1

Arrays and Strings

Hopefully, all readers of this book are familiar with what arrays and strings are, so we won't bore you with such details. Instead, we'll focus on some of the more common techniques and issues with these data structures.

Please note that array questions and string questions are often interchangeable. That is, a question that this book states using an array may be asked instead as a string question, and vice versa.

Hash Tables

A hash table is a data structure that maps keys to values for highly efficient lookup. In a very simple implementation of a hash table, the hash table has an underlying array and a *hash function*. When you want to insert an object and its key, the hash function maps the key to an integer, which indicates the index in the array. The object is then stored at that index.

Typically, though, this won't quite work right. In the above implementation, the hash value of all possible keys must be unique, or we might accidentally overwrite data. The array would have to be extremely large—the size of all possible keys—to prevent such "collisions."

Instead of making an extremely large array and storing objects at index `hash(key)`, we can make the array much smaller and store objects in a linked list at index `hash(key) % array_length`. To get the object with a particular key, we must search the linked list for this key.

Alternatively, we can implement the hash table with a binary search tree. We can then guarantee an `O(log n)` lookup time, since we can keep the tree balanced. Additionally, we may use less space, since a large array no longer needs to be allocated in the very beginning.

Prior to your interview, we recommend you practice both implementing and using hash tables. They are one of the most common data structures for interviews, and it's almost

a sure bet that you will encounter them in your interview process.

Below is a simple Java example of working with a hash table.

```
1  public HashMap<Integer, Student> buildMap(Student[] students) {
2     HashMap<Integer, Student> map = new HashMap<Integer, Student>();
3     for (Student s : students) map.put(s.getId(), s);
4     return map;
5  }
```

Note that while the use of a hash table is sometimes explicitly required, more often than not, it's up to you to figure out that you need to use a hash table to solve the problem.

ArrayList (Dynamically Resizing Array)

An ArrayList, or a dynamically resizing array, is an array that resizes itself as needed while still providing O(1) access. A typical implementation is that when the array is full, the array doubles in size. Each doubling takes O(n) time, but happens so rarely that its amortized time is still O(1).

```
1  public ArrayList<String> merge(String[] words, String[] more) {
2     ArrayList<String> sentence = new ArrayList<String>();
3     for (String w : words) sentence.add(w);
4     for (String w : more) sentence.add(w);
5     return sentence;
6  }
```

StringBuffer

Imagine you were concatenating a list of strings, as shown below. What would the running time of this code be? For simplicity, assume that the strings are all the same length (call this x) and that there are n strings.

```
1  public String joinWords(String[] words) {
2     String sentence = "";
3     for (String w : words) {
4        sentence = sentence + w;
5     }
6     return sentence;
7  }
```

On each concatenation, a new copy of the string is created, and the two strings are copied over, character by character. The first iteration requires us to copy x characters. The second iteration requires copying 2x characters. The third iteration requires 3x, and so on. The total time therefore is $O(x + 2x + \ldots + nx)$. This reduces to $O(xn^2)$. (Why isn't it $O(xn^n)$? Because $1 + 2 + \ldots + n$ equals $n(n+1)/2$, or $O(n^2)$.)

`StringBuffer` can help you avoid this problem. `StringBuffer` simply creates an array of all the strings, copying them back to a string only when necessary.

```
1  public String joinWords(String[] words) {
2     StringBuffer sentence = new StringBuffer();
3     for (String w : words) {
```

```
4         sentence.append(w);
5     }
6     return sentence.toString();
7 }
```

A good exercise to practice strings, arrays, and general data structures is to implement your own version of `StringBuffer`.

Interview Questions

1.1 Implement an algorithm to determine if a string has all unique characters. What if you cannot use additional data structures?

pg 172

1.2 Implement a function void reverse(char* str) in C or C++ which reverses a null-terminated string.

pg 173

1.3 Given two strings, write a method to decide if one is a permutation of the other.

pg 174

1.4 Write a method to replace all spaces in a string with '%20'. You may assume that the string has sufficient space at the end of the string to hold the additional characters, and that you are given the "true" length of the string. (Note: if implementing in Java, please use a character array so that you can perform this operation in place.)

EXAMPLE

Input: `"Mr John Smith    "`

Output: `"Mr%20John%20Smith"`

pg 175

1.5 Implement a method to perform basic string compression using the counts of repeated characters. For example, the string `aabcccccaaa` would become `a2b1c5a3`. If the "compressed" string would not become smaller than the original string, your method should return the original string.

pg 176

1.6 Given an image represented by an `NxN` matrix, where each pixel in the image is 4 bytes, write a method to rotate the image by 90 degrees. Can you do this in place?

pg 179

1.7 Write an algorithm such that if an element in an `MxN` matrix is 0, its entire row and column are set to 0.

pg 180

1.8 Assume you have a method `isSubstring` which checks if one word is a substring of another. Given two strings, `s1` and `s2`, write code to check if `s2` is a rotation of `s1` using only one call to isSubstring (e.g., "`waterbottle`" is a rotation of "`erbottlewat`").

pg 181

Additional Questions: Bit Manipulation (#5.7), Object-Oriented Design (#8.10), Recursion (#9.3), Sorting and Searching (#11.6), C++ (#13.10), Moderate (#17.7, #17.8, #17.14)

2

Linked Lists

Because of the lack of constant time access and the frequency of recursion, linked list questions can stump many candidates. The good news is that there is comparatively little variety in linked list questions, and many problems are merely variants of well-known questions.

Linked list problems rely so much on the fundamental concepts, so it is essential that you can implement a linked list from scratch. We have provided the code below.

Creating a Linked List

The code below implements a very basic singly linked list.

```
1  class Node {
2    Node next = null;
3    int data;
4
5    public Node(int d) {
6      data = d;
7    }
8
9    void appendToTail(int d) {
10     Node end = new Node(d);
11     Node n = this;
12     while (n.next != null) {
13       n = n.next;
14     }
15     n.next = end;
16   }
17 }
```

Remember that when you're discussing a linked list in an interview, you must understand whether it is a singly linked list or a doubly linked list.

Deleting a Node from a Singly Linked List

Deleting a node from a linked list is fairly straightforward. Given a node n, we find the previous node prev and set prev.next equal to n.next. If the list is doubly linked, we must also update n.next to set n.next.prev equal to n.prev. The important things to remember are (1) to check for the null pointer and (2) to update the head or tail pointer as necessary.

Additionally, if you are implementing this code in C, C++ or another language that requires the developer to do memory management, you should consider if the removed node should be deallocated.

```
1  Node deleteNode(Node head, int d) {
2     Node n = head;
3
4     if (n.data == d) {
5        return head.next; /* moved head */
6     }
7
8     while (n.next != null) {
9        if (n.next.data == d) {
10          n.next = n.next.next;
11          return head; /* head didn't change */
12       }
13       n = n.next;
14    }
15    return head;
16 }
```

The "Runner" Technique

The "runner" (or second pointer) technique is used in many linked list problems. The runner technique means that you iterate through the linked list with two pointers simultaneously, with one ahead of the other. The "fast" node might be ahead by a fixed amount, or it might be hopping multiple nodes for each one node that the "slow" node iterates through.

For example, suppose you had a linked list $a_1->a_2->...->a_n->b_1->b_2->...->b_n$ and you wanted to rearrange it into $a_1->b_1->a_2->b_2->...->a_n->b_n$. You do not know the length of the linked list (but you do know that the length is an even number).

You could have one pointer p1 (the fast pointer) move every two elements for every one move that p2 makes. When p1 hits the end of the linked list, p2 will be at the midpoint. Then, move p1 back to the front and begin "weaving" the elements. On each iteration, p2 selects an element and inserts it after p1.

Recursive Problems

A number of linked list problems rely on recursion. If you're having trouble solving a

linked list problem, you should explore if a recursive approach will work. We won't go into depth on recursion here, since a later chapter is devoted to it.

However, you should remember that recursive algorithms take at least O(n) space, where n is the depth of the recursive call. All recursive algorithms *can* be implemented iteratively, although they may be much more complex.

Interview Questions

2.1 Write code to remove duplicates from an unsorted linked list.

FOLLOW UP

How would you solve this problem if a temporary buffer is not allowed?

pg 184

2.2 Implement an algorithm to find the kth to last element of a singly linked list.

pg 185

2.3 Implement an algorithm to delete a node in the middle of a singly linked list, given only access to that node.

EXAMPLE

Input: the node c from the linked list a->b->c->d->e

Result: nothing is returned, but the new linked list looks like a->b->d->e

pg 187

2.4 Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x.

pg 188

2.5 You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in *reverse* order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.

EXAMPLE

Input: (7-> 1 -> 6) + (5 -> 9 -> 2). That is, 617 + 295.

Output: 2 -> 1 -> 9. That is, 912.

FOLLOW UP

Suppose the digits are stored in forward order. Repeat the above problem.

EXAMPLE

Input: (6 -> 1 -> 7) + (2 -> 9 -> 5). That is, 617 + 295.

Output: 9 -> 1 -> 2. That is, 912.

pg 190

2.6 Given a circular linked list, implement an algorithm which returns the node at the beginning of the loop.

DEFINITION
Circular linked list: A (corrupt) linked list in which a node's next pointer points to an earlier node, so as to make a loop in the linked list.

EXAMPLE

Input: A -> B -> C -> D -> E -> C [the same C as earlier]

Output: C

pg 193

2.7 Implement a function to check if a linked list is a palindrome.

pg 196

Additional Questions: Trees and Graphs (#4.4), Object-Oriented Design (#8.10), Scalability and Memory Limits (#10.7), Moderate (#17.13)

3

Stacks and Queues

Like linked list questions, questions on stacks and queues will be much easier to handle if you are comfortable with the ins and outs of the data structure. The problems can be quite tricky though. While some problems may be slight modifications on the original data structure, others have much more complex challenges.

Implementing a Stack

Recall that a stack uses the LIFO (last-in first-out) ordering. That is, like a stack of dinner plates, the most recent item added to the stack is the first item to be removed.

