for $3^a * 5^b * 7^c$. Observe that each value in the list will be 3*(some previous value), 5*(some previous value), or 7*(some previous value).

#592. 16.21 A brute force solution is to just look through all pairs of values to find one with the right difference. This will probably look like an outer loop through A with an inner loop through B. For each value, compute the difference and compare it to what we're looking for. Can we be more specific here, though? Given a value in A and a target difference, do we know the exact value of the element within B we're looking for?

#593. 17.14 What about using a heap or tree of some sort?

#594. 16.17 If we tracked the running sum, we should reset it as soon as the subsequence becomes negative. We would never add a negative sequence to the beginning or end of another subsequence.

#595. 17.24 With precomputation, you should be able to get a runtime of $O(N^4)$. Can you make this even faster?

#596. 17.3 Try this recursively. Suppose you had an algorithm to get a subset of size `m` from n - 1 elements. Could you develop an algorithm to get a subset of size `m` from n elements?

#597. 16.24 Can we make this faster with a hash table?

#598. 17.22 Your previous algorithm probably resembles a depth-first search. Can you make this faster?

#599. 16.22 Option #3: Another thing to think about is whether you even need a grid to implement this. What information do you actually need in the problem?

#600. 16.9 Subtraction: Would a negate function (which converts a positive integer to negative) help? Can you implement this using the add operator?

#601. 17.1 Focus on just one of the steps above. If you "forgot" to carry the ones, what would the add operation look like?

#602. 16.21 What the brute force really does is look for a value within B which equals a `- target`. How can you more quickly find this element? What approaches help us quickly find out if an element exists within an array?

#603. 17.26 Solution 2: Once you have a way of easily finding the documents similar to a particular document, you can go through and just compute the similarity to those documents using a simple algorithm. Can you make this faster? Specifically, can you compute the similarity directly from the hash table?

#604. 17.10 The majority element will not necessarily look like the majority element at first. It is possible, for example, to have the majority element appear in the first element of the array and then not appear again for the next eight elements. However, in those cases, the majority element will appear later in the array (in fact, many times later on in the array). It's not necessarily critical to continue checking a specific instance of an element for majority status once it's already looking "unlikely."

#605. 17.7 Instead, X, A, B, and C should map to the same instance of the set `{X, A, B, C}`. Y, D, E, and F should map to the same instance of `{Y, D, E, F}`. When we set X and Y as synonyms, we can then just copy one of the sets into the other (e.g., add `{Y, D, E, F}` to `{X, A, B, C}`). How else do we change the hash table?

#606. 16.21 We can use a hash table here. We can also try sorting. Both help us locate elements more quickly.

#607. 17.16 Iterative solution: If you're careful about what data you really need, you should be able to solve this in `O(n)` time and `O(1)` additional space.

#608. 17.12 Think about it this way: If you had methods called `convertLeft` and `convertRight` (which would convert left and right subtrees to doubly linked lists), could you put those together to convert the whole tree to a doubly linked list?

#609. 17.19 Part 1: What if you added up all the values in the array? Could you then figure out the missing number?

#610. 17.4 How long would it take you to figure out the least significant bit of the missing number?

#611. 17.26 Solution 2: Imagine you are looking up the documents similar to {1, 4, 6} by using a hash table that maps from a word to documents. The same document ID appears multiple times when doing this lookup. What does that indicate?

#612. 17.6 Rather than counting the number of twos in each number, think about digit by digit. That is, count the number of twos in the first digit (for each number), then the number of twos in the second digit (for each number), then the number of twos in the third digit (for each number), and so on.

#613. 16.9 Multiply: it's easy enough to implement `multiply` using `add`. But how do you handle negative numbers?

#614. 16.17 You can solve this in `O(N)` time and `O(1)` space.

#615. 17.24 Suppose this was just a single array. How could we compute the subarray with the largest sum? See 16.17 for a solution to this.

