#	*Hints*	*题意 & Idea*
1.1	● Try a hash table. ● Could a bit vector be useful? ● Can you solve it in O(N log N) time? What might a solution like that look like?	
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? ● There is one solution that is 0(N log N) time. Another solution uses some space, but isO(N) time. ● Two strings that are permutations should have the same characters, but in different orders. Can you make the orders the same? ● Could a hash table be useful?	
1.3	● It's often easiest to modify strings by going from the end of the string to the beginning. ● You might find you need to know the number of spaces. Can you just count them?	
1.4	● You do not have to-and should not-generate all permutations. This would be very inefficient. ● Can you reduce the space usage by using a bit vector? ● Have you tried a hash table? You should be able to get this down to 0(N) time ● What characteristics would a string that is a permutation of a palindrome have?	
1.5	● Can you do all three checks in a single pass? ● Start with the easy thing. Can you check each of the conditions separately? ● What is the relationship between the "insert character" option and the "remove character" option? Do these need to be two separate checks?	
1.6	● Do the easy thing first. Compress the string, then compare the lengths. ● Be careful that you aren't repeatedly concatenating strings together. This can be very inefficient.	
1.7	● Try thinking about it layer by layer. Can you rotate a specific layer?	

	• 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?	
1.8	• If you just cleared the rows and columns as you found Os, you'd likely wind up clearing the whole matrix. Try finding the cells with zeros first before making any changes to the matrix. • 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? • 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 0(1) by using the matrix itself for data storage?	
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) bef o re the left half (wat). • 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. • Think about the earlier hint. Then think about what happens when you concatenate erbottlewat to itself. You get erbottlewaterbottlewat.	
2.1	• Have you tried a hash table? You should be able to do this in a single pass of the linked list. • Without extra space, you'll need O(N 2) time. Try using two pointers, where the second one searches ahead of the first one.	
2.2	• What if you knew the linked list size? What is the difference between finding the Kth-tolast element and finding the Xth element? • 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?	

	• If you don't know the linked list size, can you compute it? How does this impact the runtime? • Try implementing it recursively. If you could find the (K-l)th to last element, can you find the Kth element? • 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?	
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?	
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? • 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?	
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. • 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(l->5->9, 2->3->6->7) and sum (5->9, 3->6->7)? • Make sure you have considered linked lists that are not the same length. • Does your algorithm work on linked lists like 9->7->8 and 6->8->5? Double check that. • For the f o llow-up question: The issue is that when the linked lists aren't the same length, the head of one linked list might represent the 1 OOO's place while the other represents the 1 O'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?	
2.6	● A palindrome is something which is the same when written forwards and backwards. What if you reversed the linked list? ● Try using a stack. ● Assume you have the length of the linked list. Can you implement this recursively? ● 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? ● Go back to the previous hint. Remember: There are ways to return multiple values. You can do this with a new class.	
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. ● You can do this in O(A+B) time and 0(1) additional space. That is, you do not need a hash table (although you could do it with one). ● Examples will help you. Draw a picture of intersecting linked lists and two equivalent linked lists (by value) that do not intersect. ● Focus first on just identifying if there's an intersection. ● Observe that two intersecting linked lists will always have the same last node. Once they intersect, all the nodes after that will be equal. ● You can determine if two linked lists intersect by traversing to the end of each and comparing their tails. ● Now, you need to find where the linked lists intersect. Suppose the linked lists were the same length. How could you do this? ● If the two linked lists were the same length, you could traverse forward in each until you f o und an element in common. Now, how do you adjust this for lists of different lengths? ● Try using the difference between the lengths of the two linked lists.	

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. ● To identify if there's a cycle, try the "runner" approach described on page 93. Have one pointer move faster than the other. ● 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? ● 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?	
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. ● 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? ● If you want to allow for flexible divisions, you can shift stacks around. Can you ensure that all available capacity is used? ● Try thinking about the array as circular, such that the end of the array"wraps around" to the start of the array.	
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. ● What if we kept track of extra data at each stack node? What sort of data might make it easier to solve the problem? ● Consider having each node know the minimum of its "substack" (all the elements beneath it, including itself).	