We have provided simple sample code to implement a stack. Note that a stack can also be implemented using a linked list. In fact, they are essentially the same thing, except that a stack usually prevents the user from "peeking" at items below the top node.

```
1  class Stack {
2     Node top;
3
4     Object pop() {
5        if (top != null) {
6           Object item = top.data;
7           top = top.next;
8           return item;
9        }
10       return null;
11    }
12
13    void push(Object item) {
14       Node t = new Node(item);
15       t.next = top;
16       top = t;
17    }
18
19    Object peek() {
20       return top.data;
21    }
22 }
```

Implementing a Queue

A queue implements FIFO (first-In first-out) ordering. Like a line or queue at a ticket stand, items are removed from the data structure in the same order that they are added.

A queue can also be implemented with a linked list, with new items added to the tail of the linked list.

```
1  class Queue {
2     Node first, last;
3
4     void enqueue(Object item) {
5        if (first == null) {
6           last = new Node(item);
7           first = last;
8        } else {
9           last.next = new Node(item);
10          last = last.next;
11       }
12    }
13
14    Object dequeue() {
15       if (first != null) {
16          Object item = first.data;
17          first = first.next;
18          return item;
19       }
20       return null;
21    }
22 }
```

Interview Questions

3.1 Describe how you could use a single array to implement three stacks.

pg 202

3.2 How would you design a stack which, in addition to push and pop, also has a function `min` which returns the minimum element? Push, pop and `min` should all operate in O(1) time.

pg 206

3.3 Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure `SetOfStacks` that mimics this. `SetOfStacks` should be composed of several stacks and should create a new stack once the previous one exceeds capacity. `SetOfStacks.push()` and `SetOfStacks.pop()` should behave identically to a single stack (that is, pop() should return the same values as it would if there were just a single stack).

FOLLOW UP

Implement a function `popAt(int index)` which performs a pop operation on a specific sub-stack.

pg 208

3.4 In the classic problem of the Towers of Hanoi, you have 3 towers and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (i.e., each disk sits on top of an even larger one). You have the following constraints:

(1) Only one disk can be moved at a time.

(2) A disk is slid off the top of one tower onto the next tower.

(3) A disk can only be placed on top of a larger disk.

Write a program to move the disks from the first tower to the last using stacks.

pg 211

3.5 Implement a MyQueue class which implements a queue using two stacks.

pg 213

3.6 Write a program to sort a stack in ascending order (with biggest items on top). You may use at most one additional stack to hold items, but you may not copy the elements into any other data structure (such as an array). The stack supports the following operations: `push`, `pop`, `peek`, and `isEmpty`.

pg 215

3.7 An animal shelter holds only dogs and cats, and operates on a strictly "first in, first out" basis. People must adopt either the "oldest" (based on arrival time) of all animals at the shelter, or they can select whether they would prefer a dog or a cat (and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structures to maintain this system and implement operations such as `enqueue`, `dequeueAny`, `dequeueDog` and `dequeueCat`. You may use the built-in `LinkedList` data structure.

pg 217

Additional Questions: Linked Lists (#2.7), Mathematics and Probability (#7.7)

4

Trees and Graphs

Many interviewees find trees and graphs problems to be some of the trickiest. Searching the data structure is more complicated than in a linearly organized data structure like an array or linked list. Additionally, the worst case and average case time may vary wildly, and we must evaluate both aspects of any algorithm. Fluency in implementing a tree or graph from scratch will prove essential.

Potential Issues to Watch Out For

Trees and graphs questions are ripe for ambiguous details and incorrect assumptions. Be sure to watch out for the following issues and seek clarification when necessary.

Binary Tree vs. Binary Search Tree

When given a binary tree question, many candidates assume that the interviewer means binary *search* tree. Be sure to ask whether or not the tree is a binary search tree. A binary search tree imposes the condition that, for all nodes, the left children are less than or equal to the current node, which is less than all the right nodes.

Balanced vs. Unbalanced

While many trees are balanced, not all are. Ask your interviewer for clarification on this issue. If the tree is unbalanced, you should describe your algorithm in terms of both the average and the worst case time. Note that there are multiple ways to balance a tree, and balancing a tree implies only that the depth of subtrees will not vary by more than a certain amount. It does not mean that the left and right subtrees are exactly the same size.

Full and Complete

Full and complete trees are trees in which all leaves are at the bottom of the tree, and all non-leaf nodes have exactly two children. Note that full and complete trees are *extremely* rare, as a tree must have exactly $2^n - 1$ nodes to meet this condition.

Binary Tree Traversal

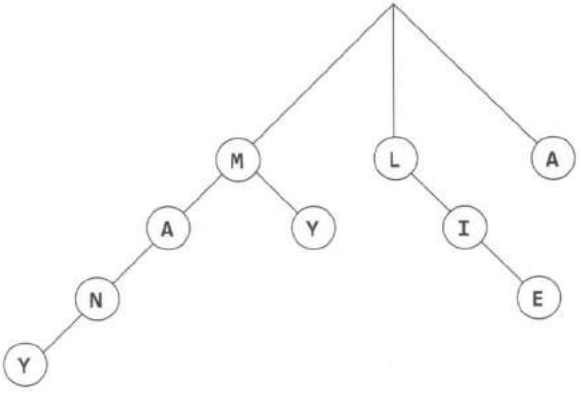
Prior to your interview, you should be comfortable implementing in-order, post-order, and pre-order traversal. The most common of these, in-order traversal, works by visiting the left side, then the current node, then the right.

Tree Balancing: Red-Black Trees and AVL Trees

Though learning how to implement a balanced tree may make you a better software engineer, it's very rarely asked during an interview. You should be familiar with the runtime of operations on balanced trees, and vaguely familiar with how you might balance a tree. The details, however, are probably unnecessary for the purposes of an interview.

Tries

A trie is a variant of an n-ary tree in which characters are stored at each node. Each path down the tree may represent a word. A simple trie might look something like:



Graph Traversal

While most candidates are reasonably comfortable with binary tree traversal, graph traversal can prove more challenging. Breadth First Search is especially difficult.

Note that Breadth First Search (BFS) and Depth First Search (DFS) are usually used in different scenarios. DFS is typically the easiest if we want to visit every node in the graph, or at least visit every node until we find whatever we're looking for. However, if we have a very large tree and want to be prepared to quit when we get too far from the original node, DFS can be problematic; we might search thousands of ancestors of the node, but never even search all of the node's children. In these cases, BFS is typically preferred.

Depth First Search (DFS)

In DFS, we visit a node `r` and then iterate through each of `r`'s adjacent nodes. When visiting a node `n` that is adjacent to `r`, we visit all of `n`'s adjacent nodes before going

on to r's other adjacent nodes. That is, n is exhaustively searched before r moves on to searching its other children.

Note that pre-order and other forms of tree traversal are a form of DFS. The key difference is that when implementing this algorithm for a graph, we must check if the node has been visited. If we don't, we risk getting stuck in infinite loop.

The pseudocode below implements DFS.

```
1  void search(Node root) {
2     if (root == null) return;
3     visit(root);
4     root.visited = true;
5     foreach (Node n in root.adjacent) {
6        if (n.visited == false) {
7           search(n);
8        }
9     }
10 }
```

Breadth First Search (BFS)

BFS is considerably less intuitive, and most interviewees struggle with it unless they are already familiar with the implementation.

In BFS, we visit each of a node r's adjacent nodes before searching any of r's "grandchildren." An iterative solution involving a queue usually works best.

```
1  void search(Node root) {
2     Queue queue = new Queue();
3     root.visited = true;
4     visit(root);
5     queue.enqueue(root); // Add to end of queue
6
7     while (!queue.isEmpty()) {
8        Node r = queue.dequeue(); // Remove from front of queue
9        foreach (Node n in r.adjacent) {
10          if (n.visited == false) {
11             visit(n);
12             n.visited = true;
13             queue.enqueue(n);
14          }
15       }
16    }
17 }
```

If you are asked to implement BFS, the key thing to remember is the use of the queue. The rest of the algorithm flows from this fact.

Interview Questions

4.1 Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.

pg 220

4.2 Given a directed graph, design an algorithm to find out whether there is a route between two nodes.

pg 221

4.3 Given a sorted (increasing order) array with unique integer elements, write an algorithm to create a binary search tree with minimal height.

pg 222

4.4 Given a binary tree, design an algorithm which creates a linked list of all the nodes at each depth (e.g., if you have a tree with depth D, you'll have D linked lists).

pg 224

4.5 Implement a function to check if a binary tree is a binary search tree.

pg 225

4.6 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.

pg 229

4.7 Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.

pg 230

4.8 You have two very large binary trees: T1, with millions of nodes, and T2, with hundreds of nodes. Create an algorithm to decide if T2 is a subtree of T1.

A tree T2 is a subtree of T1 if there exists a node n in T1 such that the subtree of n is identical to T2. That is, if you cut off the tree at node n, the two trees would be identical.

pg 235

4.9 You are given a binary tree in which each node contains a value. Design an algorithm to print all paths which sum to a given value. The path does not need to start or end at the root or a leaf.

pg 237

Additional Questions: Scalability and Memory Limits (#10.2, #10.5), Sorting and Searching (#11.8), Moderate (#17.13, #17.14), Hard (#18.6, #18.8, #18.9, #18.10, #18.13)

Concepts and Algorithms

Interview Questions and Advice

5

Bit Manipulation

Bit manipulation is used in a variety of problems. Sometimes, the question explicitly calls for bit manipulation, while at other times, it's simply a useful technique to optimize your code. You should be comfortable with bit manipulation by hand, as well as with code. But be very careful; it's easy to make little mistakes on bit manipulation problems. Make sure to test your code thoroughly after you're done writing it, or even while writing it.

Bit Manipulation By Hand

The practice exercises below will be useful if you have the oh-so-common fear of bit manipulation. When you get stuck or confused, try to work these operations through as a base 10 number. You can then apply the same process to a binary number.