#616. 16.22 Option #3: All you actually need is some way of looking up if a cell is white or black (and of course the position of the ant). Can you just keep a list of all the white cells?

#617. 17.17 One solution is to insert every suffix of the larger string into the trie. For example, if the word is `dogs`, the suffixes would be `dogs`, `ogs`, `gs`, and `s`. How would this help you solve the problem? What is the runtime here?

#618. 17.22 A breadth-first search will often be faster than a depth-first search—not necessarily in the worst case, but in many cases. Why? Can you do something even faster than this?

#619. 17.5 What if you just started from the beginning, counting the number of As and the number of Bs you've seen so far? (Try making a table of the array and the number of As and Bs thus far.)

#620. 17.10 Note also that the majority element must be the majority element for some subarray and that no subarray can have multiple majority elements.

#621. 17.24 Suppose I just wanted you to find the maximum submatrix starting at row `r1` and ending at row `r2`, how could you most efficiently do this? (See the prior hint.) If I now wanted you find the maximum subarray from `r1` to `(r2+2)`, could you do this efficiently?

#622. 17.9 Since each number is 3, 5, or 7 times a previous value in the list, we could just check all possible values and pick the next one that hasn't been seen yet. This will result in a lot of duplicated work. How can we avoid this?

#623. 17.13 Can you just try all possibilities? What might that look like?

#624. 16.26 Multiplication and division are higher priority operations. In an expression like `3*4 + 5*9/2 + 3`, the multiplication and division parts need to be grouped together.

#625. 17.14 If you picked an arbitrary element, how long would it take you to figure out the rank of this element (the number of elements bigger or smaller than it)?

#626. 17.19 Part 2: We're now looking for two missing numbers, which we will call a and b. The approach from part 1 will tell us the sum of a and b, but it won't actually tell us a and b. What other calculations could we do?

#627. 16.22 Option #3: You could consider keeping a hash set of all the white cells. How will you be able to print the whole grid, though?

#628. 17.1 The adding step alone would convert 1 + 1 -> 0, 1 + 0 -> 1, 0 + 1 -> 1, 0 + 0 -> 0. How do you do this without the + sign?

#629. 17.21 What role does the tallest bar in the histogram play?

#630. 16.25 What data structure would be most useful for the lookups? What data structure would be most useful to know and maintain the order of items?

#631. 16.18 Start with a brute force approach. Can you try all possibilities for a and b?

#632. 16.6 What if you sorted the arrays?

#633. 17.11 Can you just iterate through both arrays with two pointers? You should be able to do it in `O(A+B)` time, where A and B are the sizes of the two arrays.

#634. 17.2 You could build this algorithm recursively by swapping the nth element for any of the elements before it. What would this look like iteratively?

#635. 16.21 What if the sum of A is 11 and the sum of B is 8? Can there be a pair with the right difference? Check that your solution handles this situation appropriately.

#636. 17.26 Solution 3: There's an alternative solution. Consider taking all of the words from all of the documents, throwing them into one giant list, and sorting this list. Assume you could still know which document each word came from. How could you track the similar pairs?

#637. 16.23 Make a table indicating how each possible sequence of calls to `rand5()` would map to the result of `rand7()`. For example, if you were implementing `rand3()` with `(rand2() + rand2()) % 3`, then the table would look like the below. Analyze this table. What can it tell you?

```
1st    2nd    Result
0      0      0
0      1      1
1      0      1
1      1      2
```

#638. 17.8 This problem asks us to find the longest sequence of pairs you can build such that both sides of the pair are constantly increasing. What if you needed only one side of the pair to increase?

#639. 16.15 Try first creating an array with the frequency that each item occurs.

#640. 17.21 Picture the tallest bar, and then the next tallest bar on the left and the next tallest bar on the right. The water will fill the area between those. Can you calculate that area? What do you do about the rest?

#641. 17.6 Is there a faster way of calculating how many twos are in a particular digit across a range of numbers? Observe that roughly $\frac{1}{10}$ th of any digit should be a 2—but only roughly. How do you make that more exact?