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. ● 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.	
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? ● 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?	
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? ● 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? ● Keep the secondary stack in sorted order, with the biggest elements on the top. Use the primary stack f o r additional storage.	
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? ● Let's suppose we kept separate lists for dogs and cats. How would we find the oldest animal of any type? Be creative! ● 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?	
4.1	● Two well-known algorithms can do this. What	**Route Between Nodes.**

797	are the tradeoffs between them?	This problem can be solved by just simple graph traversal, such as depth-first search or breadth-first search. We start with one of the two nodes and, during traversal, check if the other node is found. We should mark any node found in the course of the algorithm as "already visited" to avoid cycles and repetition of the nodes. depth-first search is a bit simpler to implement since it can be done with simple recursion. Breadth-first search can also be useful to find the shortest path, whereas depth-first search may traverse one adjacent node very deeply before ever going onto the immediate neighbors.
4.2 **108**	● 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? ● 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? ● 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 f o r the function? Great! Then that's basically the entire function.	Given a sorted (increasing order) array with unique integer elements, write an algorithm to create a binary search tree with minimal height.
4.3 **102**	● Try modifying a graph search algorithm to track the depth from the root. ● A hash table or array that maps from level number to nodes at that level might also be useful. ● You should be able to come up with an algorithm involving both depth-first search and breadth-first search.	Given a binary tree, design an algorithm which creates a linked list of all the nodes at each depth
4.4 **110**	● Think about the definition of a balanced tree. Can you check that condition for a single node? Can you check it for every node? ● 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.	**Check Balanced**

	• What if you could modify the binary tree node class to allow a node to store the height of its subtree? • 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. • 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.	
4.5 **98**	• 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? • 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). • 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. • 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. • 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.	**Validate BST**
4.6 **285**	• Think about how an in-order traversal works and try to "reverse engineer" it. • 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?	Write an algorithm to find the "next" node (i.e., in-order successor) of a given node in a binary search tree. You may assume that each node has a link to its parent.

4.7 **210**	● 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? ● 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. ● Look at this graph. Is there any node you can identify that will definitely be okay to build first? ● 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? ● 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? ● 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?	**Build Order** Nodes with no incoming edges can be built immediately since they don't depend on anything.Once we've done that, it's irrelevant that some nodes are dependent on d and f since d and f have already been built. We can reflect this new state by removing d and f's outgoing edges. This solution takes O (P + D) time, where P is the number of projects and D is the number of dependency pairs. A cycle will happen if, while doing a DFS on a node, we run back into the same path. What we need therefore is a signal that indicates"I'm still processing this node, so if you see the node again, we have a problem. Like the earlier algorithm(inDegree), this solution(DFS) is O (P+D) time, where P is the number of projects and D is the number of dependency pairs.
4.8 **236** **235**	● 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. ● The first common ancestor is the deepest node such that p and q are both descendants. Think about how you might identify this node. ● How would you figure out if p is a descendent of a node n? ● 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? ● 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? ● 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	**First Common Ancestor.** If each node has a link to its parent, we could trace p and q's paths up until they intersect. This is essentially the same problem as question 2.7 which find the intersection of two linked lists. The "linked list" in this case is the path from each node up to the root.

	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? ● 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. ● 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.	
4.9	● What is the very first value that must be in each array? ● 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? ● 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. ● 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?	**Given a binary search tree with distinct elements, print all possible arrays that could have led to this tree.** **Solution**
4.10 **572**	● If T2 is a subtree of Tl, how will its in-order traversal compare to Tl's? What about its pre-order and post-order traversal? ● 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 preorder traversal, however, is much more indicative. ● You may have concluded that if T2. preorderTraversal () is a substring of Tl. preorderTraversal (), then T2 is a subtree of Tl. This is almost true, except that the trees could have duplicate values. Suppose Tl and T2 have	Tl and T2 are two very large binary trees, with Tl much bigger than T2. Create an algorithm to determine ifT2 is a subtree ofTl.