Remember that ^ indicates an XOR operation, and ~ is a not (negation) operation. For simplicity, assume that these are four-bit numbers. The third column can be solved manually, or with "tricks" (described below).

`0110 + 0010`	`0011 * 0101`	`0110 + 0110`
`0011 + 0010`	`0011 * 0011`	`0100 * 0011`
`0110 - 0011`	`1101 >> 2`	`1101 ^ (~1101)`
`1000 - 0110`	`1101 ^ 0101`	`1011 & (~0 << 2)`

Solutions: line 1 (1000, 1111, 1100); line 2 (0101, 1001, 1100); line 3 (0011, 0011, 1111); line 4 (0010, 1000, 1000).

The tricks in Column 3 are as follows:

1. `0110 + 0110` is equivalent to `0110 * 2`, which is equivalent to shifting `0110` left by 1.
2. Since `0100` equals 4, we are just multiplying `0011` by 4. Multiplying by 2^n just shifts a number by n. We shift `0011` left by 2 to get `1100`.
3. Think about this operation bit by bit. If you XOR a bit with its own negated value, you will always get 1. Therefore, the solution to `a^(~a)` will be a sequence of 1s.

4. An operation like x & (~0 << n) clears the n rightmost bits of x. The value ~0 is simply a sequence of 1s, so by shifting it left by n, we have a bunch of ones followed by n zeros. By doing an AND with x, we clear the rightmost n bits of x.

For more problems, open the Windows calculator and go to View > Programmer. From this application, you can perform many binary operations, including AND, XOR, and shifting.

Bit Facts and Tricks

In solving bit manipulation problems, it's useful to understand the following facts. Don't just memorize them though; think deeply about why each of these is true. We use "1s" and "0s" to indicate a sequence of 1s or 0s, respectively.

```
x ^ 0s = x        x & 0s = 0        x | 0s = x
x ^ 1s = ~x       x & 1s = x        x | 1s = 1s
 x ^ x = 0         x & x = x         x | x = x
```

To understand these expressions, recall that these operations occur bit-by-bit, with what's happening on one bit never impacting the other bits. This means that if one of the above statements is true for a single bit, then it's true for a sequence of bits.

Common Bit Tasks: Get, Set, Clear, and Update Bit

The following operations are very important to know, but do not simply memorize them. Memorizing leads to mistakes that are impossible to recover from. Rather, understand *how* to implement these methods, so that you can implement these, and other, bit problems.

Get Bit

This method shifts 1 over by i bits, creating a value that looks like 00010000. By performing an AND with num, we clear all bits other than the bit at bit i. Finally, we compare that to 0. If that new value is not zero, then bit i must have a 1. Otherwise, bit i is a 0.

```
1  boolean getBit(int num, int i) {
2     return ((num & (1 << i)) != 0);
3  }
```

Set Bit

SetBit shifts 1 over by i bits, creating a value like 00010000. By performing an OR with num, only the value at bit i will change. All other bits of the mask are zero and will not affect num.

```
1  int setBit(int num, int i) {
2     return num | (1 << i);
3  }
```

Clear Bit

This method operates in almost the reverse of `setBit`. First, we create a number like `11101111` by creating the reverse of it (`00010000`) and negating it. Then, we perform an AND with `num`. This will clear the `i`th bit and leave the remainder unchanged.

```
1  int clearBit(int num, int i) {
2     int mask = ~(1 << i);
3     return num & mask;
4  }
```

To clear all bits from the most significant bit through `i` (inclusive), we do:

```
1  int clearBitsMSBthroughI(int num, int i) {
2     int mask = (1 << i) - 1;
3     return num & mask;
4  }
```

To clear all bits from `i` through 0 (inclusive), we do:

```
1  int clearBitsIthrough0(int num, int i) {
2     int mask = ~((1 << (i+1)) - 1);
3     return num & mask;
4  }
```

Update Bit

This method merges the approaches of `setBit` and `clearBit`. First, we clear the bit at position `i` by using a mask that looks like `11101111`. Then, we shift the intended value, `v`, left by `i` bits. This will create a number with bit `i` equal to `v` and all other bits equal to `0`. Finally, we OR these two numbers, updating the `i`th bit if `v` is `1` and leaving it as `0` otherwise.

```
1  int updateBit(int num, int i, int v) {
2     int mask = ~(1 << i);
3     return (num & mask) | (v << i);
4  }
```

Interview Questions

5.1 You are given two 32-bit numbers, `N` and `M`, and two bit positions, `i` and `j`. Write a method to insert `M` into `N` such that `M` starts at bit `j` and ends at bit `i`. You can assume that the bits `j` through `i` have enough space to fit all of `M`. That is, if `M = 10011`, you can assume that there are at least 5 bits between `j` and `i`. You would not, for example, have `j = 3` and `i = 2`, because `M` could not fully fit between bit 3 and bit 2.

EXAMPLE

Input: `N = 10000000000, M = 10011, i = 2, j = 6`

Output: `N = 10001001100`

pg 242

5.2 Given a real number between 0 and 1 (e.g., 0.72) that is passed in as a double, print the binary representation. If the number cannot be represented accurately in binary with at most 32 characters, print "ERROR."

pg 243

5.3 Given a positive integer, print the next smallest and the next largest number that have the same number of 1 bits in their binary representation.

pg 244

5.4 Explain what the following code does: `((n & (n-1)) == 0)`.

pg 250

5.5 Write a function to determine the number of bits required to convert integer A to integer B.

EXAMPLE

Input: 31, 14

Output: 2

pg 250

5.6 Write a program to swap odd and even bits in an integer with as few instructions as possible (e.g., bit 0 and bit 1 are swapped, bit 2 and bit 3 are swapped, and so on).

pg 251

5.7 An array A contains all the integers from 0 to n, except for one number which is missing. In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the `j`th bit of `A[i]`," which takes constant time. Write code to find the missing integer. Can you do it in `O(n)` time?

pg 252

5.8 A monochrome screen is stored as a *single* array of bytes, allowing eight consecutive pixels to be stored in one byte. The screen has width w, where w is divisible by 8 (that is, no byte will be split across rows). The height of the screen, of course, can be derived from the length of the array and the width. Implement a function `drawHorizontalLine(byte[] screen, int width, int x1, int x2, int y)` which draws a horizontal line from `(x1, y)` to `(x2, y)`.

pg 255

Additional Questions: Arrays and Strings (#1.1, #1.7), Recursion (#9.4, #9.11), Scalability and Memory Limits (#10.3, #10.4), C++ (#13.9), Moderate (#17.1, #17.4), Hard (#18.1)

6

Brain Teasers

Brain teasers are some of the most hotly debated questions, and many companies have policies banning them. Unfortunately, even when these questions are banned, you still may find yourself being asked a brain teaser. Why? Because no one can agree on a definition of what a brain teaser is.

The good news is that if you are asked a brain teaser, it's likely to be a reasonably fair one. It probably won't rely on a trick of wording, and it can almost always be logically deduced. Many brain teasers even have their foundations in mathematics or computer science.

We'll go through some common approaches for tackling brain teasers.

Start Talking

Don't panic when you get a brain teaser. Like algorithm questions, interviewers want to see how you tackle a problem; they don't expect you to immediately know the answer. Start talking, and show the interviewer how you approach a problem.

Develop Rules and Patterns

In many cases, you will find it useful to write down "rules" or patterns that you discover while solving the problem. And yes, you really should write these down—it will help you remember them as you solve the problem. Let's demonstrate this approach with an example.

You have two ropes, and each takes exactly one hour to burn. How would you use them to time exactly 15 minutes? Note that the ropes are of uneven densities, so half the rope length-wise does not necessarily take half an hour to burn.

Tip: Stop here and spend some time trying to solve this problem on your own. If you absolutely must, read through this section for hints—but do so slowly. Every paragraph will get you a bit closer to the solution.

From the statement of the problem, we immediately know that we can time one hour.

We can also time two hours, by lighting one rope, waiting until it is burnt, and then lighting the second. We can generalize this into a rule.

Rule 1: Given a rope that takes x minutes to burn and another that takes y minutes, we can time x+y minutes.

What else can we do with the rope? We can probably assume that lighting a rope in the middle (or anywhere other than the ends) won't do us much good. The flames would expand in both directions, and we have no idea how long it would take to burn.

However, we can light a rope at both ends. The two flames would meet after 30 minutes.

Rule 2: Given a rope that takes x minutes to burn, we can time x/2 minutes.

We now know that we can time 30 minutes using a single rope. This also means that we can remove 30 minutes of burning time from the second rope, by lighting rope 1 on both ends and rope 2 on just one end.

Rule 3: If rope 1 takes x minutes to burn and rope 2 takes y minutes, we can turn rope 2 into a rope that takes (y-x) minutes or (y-x/2) minutes.

Now, let's piece all of these together. We can turn rope 2 into a rope with 30 minutes of burn time. If we then light rope 2 on the other end (see rule 2), rope 2 will be done after 15 minutes.

From start to end, our approach is as follows:

1. Light rope 1 at both ends and rope 2 at one end.
2. When the two flames on Rope 1 meet, 30 minutes will have passed. Rope 2 has 30 minutes left of burn-time.
3. At that point, light Rope 2 at the other end.
4. In exactly fifteen minutes, Rope 2 will be completely burnt.

Note how solving this problem is made easier by listing out what you've learned and what "rules" you've discovered.

Worst Case Shifting

Many brain teasers are worst-case minimization problems, worded either in terms of *minimizing* an action or in doing something at most a specific number of times. A useful technique is to try to "balance" the worst case. That is, if an early decision results in a skewing of the worst case, we can sometimes change the decision to balance out the worst case. This will be clearest when explained with an example.

The "nine balls" question is a classic interview question. You have nine balls. Eight are of the same weight, and one is heavier. You are given a balance which tells you only whether the left side or the right side is heavier. Find the heavy ball in just two uses of the scale.