#642. 17.1 You can do the add step with an XOR.

#643. 16.18 Observe that one of the substrings, either a or b, must start at the beginning of the string. That cuts down the number of possibilities.

#644. 16.24 What if the array were sorted?

#645. 17.18 Start with a brute force solution.

#646. 17.12 Once you have a basic idea for a recursive algorithm, you might get stuck on this: sometimes your recursive algorithm needs to return the start of the linked list, and sometimes it needs to return the end. There are multiple ways of solving this issue. Brainstorm some of them.

#647. 17.14 If you picked an arbitrary element, you would, on average, wind up with an element around the 50th percentile mark (half the elements above it and half the elements below). What if you did this repeatedly?

#648. 16.9 Divide: If you're trying to compute, where $x = \frac{a}{b}$, remember that a = bx. Can you find the closest value for x? Remember that this is integer division and x should be an integer.

#649. 17.19 Part 2: There are a lot of different calculations we could try. For example, we could multiply all the numbers, but that will only lead us to the product of a and b.

#650. 17.10 Try this: Given an element, start checking if this is the start of a subarray for which it's the majority element. Once it's become "unlikely" (appears less than half the time), start checking at the next element (the element after the subarray).

#651. 17.21 You can calculate the area between the tallest bar overall and the tallest bar on the left by just iterating through the histogram and subtracting out any bars in between. You can do the same thing with the right side. How do you handle the remainder of the graph?

#652. 17.18 One brute force solution is to take each starting position and move forward until you've found a subsequence which contains all the target characters.

#653. 16.18 Don't forget to handle the possibility that the first character in the `pattern` is b.

#654. 16.20 In the real world, we should know that some prefixes/substrings won't work. For example, consider the number `33835676368`. Although `3383` does correspond to `fftf`, there are no words that start with `fftf`. Is there a way we can short-circuit in cases like this?

#655. 17.7 An alternative approach is to think of this as a graph. How would this work?

#656. 17.13 You can think about the choices the recursive algorithm makes in one of two ways: (1) At each character, should I put a space here? (2) Where should I put the next space? You can solve both of these recursively.

#657. 17.8 If you needed only one side of the pair to increase, then you would just sort all the values on that side. Your longest sequence would in fact be all of the pairs (other than any duplicates, since the longest sequence needs to strictly increase). What does this tell you about the original problem?

#658. 17.21 You can handle the remainder of the graph by just repeating this process: find the tallest bar and the second tallest bar, and subtract out the bars in between.

#659. 17.4 To find the least significant bit of the missing number, note that you know how many `0`s and `1`s to expect. For example, if you see three `0`s and three `1`s in the least significant bit, then the missing number's least significant bit must be a `1`. Think about it: in any sequence of `0`s and `1`s, you'd get a `0`, then a `1`, then a `0`, then a `1`, and so on.

#660. 17.9 Rather than checking all values in the list for the next value (by multiplying each by 3, 5, and 7), think about it this way: when you insert a value `x` into the list, you can "create" the values `3x`, `5x`, and `7x` to be used later.

#661. 17.14 Think about the previous hint some more, particularly in the context of quicksort.

#662. 17.21 How can you make the process of finding the next tallest bar on each side faster?

#663. 16.18 Be careful with how you analyze the runtime. If you iterate through $O(n^2)$ substrings and each one does an $O(n)$ string comparison, then the total runtime is $O(n^3)$.

#664. 17.1 Now focus on the carrying. In what cases will values carry? How do you apply the carry to the number?

#665. 16.26 Consider thinking about it as, when you get to a multiplication or division sign, jumping to a separate "process" to compute the result of this chunk.

#666. 17.8 If you sort the values based on height, then this will tell you the ordering of the final pairs. The longest sequence must be in this relative order (but not necessarily containing all of the pairs). You now just need to find the longest increasing subsequence on weight while keeping the items in the same relative order. This is essentially the same problem as having an array of integers and trying to find the longest sequence you can build (without reordering those items).