	all duplicate values but different structures. The pre-order traversals will look the same even though T2 is not a subtree of TI. How can you handle situations like this? ● 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. ● Alternatively, we can handle this problem recursively. Given a specific node within TI, can we check to see if its subtree matches T2?	
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. ● 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. ● As a naive "brute force" algorithm, can you use a tree traversal algorithm to implement this algorithm? What is the runtime of this? ● 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? ● 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.	**Random Node**

	• 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? • 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 1/3 even if there are 50+ nodes in the tree. Clearly, not all the nodes have probability ⅓ 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? • 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?	
4.12 **437** **112** **113**	• Try simplifying the problem. What if the path had to start at the root? • 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. • 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. • To extend this to paths that start anywhere, we can just repeat this process for all nodes. • 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.	**Design an algorithm to count the number of paths that sum to a given value. The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes).**

	• What work is duplicated in the current brute-force algorithm? • 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. • We are looking for subarrays with sum targetSum. Observe that we can track in constant time the value of runningSum i , where this is the sum from element O through element i. For a subarray of element i through element j to have sum targetSum, runningSum i -i + targetSum must equal 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? • Try using a hash table that maps from a runningSum value to the number of elements with this runningSum. • 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.	
8.1 **70**	• Approach this from the top down. What is the very last hop the child made? • If we knew the number of paths to each of the steps before step 100, could we compute the number of steps to 100? • 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)? • We multiply the values when it's "we do this then this:' We add them when it's "we do this or this:' • What is the runtime of this method? Think carefully. Can you optimize it?	**Triple Step**

	● Try memoization as a way to optimize an inefficient recursive program.	
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. ● Simplify this problem a bit by first figuring out if there's a path. Then, modify your algorithm to track the path. ● Think again about the efficiency of your algorithm. Can you optimize it?	**Robot in a Grid**
8.3	● Given a specific index and value, can you identify after it? ● Start with a brute force algorithm. ● Can you solve the problem in O(log N)? ● **Binary search has a runtime of O(log N).** Can you apply a form of binary search to the problem?	**Magic Index**
8.4	● How can you build all subsets of {a, b, c} from the subsets of {a, b}? ● 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}? ● 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}? ● You can also do this by mapping each subset to a binary number. The ith bit could represent a "boolean"flag for whether an element is in the set.	**Power Set**
8.5	● Think about multiplying 8 by 9 as counting the number of cells in a matrix with width 8 and height 9. ● Alternatively, if you're doing 9 * 7, you could do 4*7, double that, and then add 7. ● If you wanted to count the cells in an 8x9 matrix, you could count the cells in a 4x9 matrix and then double it. ● Think about how you might handle this for odd numbers. ● If there's duplicated work across different recursive calls,can you cache it? ● If you're doing 9*7 (both odd numbers), then you could do 4*7 and 5*7.	**Recursive Multiply**
8.6	● Observe that it doesn't really matter which tower	

	is the source, destination, or buffer. You can dof(3, X=0, Y=2, Z=l) by first doing f(2, X=0, Y=l, Z=2) (moving two disks from tower O to tower 1, using tower 2 as a buffer), then moving disk 3 from tower O to tower 2, then doing f(2, X=l, Y=2, Z=0) (moving two disks from tower 1 to tower 2, using tower O as a buffer). How does this process repeat? ● Try the Base Case and Build approach. ● 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? ● Think about moving the smallest disk from tower X=0 to towerY=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 isf(2, X=0, Y=2, Z=1). Given that you have a solution for f(l, X=0, Y=2, Z=l) and f(2, X=0, Y=2, Z=1),can you solve f(3, X=0, Y=2, Z=1)? ● 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 O 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?	
8.7	● Approach 2: To generate all permutations of abed, 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. ● Approach 1: You can create all permutations of abed by computing all permutations of abc and then inserting d into each possible location within those. ● Approach 2: If you had all permutations of two-character substrings, could you generate all permutations of three-character substrings? ● Approach 1: Suppose you had all permutations of abc. How can you use that to get all permutations of abed? ● Approach 1: The permutations of abc represent all ways of ordering abc. Now, we want to create all orderings of abed. Take a specific	**Permutations without Dups**