A first approach is to divide the balls in sets of four, with the ninth ball sitting off to the side. The heavy ball is in the heavier set. If they are the same weight, then we know that the ninth ball is the heavy one. Replicating this approach for the remaining sets would result in a worst case of three weighings—one too many!

This is an imbalance in the worst case: the ninth ball takes just one weighing to discover if it's heavy, whereas others take three. If we *penalize* the ninth ball by putting more balls off to the side, we can lighten the load on the others. This is an example of "worst case balancing."

If we divide the balls into sets of three items each, we will know after just one weighing which set has the heavy one. We can even formalize this into a *rule*: given N balls, where N is divisible by 3, one use of the scale will point us to a set of N/3 balls with the heavy ball.

For the final set of three balls, we simply repeat this: put one ball off to the side and weigh two. Pick the heavier of the two. Or, if the balls are the same weight, pick the third one.

Algorithm Approaches

If you're stuck, consider applying one of the five approaches for solving algorithm questions. Brain teasers are often nothing more than algorithm questions with the technical aspects removed. Examplify, Simplify and Generalize, Pattern Matching, and Base Case and Build can be especially useful.

Interview Questions

6.1 You have 20 bottles of pills. 19 bottles have 1.0 gram pills, but one has pills of weight 1.1 grams. Given a scale that provides an exact measurement, how would you find the heavy bottle? You can only use the scale once.

pg 258

6.2 There is an 8x8 chess board in which two diagonally opposite corners have been cut off. You are given 31 dominos, and a single domino can cover exactly two squares. Can you use the 31 dominos to cover the entire board? Prove your answer (by providing an example or showing why it's impossible).

pg 258

6.3 You have a five-quart jug, a three-quart jug, and an unlimited supply of water (but no measuring cups). How would you come up with exactly four quarts of water? Note that the jugs are oddly shaped, such that filling up exactly "half" of the jug would be impossible.

pg 259

6.4 A bunch of people are living on an island, when a visitor comes with a strange order: all blue-eyed people must leave the island as soon as possible. There will be a flight out at 8:00pm every evening. Each person can see everyone else's eye color, but they do not know their own (nor is anyone allowed to tell them). Additionally, they do not know how many people have blue eyes, although they do know that at least one person does. How many days will it take the blue-eyed people to leave?

pg 260

6.5 There is a building of 100 floors. If an egg drops from the Nth floor or above, it will break. If it's dropped from any floor below, it will not break. You're given two eggs. Find N, while minimizing the number of drops for the worst case.

pg 261

6.6 There are 100 closed lockers in a hallway. A man begins by opening all 100 lockers. Next, he closes every second locker. Then, on his third pass, he toggles every third locker (closes it if it is open or opens it if it is closed). This process continues for 100 passes, such that on each pass `i`, the man toggles every `i`th locker. After his 100th pass in the hallway, in which he toggles only locker #100, how many lockers are open?

pg 262

7

Mathematics and Probability

Although many mathematical problems given during an interview read as brain teasers, most can be tackled with a logical, methodical approach. They are typically rooted in the rules of mathematics or computer science, and this knowledge can facilitate either solving the problem or validating your solution. We'll cover the most relevant mathematical concepts in this section.

Prime Numbers

As you probably know, every positive integer can be decomposed into a product of primes. For example:

$$84 = 2^2 * 3^1 * 5^0 * 7^1 * 11^0 * 13^0 * 17^0 * \ldots$$

Note that many of these primes have an exponent of zero.

Divisibility

The prime number law stated above means that, in order for a number x to divide a number y (written x\y, or mod(y, x) = 0), all primes in x's prime factorization must be in y's prime factorization. Or, more specifically:

Let $x = 2^{j0} * 3^{j1} * 5^{j2} * 7^{j3} * 11^{j4} * \ldots$

Let $y = 2^{k0} * 3^{k1} * 5^{k2} * 7^{k3} * 11^{k4} * \ldots$

If x\y, then for all i, ji <= ki.

In fact, the greatest common divisor of x and y will be:

$$gcd(x, y) = 2^{min(j0, k0)} * 3^{min(j1, k1)} * 5^{min(j2, k2)} * \ldots$$

The least common multiple of x and y will be:

$$lcm(x, y) = 2^{max(j0, k0)} * 3^{max(j1, k1)} * 5^{max(j2, k2)} * \ldots$$

As a fun exercise, stop for a moment and think what would happen if you did gcd * lcm:

$$gcd * lcm = 2^{min(j0, k0)} * 2^{max(j0, k0)} * 3^{min(j1, k1)} * 3^{max(j1, k1)} * \ldots$$

$$= 2^{\min(j0,\ k0)\ +\ \max(j0,\ k0)} * 3^{\min(j1,\ k1)\ +\ \max(j1,\ k1)} * \ldots$$
$$= 2^{j0\ +\ k0} * 3^{j1\ +\ k1} * \ldots$$
$$= 2^{j0} * 2^{k0} * 3^{j1} * 3^{k1} * \ldots$$
$$= xy$$

Checking for Primality

This question is so common that we feel the need to specifically cover it. The naive way is to simply iterate from 2 through `n-1`, checking for divisibility on each iteration.

```
1  boolean primeNaive(int n) {
2     if (n < 2) {
3        return false;
4     }
5     for (int i = 2; i < n; i++) {
6        if (n % i == 0) {
7           return false;
8        }
9     }
10    return true;
11 }
```

A small but important improvement is to iterate only up through the square root of n.

```
1  boolean primeSlightlyBetter(int n) {
2     if (n < 2) {
3        return false;
4     }
5     int sqrt = (int) Math.sqrt(n);
6     for (int i = 2; i <= sqrt; i++) {
7        if (n % i == 0) return false;
8     }
9     return true;
10 }
```

The `sqrt` is sufficient because, for every number a which divides n evenly, there is a complement b, where `a * b = n`. If `a > sqrt`, then `b < sqrt` (since `sqrt * sqrt = n`). We therefore don't need a to check n's primality, since we would have already checked with b.

Of course, in reality, all we *really* need to do is to check if n is divisible by a prime number. This is where the Sieve of Eratosthenes comes in.

Generating a List of Primes: The Sieve of Eratosthenes

The Sieve of Eratosthenes is a highly efficient way to generate a list of primes. It works by recognizing that all non-prime numbers are divisible by a prime number.

We start with a list of all the numbers up through some value `max`. First, we cross off all numbers divisible by 2. Then, we look for the next prime (the next non-crossed off number) and cross off all numbers divisible by it. By crossing off all numbers divisible by 2, 3, 5, 7, 11, and so on, we wind up with a list of prime numbers from 2 through `max`.

The code below implements the Sieve of Eratosthenes.

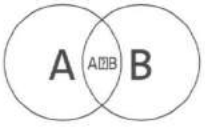
```
boolean[] sieveOfEratosthenes(int max) {
   boolean[] flags = new boolean[max + 1];
   int count = 0;

   init(flags); // Set all flags to true other than 0 and 1
   int prime = 2;

   while (prime <= Math.sqrt(max)) {
      /* Cross off remaining multiples of prime */
      crossOff(flags, prime);

      /* Find next value which is true */
      prime = getNextPrime(flags, prime);

      if (prime >= flags.length) {
         break;
      }
   }

   return flags;
}

void crossOff(boolean[] flags, int prime) {
   /* Cross off remaining multiples of prime. We can start with
    * (prime*prime), because if we have a k * prime, where
    * k < prime, this value would have already been crossed off in
    * a prior iteration. */
   for (int i = prime * prime; i < flags.length; i += prime) {
      flags[i] = false;
   }
}

int getNextPrime(boolean[] flags, int prime) {
   int next = prime + 1;
   while (next < flags.length && !flags[next]) {
      next++;
   }
   return next;
}
```

Of course, there are a number of optimizations that can be made to this. One simple one is to only use odd numbers in the array, which would allow us to reduce our space usage by half.

Probability

Probability can be a complex topic, but it's based in a few basic laws that can be logically derived.

Let's look at a Venn diagram to visualize two events A and B. The areas of the two circles represent their relative probability, and the overlapping area is the event {A and B}.



Probability of A and B

Imagine you were throwing a dart at this Venn diagram. What is the probability that you would land in the intersection between A and B? If you knew the odds of landing in A, and you also knew the percent of A that's also in B (that is, the odds of being in B given that you were in A), then you could express the probability as:

```
P(A and B) = P(B given A) P(A)
```

For example, imagine we were picking a number between 1 and 10 (inclusive). What's the probability of picking an even number *and* a number between 1 and 5? The odds of picking a number between 1 and 5 is 50%, and the odds of a number between 1 and 5 being even is 40%. So, the odds of doing both are:

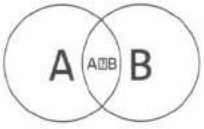
```
P(x is even and x <= 5)
    = P(x is even given x <= 5) P(x <= 5)
    = (2/5) * (1/2)
    = 1/5
```

Probability of A or B

Now, imagine you wanted to know what the probability of landing in A or B is. If you knew the odds of landing in each individually, and you also knew the odds of landing in their intersection, then you could express the probability as:

```
P(A or B) = P(A) + P(B) - P(A and B)
```

Logically, this makes sense. If we simply added their sizes, we would have double-counted their intersection. We need to subtract this out. We can again visualize this through a Venn diagram:



For example, imagine we were picking a number between 1 and 10 (inclusive). What's the probability of picking an even number *or* a number between 1 and 5? We have a 50% probability of picking an even number and a 50% probability of picking a number

between 1 and 5. The odds of doing both are 20%. So the odds are:

```
P(x is even or x <=5)
    = P(x is even) + P(x <= 5) - P(x is even and x <= 5)
    = (1/2) + (1/2) - (1/5)
    = 4/5
```

From here, getting the special case rules for independent events and for mutually exclusive events is easy.