#667. 16.16 Consider the three subarrays: `LEFT`, `MIDDLE`, `RIGHT`. Focus on just this question: Can you sort middle such that the entire array becomes sorted? How would you check this?

#668. 16.23 Looking at this table again, note that the number of rows will be 5^k, where k is the max number of calls to `rand5()`. In order to make each value between 0 and 6 have equal probability, $\frac{1}{7}$th of the rows must map to 0, $\frac{1}{7}$th to 1, and so on. Is this possible?

#669. 17.18 Another way of thinking about the brute force is that we take each starting index and find the next instance of each element in the target string. The maximum of all these next instances marks the end of a subsequence which contains all the target characters. What is the runtime of this? How can we make it faster?

#670. 16.6 Think about how you would merge two sorted arrays.

#671. 17.5 When the above tables have equal values for the number of As and Bs, the entire subarray (starting from index 0) has an equal number of As and Bs. How could you use this table to find qualifying subarrays that don't start at index 0?

#672. 17.19 Part 2: Adding the numbers together will tell us the result of a + b. Multiplying the numbers together will tell us the result of a * b. How can we get the exact values for a and b?

#673. 16.24 If we sorted the array, we could do repeated binary searches for the complement of a number. What if, instead, the array is given to us sorted? Could we then solve the problem in `O(N)` time and `O(1)` space?

#674. 16.19 If you were given the row and column of a water cell, how can you find all connected spaces?

#675. 17.7 We can treat adding X, Y as synonyms as adding an edge between the X node and the Y node. How then do we figure out the groups of synonyms?

#676. 17.21 Can you do precomputation to compute the next tallest bar on each side?

#677. 17.13 Will the recursive algorithm hit the same subproblems repeatedly? Can you optimize with a hash table?

#678. 17.14 What if, when you picked an element, you swapped elements around (as you do in quicksort) so that the elements below it would be located before the elements above it? If you did this repeatedly, could you find the smallest one million numbers?

#679. 16.6 Imagine you had the two arrays sorted and you were walking through them. If the pointer in the first array points to 3 and the pointer in the second array points to 9, what effect will moving the second pointer have on the difference of the pair?

#680. 17.12 To handle whether your recursive algorithm should return the start or the end of the linked list, you could try to pass a parameter down that acts as a flag. This won't work very well, though. The problem is that when you call `convert(current.left)`, you want to get the end of `left`'s linked list. This way you can join the end of the linked list to `current`. But, if `current` is someone else's right subtree, `convert(current)` needs to pass back the start of the linked list (which is actually the start of `current.left`'s linked list). Really, you need both the start and end of the linked list.

#681. 17.18 Consider the previously explained brute force solution. A bottleneck is repeatedly asking for the next instance of a particular character. Is there a way you can optimize this? You should be able to do this in `O(1)` time.

#682. 17.8 Try a recursive approach that just evaluates all possibilities.

#683. 17.4 Once you've identified that the least significant bit is a 0 (or a 1), you can rule out all the numbers without 0 as the least significant bit. How is this problem different from the earlier part?

#684. 17.23 Start with a brute force solution. Can you try the biggest possible square first?

#685. 16.18 Suppose you decide on a specific value for the "a" part of a pattern. How many possibilities are there for b?

#686. 17.9 When you add x to the list of the first k values, you can add `3x`, `5x`, and `7x` to some new list. How do you make this as optimal as possible? Would it make sense to keep multiple queues of values? Do you always need to insert `3x`, `5x`, and `7x`? Or, perhaps sometimes you need to insert only `7x`? You want to avoid seeing the same number twice.

#687. 16.19 Try recursion to count the number of water cells.

#688. 16.8 Consider dividing up a number into sequences of three digits.