	ordering of abed, such as bdea. This bdea string represents an ordering of abe, too: Remove the d and you get bea. Given the string bca, can you create all the"related" orderings that include d, too? ● Approach 1: Given a string such as bca, you can create all permutations of abed that have {a, b, c} in the order bca by inserting d into each possible location: dbca, bdca, beda, bead. Given all permutations of abc, can you then create all permutations of abed? ● Approach 2: To generate a permutation of abed, 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? ● 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.	
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? ● If you haven't solved 8.7 yet, do that one first. ● Try getting the count of each character. For example, ABCMC has 3 As, 2 Cs, and 1 B. ● 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.	**Permutations with Duplicates**
8.9 **22**	● Try the Base Case and Build approach. ● 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? ● 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? ● 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) <=	**Parens**

	count(right parens). ● 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. ● Try to break it down into subproblems. ● 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? ● 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.	
8.10	● Think about this as a graph. ● You can implement this using depth-first search (or breadth-first search). Each adjacent pixel of the "right" color is a connected edge.	**Paint Fill**
8.11	● Try using memoization. ● Try breaking it down into subproblems. If you were making change, what is the first choice you would make? ● 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. ● If you were making change, the first choice you might make is how many quarters you need to use. ● Analyze your algorithm. Is there any repeated work? Can you optimize this?	
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? ● 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. ● We know that each row must have a queen. Can you try all possibilities?	**Eight Queens**
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	**Stack of Boxes:**

	on. ● Once you have a basic recursive algorithm implemented, think about if you can optimize it. Are there any repeated subproblems? ● Will sorting the boxes help in any way? ● 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. ● Think about the first decision you have to make. The first decision is which box will be at the bottom. ● Once we pick the box on the bottom, we need to pick the second box. Then the third box.	
8.14	● Look at your recursion. Do you have repeated calls anywhere? Can you memorize it? ● 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. ● Can we just try all possibilities? What would this look like? ● 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? ● The base case is when we have a single value, 1 or 0.	**Boolean Evaluation**
10.1	● Try moving from the end of the array to the beginning.	
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. ● Two words are anagrams if they contain the same characters but in different orders. How can you put characters in order? ● Can you leverage a standard sorting algorithm? ● Do you even need to truly "sort"? Or is just	

	reorganizing the list sufficient?	
10.3	● Can you modify binary search for this purpose? ● What is the runtime of your algorithm? What will happen if the array has duplicates?	
10.4	● Think about how binary search works. What will be the issue with just implementing binary search? ● 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? ● 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?	
10.5	● Try modifying binary search to handle this.	
10.6	● Think about merge sort versus quick sort. Wou ld one of them work well for this purpose?	
10.7	● Would a bit vector help? ● To do it with less memory, can you try multiple passes? ● Try using one pass to get it down to a range of values, and then a second pass to find a specific value.	
10.8	● Can you use a bit vector? ● Consider implementing your own bit vector class. It's a good exercise and an important part of this problem.	
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. ● Think about the previous hint in the context of rows. ● Can we use the previous hints to move up, down, left, and right around the rows and columns? ● What would happen if we tried to keep track of this using an array ? What are the pros and cons of this? ● If the value xis smaller than the start of the column, then it also can't be in any columns to the right. ● Start with a naive solution. (But hopefully not too naive. You should be able to use the fact	

	that the matrix is sorted.) ● We can do a binary search in each row. How long will this take? How can we do better? ● If you're considering a particular column, is there a way to quickly eliminate it (in some cases at least)? ● 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? ● 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. ● 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?	
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? ● Would it work well to use a binary search tree? ● Consider a binary search tree where each node stores some additional data.	
10.11	● You should be able to design an O(n) algorithm. ● 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? ● Imagine the array were sorted in ascending order. Is there any way you could "fix it"to be sorted into alternating peaks and valleys? ● Do you necessarily need the arrays to be sorted? Can you do it with an unsorted array? ● Try walking through a sorted array. Can you just swap elements until you have fixed the array? ● 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. ● 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.	