Independence

If A and B are independent (that is, one happening tells you nothing about the other happening), then `P(A and B) = P(A) P(B)`. This rule simply comes from recognizing that `P(B given A) = P(B)`, since A indicates nothing about B.

Mutual Exclusivity

If A and B are mutually exclusive (that is, if one happens, then the other cannot happen), then `P(A or B) = P(A) + P(B)`. This is because `P(A and B) = 0`, so this term is removed from the earlier `P(A or B)` equation.

Many people, strangely, mix up the concepts of independence and mutual exclusivity. They are *entirely* different. In fact, two events cannot be both independent and mutually exclusive (provided both have probabilities greater than 0). Why? Because mutual exclusivity means that if one happens then the other cannot. Independence, however, says that one event happening means absolutely *nothing* about the other event. Thus, as long as two events have non-zero probabilities, they will never be both mutually exclusive and independent.

If one or both events have a probability of zero (that is, it is impossible), then the events are both independent and mutually exclusive. This is provable through a simple application of the definitions (that is, the formulas) of independence and mutual exclusivity.

Things to Watch Out For

1. Be careful with the difference in precision between floats and doubles.
2. Don't assume that a value (such as the slope of a line) is an `int` unless you've been told so.
3. Unless otherwise specified, do not assume that events are independent (or mutually exclusive). You should be careful, therefore, of blindly multiplying or adding probabilities.

Interview Questions

7.1 You have a basketball hoop and someone says that you can play one of two games.

Game 1: You get one shot to make the hoop.

Game 2: You get three shots and you have to make two of three shots.

If p is the probability of making a particular shot, for which values of p should you pick one game or the other?

pg 264

7.2 There are three ants on different vertices of a triangle. What is the probability of collision (between any two or all of them) if they start walking on the sides of the triangle? Assume that each ant randomly picks a direction, with either direction being equally likely to be chosen, and that they walk at the same speed.

Similarly, find the probability of collision with n ants on an n-vertex polygon.

pg 265

7.3 Given two lines on a Cartesian plane, determine whether the two lines would intersect.

pg 266

7.4 Write methods to implement the multiply, subtract, and divide operations for integers. Use only the add operator.

pg 267

7.5 Given two squares on a two-dimensional plane, find a line that would cut these two squares in half. Assume that the top and the bottom sides of the square run parallel to the x-axis.

pg 269

7.6 Given a two-dimensional graph with points on it, find a line which passes the most number of points.

pg 271

7.7 Design an algorithm to find the kth number such that the only prime factors are 3, 5, and 7.

pg 274

Additional Questions: Moderate (#17.11), Hard (#18.2)

8

Object-Oriented Design

Object-oriented design questions require a candidate to sketch out the classes and methods to implement technical problems or real-life objects. These problems give—or at least are believed to give—an interviewer insight into your coding style.

These questions are not so much about regurgitating design patterns as they are about demonstrating that you understand how to create elegant, maintainable object-oriented code. Poor performance on this type of question may raise serious red flags.

How to Approach Object-Oriented Design Questions

Regardless of whether the object is a physical item or a technical task, object-oriented design questions can be tackled in similar ways. The following approach will work well for many problems.

Step 1: Handle Ambiguity

Object-oriented design (OOD) questions are often intentionally vague in order to test whether you'll make assumptions or if you'll ask clarifying questions. After all, a developer who just codes something without understanding what she is expected to create wastes the company's time and money, and may create much more serious issues.

When being asked an object-oriented design question, you should inquire *who* is going to use it and *how* they are going to use it. Depending on the question, you may even want to go through the "six Ws": who, what, where, when, how, why.

For example, suppose you were asked to describe the object-oriented design for a coffee maker. This seems straightforward enough, right? Not quite.

Your coffee maker might be an industrial machine designed to be used in a massive restaurant servicing hundreds of customers per hour and making ten different kinds of coffee products. Or it might be a very simple machine, designed to be used by the elderly for just simple black coffee. These use cases will significantly impact your design.

Step 2: Define the Core Objects

Now that we understand what we're designing, we should consider what the "core objects" in a system are. For example, suppose we are asked to do the object-oriented design for a restaurant. Our core objects might be things like `Table`, `Guest`, `Party`, `Order`, `Meal`, `Employee`, `Server`, and `Host`.

Step 3: Analyze Relationships

Having more or less decided on our core objects, we now want to analyze the relationships between the objects. Which objects are members of which other objects? Do any objects inherit from any others? Are relationships many-to-many or one-to-many?

For example, in the restaurant question, we may come up with the following design:

- `Party` should have an array of `Guests`.
- `Server` and `Host` inherit from `Employee`.
- Each `Table` has one `Party`, but each `Party` may have multiple `Tables`.
- There is one `Host` for the `Restaurant`.

Be very careful here—you can often make incorrect assumptions. For example, a single `Table` may have multiple `Parties` (as is common in the trendy "communal tables" at some restaurants). You should talk to your interviewer about how general purpose your design should be.

Step 4: Investigate Actions

At this point, you should have the basic outline of your object-oriented design. What remains is to consider the key actions that the objects will take and how they relate to each other. You may find that you have forgotten some objects, and you will need to update your design.

For example, a `Party` walks into the `Restaurant`, and a `Guest` requests a `Table` from the `Host`. The `Host` looks up the `Reservation` and, if it exists, assigns the `Party` to a `Table`. Otherwise, the `Party` is added to the end of the list. When a `Party` leaves, the `Table` is freed and assigned to a new `Party` in the list.

Design Patterns

Because interviewers are trying to test your capabilities and not your knowledge, design patterns are mostly beyond the scope of an interview. However, the Singleton and Factory Method design patterns are especially useful for interviews, so we will cover them here.

There are far more design patterns than this book could possibly discuss. A fantastic way to improve your software engineering skills is to pick up a book that focuses on this area specifically.

Singleton Class

The Singleton pattern ensures that a class has only one instance and ensures access to the instance through the application. It can be useful in cases where you have a "global" object with exactly one instance. For example, we may want to implement `Restaurant` such that it has exactly one instance of `Restaurant`.

```
1  public class Restaurant {
2     private static Restaurant _instance = null;
3     public static Restaurant getInstance() {
4        if (_instance == null) {
5           _instance = new Restaurant();
6        }
7        return _instance;
8     }
9  }
```

Factory Method

The Factory Method offers an interface for creating an instance of a class, with its subclasses deciding which class to instantiate. You might want to implement this with the creator class being abstract and not providing an implementation for the Factory method. Or, you could have the Creator class be a concrete class that provides an implementation for the Factory method. In this case, the Factory method would take a parameter representing which class to instantiate.

```
1  public class CardGame {
2     public static CardGame createCardGame(GameType type) {
3        if (type == GameType.Poker) {
4           return new PokerGame();
5        } else if (type == GameType.BlackJack) {
6           return new BlackJackGame();
7        }
8        return null;
9     }
10 }
```

Interview Questions

8.1 Design the data structures for a generic deck of cards. Explain how you would subclass the data structures to implement blackjack.

pg 280

8.2 Imagine you have a call center with three levels of employees: respondent, manager, and director. An incoming telephone call must be first allocated to a respondent who is free. If the respondent can't handle the call, he or she must escalate the call to a manager. If the manager is not free or not able to handle it, then the call should be escalated to a director. Design the classes and data structures for this problem. Implement a method `dispatchCall()` which assigns a call to the first available employee.

pg 283

8.3 Design a musical jukebox using object-oriented principles.

pg 286

8.4 Design a parking lot using object-oriented principles.

pg 289

8.5 Design the data structures for an online book reader system.

pg 292

8.6 Implement a jigsaw puzzle. Design the data structures and explain an algorithm to solve the puzzle. You can assume that you have a `fitsWith` method which, when passed two puzzle pieces, returns true if the two pieces belong together.

pg 296

8.7 Explain how you would design a chat server. In particular, provide details about the various backend components, classes, and methods. What would be the hardest problems to solve?

pg 300

8.8 Othello is played as follows: Each Othello piece is white on one side and black on the other. When a piece is surrounded by its opponents on both the left and right sides, or both the top and bottom, it is said to be captured and its color is flipped. On your turn, you must capture at least one of your opponent's pieces. The game ends when either user has no more valid moves. The win is assigned to the person with the most pieces. Implement the object-oriented design for Othello.

pg 305

8.9 Explain the data structures and algorithms that you would use to design an in-memory file system. Illustrate with an example in code where possible.

pg 308

8.10 Design and implement a hash table which uses chaining (linked lists) to handle collisions.

pg 311

Additional Questions: Threads and Locks (#16.3)

9

Recursion and Dynamic Programming

While there is a wide variety of recursive problems, many follow similar patterns. A good hint that a problem is recursive is that it can be built off of sub-problems.

When you hear a problem beginning with the following statements, it's often (though not always) a good candidate for recursion: "Design an algorithm to compute the nth ..."; "Write code to list the first n..."; "Implement a method to compute all..."; etc..

Practice makes perfect! The more problems you do, the easier it will be to recognize recursive problems.

How to Approach

Recursive solutions, by definition, are built off solutions to sub-problems. Many times, this will mean simply to compute `f(n)` by adding something, removing something, or otherwise changing the solution for `f(n-1)`. In other cases, you might have to do something more complicated.

You should consider both bottom-up and top-down recursive solutions. The Base Case and Build approach works quite well for recursive problems.

Bottom-Up Recursion

Bottom-up recursion is often the most intuitive. We start with knowing how to solve the problem for a simple case, like a list with only one element, and figure out how to solve the problem for two elements, then for three elements, and so on. The key here is to think about how you can *build* the solution for one case off of the previous case.

Top-Down Recursion

Top-Down Recursion can be more complex, but it's sometimes necessary for problems. In these problems, we think about how we can divide the problem for case N into sub-problems. Be careful of overlap between the cases.

Dynamic Programming

Dynamic programming (DP) problems are rarely asked because, quite simply, they're too difficult for a 45-minute interview. Even good candidates would generally do so poorly on these problems that it's not a good evaluation technique.