#689. 17.19 Part 2: We could do both. If we know that `a + b = 87` and `a * b = 962`, then we can solve for a and b: `a = 13` and `b = 74`. But this will also result in having to multiply really large numbers. The product of all the numbers could be larger than 10^{157}. Is there a simpler calculation you can make?

#690. 16.11 Consider building a diving board. What are the choices you make?

#691. 17.18 Can you precompute the next instance of a particular character from each index? Try using a multi-dimensional array.

#692. 17.1 The carry will happen when you are doing 1 + 1. How do you apply the carry to the number?

#693. 17.21 As an alternative solution, think about it from the perspective of each bar. Each bar will have water on top of it. How much water will be on top of each bar?

#694. 16.25 Both a hash table and a doubly linked list would be useful. Can you combine the two?

#695. 17.23 The biggest possible square is NxN. So if you try that square first and it works, then you know that you've found the best square. Otherwise, you can try the next smallest square.

#696. 17.19 Part 2: Almost any "equation" we can come up with will work here (as long as it's not equivalent to a linear sum). It's just a matter of keeping this sum small.

#697. 16.23 It is not possible to divide 5^k evenly by 7. Does this mean that you can't implement `rand7()` with `rand5()`?

#698. 16.26 You can also maintain two stacks, one for the operators and one for the numbers. You push a number onto the stack every time you see it. What about the operators? When do you pop operators from the stack and apply them to the numbers?

#699. 17.8 Another way to think about the problem is this: if you had the longest sequence ending at each element `A[0]` through `A[n-1]`, could you use that to find the longest sequence ending at element `A[n-1]`?

#700. 16.11 Consider a recursive solution.

#701. 17.12 Many people get stuck at this point and aren't sure what to do. Sometimes they need the start of the linked list, and sometimes they need the end. A given node doesn't necessarily know what to return on its `convert` call. Sometimes the simple solution is easiest: always return both. What are some ways you could do this?

#702. 17.19 Part 2: Try a sum of squares of the values.

#703. 16.20 A trie might help us short-circuit. What if you stored the whole list of words in the trie?

#704. 17.7 Each connected subgraph represents a group of synonyms. To find each group, we can do repeated breadth-first (or depth-first) searches.

#705. 17.23 Describe the runtime of the brute force solution.

#706. 16.19 How can you make sure that you're not revisiting the same cells? Think about how breadth-first search or depth-first search on a graph works.

#707. 16.7 When `a > b`, then `a - b > 0`. Can you get the sign bit of `a - b`?

#708. 16.16 In order to be able to sort `MIDDLE` and have the whole array become sorted, you need `MAX(LEFT) <= MIN(MIDDLE and RIGHT)` and `MAX(LEFT and MIDDLE) <= MIN(RIGHT)`.

#709. 17.20 What if you used a heap? Or two heaps?

#710. 16.4 If you were calling `hasWon` multiple times, how might your solution change?

#711. 16.5 Each zero in `n!` corresponds to n being divisible by a factor of 10. What does that mean?

#712. 17.1 You can use an AND operation to compute the carry. What do you do with it?

#713. 17.5 Suppose, in this table, index `i` has `count(A, 0->i) = 3` and `count(B, 0->i) = 7`. This means that there are four more Bs than As. If you find a later spot `j` with the same difference (`count(B, 0->j) - count(A, 0->j)`), then this indicates a subarray with an equal number of As and Bs.

#714. 17.23 Can you do preprocessing to optimize this solution?

#715. 16.11 Once you have a recursive algorithm, think about the runtime. Can you make this faster? How?

#716. 16.1 Let `diff` be the difference between a and b. Can you use `diff` in some way? Then can you get rid of this temporary variable?

#717. 17.19 Part 2: You might need the quadratic formula. It's not a big deal if you don't remember it. Most people won't. Remember that there is such a thing as good enough.

#718. 16.18 Since the value of a determines the value of b (and vice versa) and either a or b must start at the beginning of the value, you should have only `O(n)` possibilities for how to split up the pattern.