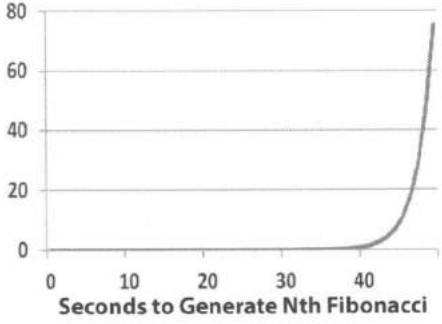
If you're unlucky enough to get a DP problem, you can approach it much the same way as a recursion problem. The difference is that intermediate results are "cached" for future calls.

Simple Example of Dynamic Programming: Fibonacci Numbers

As a very simple example of dynamic programming, imagine you're asked to implement a program to generate the nth Fibonacci number. Sounds simple, right?

```
1  int fibonacci(int i) {
2     if (i == 0) return 0;
3     if (i == 1) return 1;
4     return fibonacci(i - 1) + fibonacci(i - 2);
5  }
```

What is the runtime of this function? Computing the nth Fibonacci number depends on the previous `n-1` numbers. But *each* call does two recursive calls. This means that the runtime is $O(2^n)$. The below graph shows this exponential increase, as computed on a standard desktop computer.



With just a small modification, we can tweak this function to run in $O(N)$ time. We simply "cache" the results of `fibonacci(i)` between calls.

```
1  int[] fib = new int[max];
2  int fibonacci(int i) {
3     if (i == 0) return 0;
4     if (i == 1) return 1;
5     if (fib[i] != 0) return fib[i]; // Return cached result.
6     fib[i] = fibonacci(i - 1) + fibonacci(i - 2); // Cache result
7     return fib[i];
8  }
```

While the first recursive one may take over a minute to generate the 50th Fibonacci number on a standard computer, the dynamic programming method can generate the

10,000th Fibonacci number in just fractions of a millisecond. (Of course, with this exact code, the `int` would have overflowed very early on.)

Dynamic programming, as you can see, is nothing to be scared of. It's little more than recursion where you cache the results. A good way to approach such a problem is often to implement it as a normal recursive solution, and then to add the caching part.

Recursive vs. Iterative Solutions

Recursive algorithms can be very space inefficient. Each recursive call adds a new layer to the stack, which means that if your algorithm has `O(n)` recursive calls, then it uses `O(n)` memory. Ouch!

All recursive code can be implemented iteratively, although sometimes the code to do so is much more complex. Before diving into recursive code, ask yourself how hard it would be to implement it iteratively, and discuss the trade-offs with your interviewer.

Interview Questions

9.1 A child is running up a staircase with n steps, and can hop either 1 step, 2 steps, or 3 steps at a time. Implement a method to count how many possible ways the child can run up the stairs.

pg 316

9.2 Imagine a robot sitting on the upper left corner of an X by Y grid. The robot can only move in two directions: right and down. How many possible paths are there for the robot to go from `(0,0)` to `(X,Y)`?

FOLLOW UP

Imagine certain spots are "off limits," such that the robot cannot step on them. Design an algorithm to find a path for the robot from the top left to the bottom right.

pg 317

9.3 A magic index in an array `A[0...n-1]` is defined to be an index such that `A[i] = i`. Given a sorted array of distinct integers, write a method to find a magic index, if one exists, in array A.

FOLLOW UP

What if the values are not distinct?

pg 319

9.4 Write a method to return all subsets of a set.

pg 321

9.5 Write a method to compute all permutations of a string.

pg 324

9.6 Implement an algorithm to print all valid (e.g., properly opened and closed) combinations of n-pairs of parentheses.

EXAMPLE

Input: 3

Output: `((()))`, `(()())`, `(())()`, `()(())`, `()()()`

pg 325

9.7 Implement the "paint fill" function that one might see on many image editing programs. That is, given a screen (represented by a two-dimensional array of colors), a point, and a new color, fill in the surrounding area until the color changes from the original color.

pg 327

9.8 Given an infinite number of quarters (25 cents), dimes (10 cents), nickels (5 cents) and pennies (1 cent), write code to calculate the number of ways of representing n cents.

pg 328

9.9 Write an algorithm to print all ways of arranging eight queens on an 8x8 chess board so that none of them share the same row, column or diagonal. In this case, "diagonal" means all diagonals, not just the two that bisect the board.

pg 331

9.10 You have a stack of n boxes, with widths w_i, heights h_i, and depths d_i. The boxes cannot be rotated and can only be stacked on top of one another if each box in the stack is strictly larger than the box above it in width, height, and depth. Implement a method to build the tallest stack possible, where the height of a stack is the sum of the heights of each box.

pg 333

9.11 Given a boolean expression consisting of the symbols `0`, `1`, `&`, `|`, and `^`, and a desired boolean result value `result`, implement a function to count the number of ways of parenthesizing the expression such that it evaluates to `result`.

EXAMPLE

Expression: `1^0|0|1`

Desired result: `false (0)`

Output: 2 ways. `1^((0|0)|1)` and `1^(0|(0|1))`.

pg 335

Additional Questions: Linked Lists (#2.2, #2.5, #2.7), Stacks and Queues (#3.3), Trees and Graphs (#4.1, #4.3, #4.4, #4.5, #4.7, #4.8, #4.9), Bit Manipulation (#5.7), Brain Teasers (#6.4), Sorting and Searching (#11.5, #11.6, #11.7, #11.8), C++ (#13.7), Moderate (#17.13, #17.14), Hard (#18.4, #18.7, #18.12, #18.13)

10

Scalability and Memory Limits

Despite how intimidating they seem, scalability questions can be among the easiest questions. There are no "gotchas," no tricks, and no fancy algorithms—at least not usually. You don't need any courses in distributed systems, nor do you need any experience in system design. With a bit of practice, any thorough and intelligent software engineer can handle these questions with ease.

The Step-By-Step Approach

Interviewers are not trying to test your knowledge of system design; in fact, interviewers rarely try to test knowledge of anything but the most basic Computer Science concepts. Instead, they are evaluating your ability to break down a tricky problem and to solve problems using what you do know. The following approach works well for many system design problems.

Step 1: Make Believe

Pretend that the data can all fit on one machine and there are no memory limitations. How would you solve the problem? This answer to this question will provide the general outline for your solution.

Step 2: Get Real

Now, go back to the original problem. How much data can you fit on one machine, and what problems will occur when you split the data up? Common problems include figuring out how to logically divide the data up, and how one machine would identify where to look up a different piece of data.

Step 3: Solve Problems

Finally, think about how to solve the issues you identified in Step 2. Remember that the solution for each issue might be to actually remove the issue entirely, or it might be to simply mitigate the issue. Usually, you can continue to use (with modifications) the approach you outlined in Step 1, but occasionally you will need to fundamentally alter

the approach.

Note that an iterative approach is typically useful. That is, once you have solved the problems from Step 2, new problems may have emerged, and you must tackle those as well.

Your goal is not to re-architect a complex system that companies have spent millions of dollars building, but rather, to demonstrate that you can analyze and solve problems. Poking holes in your own solution is a fantastic way to demonstrate this.

What You Need to Know: Information, Strategies and Issues

A Typical System

Though super-computers are still in use, most web-based companies use large systems of interconnected machines. You can almost always assume that you will be working in such a system.

Prior to your interview, you should fill in the following chart. This chart will help you to ballpark how much data a computer can store.

Component	Typical Capacity / Value
Hard Drive Space	
Memory	
Internet Transfer Latency	

Dividing Up Lots of Data

Though we can sometimes increase hard drive space in a computer, there comes a point where data simply must be divided up across machines. The question, then, is what data belongs on which machine? There are a few strategies for this.

- *By Order of Appearance:*

  We could simply divide up data by order of appearance. That is, as new data comes in, we wait for our current machine to fill up before adding an additional machine. This has the advantage of never using more machines than are necessary. However, depending on the problem and our data set, our lookup table may be more complex and potentially very large.

- *By Hash Value:*

  An alternative approach is to store the data on the machine corresponding to the hash value of the data. More specifically, we do the following: (1) pick some sort of *key* relating to the data, (2) hash the key, (3) mod the hash value by the number of machines, and (4) store the data on the machine with that value. That is, the data is stored on machine `#[mod(hash(key), N)]`.

  The nice thing about this is that there's no need for a lookup table. Every machine

will automatically know where a piece of data is. The problem, however, is that a machine may get more data and eventually exceed its capacity. In this case, we may need to either shift data around the other machines for better load balancing (which is very expensive), or split this machine's data into two machines (causing a tree-like structure of machines).

- *By Actual Value:*

  Dividing up data by hash value is essentially arbitrary; there is no relationship between what the data represents and which machine stores the data. In some cases, we may be able to reduce system latency by using information about what the data represents.

  For example, suppose you're designing a social network. While people do have friends around the world, the reality is that someone living in Mexico will probably have a lot more friends from Mexico than an average Russian citizen. We could, perhaps, store "similar" data on the same machine so that looking up the Mexican person's friends requires fewer machine hops.

- *Arbitrarily:*

  Frequently, data just gets arbitrarily broken up and we implement a lookup table to identify which machine holds a piece of data. While this does necessitate a potentially large lookup table, it simplifies some aspects of system design and can enable us to do better load balancing.

Example: Find all documents that contain a list of words

Given a list of millions of documents, how would you find all documents that contain a list of words? The words do not need to appear in any particular order, but they must be complete words. That is, "book" does not match "bookkeeper."

Before we start solving the problem, we need to understand whether this is a one time only operation, or if this `findWords` procedure will be called repeatedly. Let's assume that we will be calling `findWords` many times for the same set of documents, and we can therefore accept the burden of pre-processing.

Step 1

The first step is to forget about the millions of documents and pretend we just had a few dozen documents. How would we implement `findWords` in this case? (Tip: stop here and try to solve this yourself, before reading on.)

One way to do this is to pre-process each document and create a hash table index. This hash table would map from a word to a list of the documents that contain that word.

```
"books" -> {doc2, doc3, doc6, doc8}
"many"  -> {doc1, doc3, doc7, doc8, doc9}
```

To search for "many books," we would simply do an intersection on the values for

"books" and "many", and return {doc3, doc8} as the result.

Step 2

Now, go back to the original problem. What problems are introduced with millions of documents? For starters, we probably need to divide up the documents across many machines. Also, depending on a variety of factors, such as the number of possible words and the repetition of words in a document, we may not be able to fit the full hash table on one machine. Let's assume that this is the case.

This division introduces the following key concerns:

1. How will we divide up our hash table? We could divide it up by keyword, such that a given machine contains the full document list for a given word. Or, we could divide by document, such that a machine contains the keyword mapping for only a subset of the documents.
2. Once we decide how to divide up the data, we may need to process a document on one machine and push the results off to other machines. What does this process look like? (Note: if we divide the hash table by document, this step may not be necessary.)
3. We will need a way to knowing which machine holds a piece of data. What does this lookup table look like, and where is it stored?

These are just three concerns. There may be many others.

Step 3

In step 3, we find solutions to each of these issues. One solution is to divide up the words alphabetically by keyword, such that each machine controls a range of words (e.g., "after" through "apple").

We can implement a simple algorithm in which we iterate through the keywords alphabetically, storing as much data as possible on one machine. When that machine is full, we will move to the next machine.

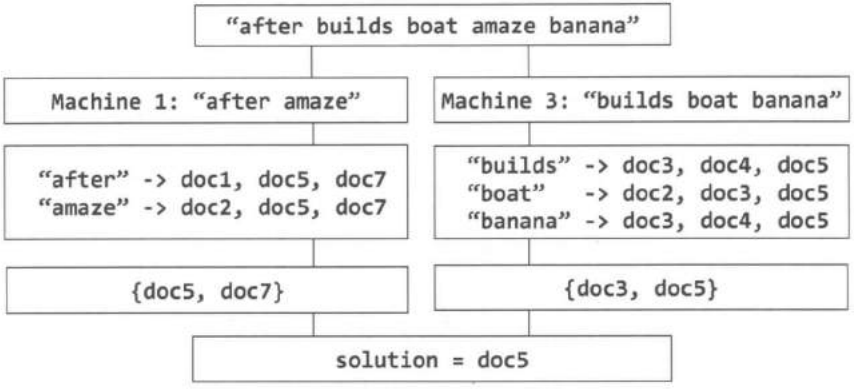
The advantage of this approach is that the lookup table is small and simple (since it must only specify a range of values), and each machine can store a copy of the lookup table. The disadvantage, however, is that if new documents or words are added, we may need to perform an expensive shift of keywords.

To find all the documents that match a list of strings, we would first sort the list and then send each machine a lookup request for the strings that the machine owns. For example, if our string is "after builds boat amaze banana", machine 1 would get a lookup request for {"after", "amaze"}.

Machine 1 would look up the documents containing "after" and "amaze," and perform an intersection on these document lists. Machine 3 would do the same for {"banana", "boat", "builds"}, and intersect their lists.

In the final step, the initial machine would do an intersection on the results from Machine 1 and Machine 3.

The following diagram explains this process.



Interview Questions

10.1 Imagine you are building some sort of service that will be called by up to 1000 client applications to get simple end-of-day stock price information (open, close, high, low). You may assume that you already have the data, and you can store it in any format you wish. How would you design the client-facing service which provides the information to client applications? You are responsible for the development, rollout, and ongoing monitoring and maintenance of the feed. Describe the different methods you considered and why you would recommend your approach. Your service can use any technologies you wish, and can distribute the information to the client applications in any mechanism you choose.

pg 342

10.2 How would you design the data structures for a very large social network like Facebook or LinkedIn? Describe how you would design an algorithm to show the connection, or path, between two people (e.g., Me -> Bob -> Susan -> Jason -> You).

pg 344

10.3 Given an input file with four billion non-negative integers, provide an algorithm to generate an integer which is not contained in the file. Assume you have 1 GB of memory available for this task.

FOLLOW UP

What if you have only 10 MB of memory? Assume that all the values are distinct

and we now have no more than one billion non-negative integers.

pg 347

10.4 You have an array with all the numbers from 1 to N, where N is at most 32,000. The array may have duplicate entries and you do not know what N is. With only 4 kilobytes of memory available, how would you print all duplicate elements in the array?

pg 350

10.5 If you were designing a web crawler, how would you avoid getting into infinite loops?

pg 351

10.6 You have 10 billion URLs. How do you detect the duplicate documents? In this case, assume that "duplicate" means that the URLs are identical.

pg 353

10.7 Imagine a web server for a simplified search engine. This system has 100 machines to respond to search queries, which may then call out using `processSearch(string query)` to another cluster of machines to actually get the result. The machine which responds to a given query is chosen at random, so you can not guarantee that the same machine will always respond to the same request. The method `processSearch` is very expensive. Design a caching mechanism for the most recent queries. Be sure to explain how you would update the cache when data changes.

pg 354

Additional Questions: Object-Oriented Design (#8.7)

11

Sorting and Searching

Understanding the common sorting and searching algorithms is incredibly valuable, as many of sorting and searching problems are tweaks of the well-known algorithms. A good approach is therefore to run through the different sorting algorithms and see if one applies particularly well.

For example, suppose you are asked the following question: Given a very large array of `Person` objects, sort the people in increasing order of age.

We're given two interesting bits of knowledge here:

1. It's a large array, so efficiency is very important.
2. We are sorting based on ages, so we know the values are in a small range.

By scanning through the various sorting algorithms, we might notice that bucket sort (or radix sort) would be a perfect candidate for this algorithm. In fact, we can make the buckets small (just 1 year each) and get $O(n)$ running time.

Common Sorting Algorithms

Learning (or re-learning) the common sorting algorithms is a great way to boost your performance. Of the five algorithms explained below, Merge Sort, Quick Sort and Bucket Sort are the most commonly used in interviews.

Bubble Sort | Runtime: $O(n^2)$ average and worst case. Memory: $O(1)$.

In bubble sort, we start at the beginning of the array and swap the first two elements if the first is greater than the second. Then, we go to the next pair, and so on, continuously making sweeps of the array until it is sorted.

Selection Sort | Runtime: $O(n^2)$ average and worst case. Memory: $O(1)$.

Selection sort is the child's algorithm: simple, but inefficient. Find the smallest element using a linear scan and move it to the front (swapping it with the front element). Then, find the second smallest and move it, again doing a linear scan. Continue doing this

until all the elements are in place.

Merge Sort | Runtime: O(n Log(n)) average and worst case. Memory: Depends.

Merge sort divides the array in half, sorts each of those halves, and then merges them back together. Each of those halves has the same sorting algorithm applied to it. Eventually, you are merging just two single-element arrays. It is the "merge" part that does all the heavy lifting.

The merge method operates by copying all the elements from the target array segment into a helper array, keeping track of where the start of the left and right halves should be (`helperLeft` and `helperRight`). We then iterate through `helper`, copying the smaller element from each half into the array. At the end, we copy any remaining elements into the target array.

```
1  void mergesort(int[] array) {
2     int[] helper = new int[array.length];
3     mergesort(array, helper, 0, array.length - 1);
4  }
5
6  void mergesort(int[] array, int[] helper, int low, int high) {
7     if (low < high) {
8        int middle = (low + high) / 2;
9        mergesort(array, helper, low, middle); // Sort left half
10       mergesort(array, helper, middle+1, high); // Sort right half
11       merge(array, helper, low, middle, high); // Merge them
12    }
13 }
14
15 void merge(int[] array, int[] helper, int low, int middle,
16            int high) {
17    /* Copy both halves into a helper array */
18    for (int i = low; i <= high; i++) {
19       helper[i] = array[i];
20    }
21
22    int helperLeft = low;
23    int helperRight = middle + 1;
24    int current = low;
25
26    /* Iterate through helper array. Compare the left and right
27     * half, copying back the smaller element from the two halves
28     * into the original array. */
29    while (helperLeft <= middle && helperRight <= high) {
30       if (helper[helperLeft] <= helper[helperRight]) {
31          array[current] = helper[helperLeft];
32          helperLeft++;
33       } else { // If right element is smaller than left element
34          array[current] = helper[helperRight];
35          helperRight++;
```

```
36        }
37        current++;
38     }
39
40     /* Copy the rest of the left side of the array into the
41      * target array */
42     int remaining = middle - helperLeft;
43     for (int i = 0; i <= remaining; i++) {
44        array[current + i] = helper[helperLeft + i];
45     }
46  }
```

You may notice that only the remaining elements from the left half of the helper array are copied into the target array. Why not the right half? The right half doesn't need to be copied because it's *already* there.

Consider, for example, an array like `[1, 4, 5 || 2, 8, 9]` (the "||" indicates the partition point). Prior to merging the two halves, both the helper array and the target array segment will end with `[8, 9]`. Once we copy over four elements (`1, 4, 5,` and 2) into the target array, the `[8, 9]` will still be in place in both arrays. There's no need to copy them over.

Quick Sort | Runtime: O(n Log(n)) average, O(n²) worst case. Memory: O(Log(n)).

In quick sort, we pick a random element and partition the array, such that all numbers that are less than the partitioning element come before all elements that are greater than it. The partitioning can be performed efficiently through a series of swaps (see below).