#719. 17.12 You could return both the start and end of a linked list in multiple ways. You could return a two-element array. You could define a new data structure to hold the start and end. You could re-use the `BiNode` data structure. If you're working in a language that supports this (like Python), you could just return multiple values. You could solve the problem as a circular linked list, with the start's previous pointer pointing to the end (and then break the circular list in a wrapper method). Explore these solutions. Which one do you like most and why?

#720. 16.23 You can implement `rand7()` with `rand5()`, you just can't do it deterministically (such that you know it will definitely terminate after a certain number of calls). Given this, write a solution that works.

#721. 17.23 You should be able to do this in $O(N^3)$ time, where N is the length of one dimension of the square.

#722. 16.11 Consider memoization to optimize the runtime. Think carefully about what exactly you cache. What is the runtime? The runtime is closely related to the max size of the table.

#723. 16.19 You should have an algorithm that's $O(N^2)$ on an NxN matrix. If your algorithm isn't, consider if you've miscomputed the runtime or if your algorithm is suboptimal.

#724. 17.1 You might need to do the add/carry operation more than once. Adding `carry` to `sum` might cause new values to carry.

#725. 17.18 Once you have the precomputation solution figured out, think about how you can reduce the space complexity. You should be able to get it down to `O(SB)` time and `O(B)` space (where B is the size of the larger array and S is the size of the smaller array).

#726. 16.20 We're probably going to run this algorithm many times. If we did more preprocessing, is there a way we could optimize this?

#727. 16.18 You should be able to have an $O(n^2)$ algorithm.

#728. 16.7 Have you considered how to handle integer overflow in `a - b`?

#729. 16.5 Each factor of 10 in `n!` means `n!` is divisible by 5 and 2.

#730. 16.15 For ease and clarity in implementation, you might want to use other methods and classes.

#731. 17.18 Another way to think about it is this: Imagine you had a list of the indices where each item appeared. Could you find the first possible subsequence with all the elements? Could you find the second?

#732. 16.4 If you were designing this for an NxN board, how might your solution change?

#733. 16.5 Can you count the number of factors of 5 and 2? Do you need to count both?

#734. 17.21 Each bar will have water on top of it that matches the minimum of the tallest bar on the left and the tallest bar on the right. That is, `water_on_top[i] = min(tallest_bar(0->i), tallest_bar(i, n))`.

#735. 16.16 Can you expand the middle until the earlier condition is met?

#736. 17.23 When you're checking to see if a particular square is valid (all black borders), you check how many black pixels are above (or below) a coordinate and to the left (or right) of this coordinate. Can you precompute the number of black pixels above and to the left of a given cell?

#737. 16.1 You could also try using XOR.

#738. 17.22 What if you did a breadth-first search starting from both the source word and the destination word?

#739. 17.13 In real life, we would know that some paths will not lead to a word. For example, there are no words that start with `hellothisism`. Can we terminate early when going down a path that we know won't work?

#740. 16.11 There's an alternate, clever (and very fast) solution. You can actually do this in linear time without recursion. How?

#741. 17.18 Consider using a heap.

#742. 17.21 You should be able to solve this in `O(N)` time and `O(N)` space.

#743. 17.17 Alternatively, you could insert each of the smaller strings into the trie. How would this help you solve the problem? What is the runtime?

#744. 16.20 With preprocessing, we can actually get the lookup time down to `O(1)`.

#745. 16.5 Have you considered that 25 actually accounts for two factors of 5?

#746. 16.16 You should be able to solve this in `O(N)` time.

#747. 16.11 Think about it this way. You are picking K planks and there are two different types. All choices with 10 of the first type and 4 of the second type will have the same sum. Can you just iterate through all possible choices?

#748. 17.25 Can you use a trie to terminate early when a rectangle looks invalid?

#749. 17.13 For early termination, try a trie.

XIV

About the Author

Gayle Laakmann McDowell has a strong background in software development with extensive experience on both sides of the hiring table.

She has worked for Microsoft, Apple, and Google as a software engineer. She spent three years at Google, where she was one of the top interviewers and served on the hiring committee. She interviewed hundreds of candidates in the U.S. and abroad, assessed thousands of candidate interview packets for the hiring committee, and reviewed many more resumes.

As a candidate, she interviewed with—and received offers from—twelve tech companies, including Microsoft, Google, Amazon, IBM, and Apple.

Gayle founded CareerCup to enable candidates to perform at their best during these challenging interviews. CareerCup.com offers a database of thousands of interview questions from major companies and a forum for interview advice.

In addition to ***Cracking the Coding Interview***, Gayle has written other two books:

- ***Cracking the Tech Career: Insider Advice on Landing a Job at Google, Microsoft, Apple, or Any Top Tech Company*** provides a broader look at the interview process for major tech companies. It offers insight into how anyone, from college freshmen to marketing professionals, can position themselves for a career at one of these companies.
- ***Cracking the PM Interview: How to Land a Product Manager Job in Technology*** focuses on product management roles at startups and big tech companies. It offers strategies to break into these roles and teaches job seekers how to prepare for PM interviews.

Through her role with CareerCup, she consults with tech companies on their hiring process, leads technical interview training workshops, and coaches engineers at startups for acquisition interviews.

She holds bachelor's degree and master's degrees in computer science from the University of Pennsylvania and an MBA from the Wharton School.

She lives in Palo Alto, California, with her husband, two sons, dog, and computer science books. She still codes daily.

CPSIA information can be obtained
at www.ICGtesting.com
Printed in the USA
LVOW03s0711310516
490543LV00025B/265/P

9 780984 782857

Amazon.com's #1 Best-Selling Interview Book

CRACKING *the* CODING INTERVIEW

I am not a recruiter. I am a software engineer. And as such, I know what it's like to be asked to whip up brilliant algorithms on the spot and then write flawless code on a whiteboard. I've been through this—as a candidate and as an interviewer.

Cracking the Coding Interview, 6th Edition is here to help you through this process, teaching you what you need to know and enabling you to perform at your very best. I've coached and interviewed hundreds of software engineers. The result is this book.

Learn how to uncover the hints and hidden details in a question, discover how to break down a problem into manageable chunks, develop techniques to unstick yourself when stuck, learn (or re-learn) core computer science concepts, and practice on 189 interview questions and solutions.

These interview questions are real; they are not pulled out of computer science textbooks. They reflect what's truly being asked at the top companies, so that you can be as prepared as possible.

WHAT'S INSIDE?

- 189 programming interview questions, ranging from the basics to the trickiest algorithm problems.
- A walk-through of how to derive each solution, so that you can learn how to get there yourself.
- Hints on how to solve each of the 189 questions, just like what you would get in a real interview.
- Five proven strategies to tackle algorithm questions, so that you can solve questions you haven't seen.
- Extensive coverage of essential topics, such as big O time, data structures, and core algorithms.
- A "behind the scenes" look at how top companies, like Google and Facebook, hire developers.
- Techniques to prepare for and ace the "soft" side of the interview: behavioral questions.
- For interviewers and companies: details on what makes a good interview question and hiring process.

GAYLE LAAKMANN MCDOWELL

Gayle Laakmann McDowell is the founder and CEO of CareerCup and the author of Cracking the PM Interview and Cracking the Tech Career.

Gayle has a strong background in software development, having worked as a software engineer at Google, Microsoft, and Apple. At Google, she interviewed hundreds of software engineers and evaluated thousands of hiring packets as part of the hiring committee. She holds a B.S.E. and M.S.E. in computer science from the University of Pennsylvania and an MBA from the Wharton School.

She now consults with tech companies to improve their hiring process and with startups to prepare them for acquisition interviews.

ISBN 978-0-9847828-5-7
90000>
9 780984 782857

6TH EDITION