If we repeatedly partition the array (and its sub-arrays) around an element, the array will eventually become sorted. However, as the partitioned element is not guaranteed to be the median (or anywhere near the median), our sorting could be very slow. This is the reason for the $O(n^2)$ worst case runtime.

```
1  void quickSort(int arr[], int left, int right) {
2     int index = partition(arr, left, right);
3     if (left < index - 1) { // Sort left half
4        quickSort(arr, left, index - 1);
5     }
6     if (index < right) { // Sort right half
7        quickSort(arr, index, right);
8     }
9  }
10
11 int partition(int arr[], int left, int right) {
12    int pivot = arr[(left + right) / 2]; // Pick pivot point
13    while (left <= right) {
14       // Find element on left that should be on right
15       while (arr[left] < pivot) left++;
16
```

```
17        // Find element on right that should be on left
18        while (arr[right] > pivot) right--;
19
20        // Swap elements, and move left and right indices
21        if (left <= right) {
22           swap(arr, left, right); // swaps elements
23           left++;
24           right--;
25        }
26     }
27     return left;
28  }
29
```

Radix Sort | Runtime: O(kn) (see below)

Radix sort is a sorting algorithm for integers (and some other data types) that takes advantage of the fact that integers have a finite number of bits. In radix sort, we iterate through each digit of the number, grouping numbers by each digit. For example, if we have an array of integers, we might first sort by the first digit, so that the 0s are grouped together. Then, we sort each of these groupings by the next digit. We repeat this process sorting by each subsequent digit, until finally the whole array is sorted.

Unlike comparison sorting algorithms, which cannot perform better than O(n log(n)) in the average case, radix sort has a runtime of O(kn), where n is the number of elements and k is the number of passes of the sorting algorithm.

Searching Algorithms

When we think of searching algorithms, we generally think of binary search. Indeed, this is a very useful algorithm to study.

In binary search, we look for an element x in a sorted array by first comparing x to the midpoint of the array. If x is less than the midpoint, then we search the left half of the array. If x is greater than the midpoint, then we search the right half of the array. We then repeat this process, treating the left and right halves as subarrays. Again, we compare x to the midpoint of this subarray and then search either its left or right side. We repeat this process until we either find x or the subarray has size 0.

Note that although the concept is fairly simple, getting all the details right is far more difficult than you might think. As you study the code below, pay attention to the plus ones and minus ones.

```
1  int binarySearch(int[] a, int x) {
2     int low = 0;
3     int high = a.length - 1;
4     int mid;
5
6     while (low <= high) {
```

```
7         mid = (low + high) / 2;
8         if (a[mid] < x) {
9            low = mid + 1;
10        } else if (a[mid] > x) {
11           high = mid - 1;
12        } else {
13           return mid;
14        }
15     }
16     return -1; // Error
17  }
18
19  int binarySearchRecursive(int[] a, int x, int low, int high) {
20     if (low > high) return -1; // Error
21
22     int mid = (low + high) / 2;
23     if (a[mid] < x) {
24        return binarySearchRecursive(a, x, mid + 1, high);
25     } else if (a[mid] > x) {
26        return binarySearchRecursive(a, x, low, mid - 1);
27     } else {
28        return mid;
29     }
30  }
```

Potential ways to search a data structure extend beyond binary search, and you would do best not to limit yourself to just this option. You might, for example, search for a node by leveraging a binary tree, or by using a hash table. Think beyond binary search!

Interview Questions

11.1 You are given two sorted arrays, A and B, where A has a large enough buffer at the end to hold B. Write a method to merge B into A in sorted order.

pg 360

11.2 Write a method to sort an array of strings so that all the anagrams are next to each other.

pg 361

11.3 Given a sorted array of n integers that has been rotated an unknown number of times, write code to find an element in the array. You may assume that the array was originally sorted in increasing order.

EXAMPLE

Input: find 5 in {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}

Output: 8 (the index of 5 in the array)

pg 362

11.4 Imagine you have a 20 GB file with one string per line. Explain how you would sort the file.

pg 364

11.5 Given a sorted array of strings which is interspersed with empty strings, write a method to find the location of a given string.

EXAMPLE

Input: find "ball" in {"at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""}

Output: 4

pg 364

11.6 Given an M x N matrix in which each row and each column is sorted in ascending order, write a method to find an element.

pg 365

11.7 A circus is designing a tower routine consisting of people standing atop one another's shoulders. For practical and aesthetic reasons, each person must be both shorter and lighter than the person below him or her. Given the heights and weights of each person in the circus, write a method to compute the largest possible number of people in such a tower.

EXAMPLE:

Input (ht, wt): (65, 100) (70, 150) (56, 90) (75, 190) (60, 95) (68, 110)

Output: The longest tower is length 6 and includes from top to bottom:

(56, 90) (60,95) (65,100) (68,110) (70,150) (75,190)

pg 371

11.8 Imagine you are reading in a stream of integers. Periodically, you wish to be able to look up the rank of a number x (the number of values less than or equal to x). Implement the data structures and algorithms to support these operations. That is, implement the method `track(int x)`, which is called when each number is generated, and the method `getRankOfNumber(int x)`, which returns the number of values less than or equal to x (not including x itself).

EXAMPLE

Stream (in order of appearance): 5, 1, 4, 4, 5, 9, 7, 13, 3

```
getRankOfNumber(1) = 0
getRankOfNumber(3) = 1
getRankOfNumber(4) = 3
```

pg 374

Additional Questions: Arrays and Strings (#1.3), Recursion (#9.3), Moderate (#17.6, #17.12), Hard (#18.5)

12

Testing

Before you flip past this chapter saying, "but I'm not a tester," stop and think. Testing is an important task for a software engineer, and for this reason, testing questions may come up during your interview. Of course, if you are applying for Testing roles (or Software Engineer in Test), then that's all the more reason why you need to pay attention.

Testing problems usually fall under one of four categories: (1) Test a real world object (like a pen); (2) Test a piece of software; (3) Write test code for a function; (4) Troubleshoot an existing issue. We'll cover approaches for each of these four types.

Remember that all four types require you to not make an assumption that the input or the user will play nice. Expect abuse and plan for it.

What the Interviewer Is Looking For

At their surface, testing questions seem like they're just about coming up with an extensive list of test cases. And to some extent, that's right. You do need to come up with a reasonable list of test cases.

But in addition, interviewers want to test the following:

- *Big Picture Understanding:* Are you a person who understands what the software is really about? Can you prioritize test cases properly? For example, suppose you're asked to test an e-commerce system like Amazon. It's great to make sure that the product images appear in the right place, but it's even more important that payments work reliably, products are added to the shipment queue, and customers are never double charged.
- *Knowing How the Pieces Fit Together:* Do you understand how software works, and how it might fit into a greater ecosystem? Suppose you're asked to test Google Spreadsheets. It's important that you test opening, saving, and editing documents. But, Google Spreadsheets is part of a larger ecosystem. You need to test integration with Gmail, with plug-ins, and with other components.

- *Organization:* Do you approach the problem in a structured manner, or do you just spout off anything that comes to your head? Some candidates, when asked to come up with test cases for a camera, will just state anything and everything that comes to their head. A good candidate will break down the parts into categories like Taking Photos, Image Management, Settings, and so on. This structured approach will also help you to do a more thorough job creating the test cases.
- *Practicality:* Can you actually create reasonable testing plans? For example, if a user reports that the software crashes when they open a specific image, and you just tell them to reinstall the software, that's typically not very practical. Your testing plans need to be feasible and realistic for a company to implement.

Demonstrating these aspects will show that you will be a valuable member of the testing team.

Testing a Real World Object

Some candidates are surprised to be asked questions like how to test a pen. After all, you should be testing software, right? Maybe, but these "real world" questions are still very common. Let's walk through this with an example.

Question: How would you test a paperclip?

Step 1: Who will use it? And why?

You need to discuss with your interviewer who is using the product and for what purpose. The answer may not be what you think. The answer could be "by teachers, to hold papers together," or it could be "by artists, to bend into the shape of animal." Or, it could be both. The answer to this question will shape how you handle the remaining questions.

Step 2: What are the use cases?

It will be useful for you to make a list of the use cases. In this case, the use case might be simply fastening paper together in a non-damaging (to the paper) way.

For other questions, there might be multiple use cases. It might be, for example, that the product needs to be able to send and receive content, or write and erase, and so on.

Step 3: What are the bounds of use?

The bounds of use might mean holding up to thirty sheets of paper in a single usage without permanent damage (e.g., bending), and thirty to fifty sheets with minimal permanent bending.

The bounds also extend to environmental factors as well. For example, should the paperclip work during very warm temperatures (90 - 110 degrees Fahrenheit)? What about extreme cold?

Step 4: What are the stress / failure conditions?

No product is fail-proof, so analyzing failure conditions needs to be part of your testing. A good discussion to have with your interviewer is about when it's acceptable (or even necessary) for the product to fail, and what failure should mean.

For example, if you were testing a laundry machine, you might decide that the machine should be able to handle at least 30 shirts or pants. Loading 30 - 45 pieces of clothing may result in minor failure, such as the clothing being inadequately cleaned. At more than 45 pieces of clothing, extreme failure might be acceptable. However, extreme failure in this case should probably mean the machine never turning on the water. It should certainly *not* mean a flood or a fire.

Step 5: How would you perform the testing?

In some cases, it might also be relevant to discuss the details of performing the testing. For example, if you need to make sure a chair can withstand normal usage for five years, you probably can't actually place it in a home and wait five years. Instead, you'd need to define what "normal" usage is (How many "sits" per year on the seat? What about the armrest?). Then, in addition to doing some manual testing, you would likely want a machine to automate some of the usage.

Testing a Piece of Software

Testing a piece of software is actually very similar to testing a real world object. The major difference is that software testing generally places a greater emphasis on the details of performing testing.

Note that software testing has two core aspects to it:

- *Manual vs. Automated Testing:* In an ideal world, we might love to automate everything, but that's rarely feasible. Some things are simply much better with manual testing because some features are too qualitative for a computer to effectively examine (such as if content represents pornography). Additionally, whereas a computer can generally recognize only issues that it's been told to look for, human observation may reveal new issues that haven't been specifically examined. Both humans and computers form an essential part of the testing process.
- *Black Box Testing vs. White Box Testing:* This distinction refers to the degree of access we have into the software. In black box testing, we're just given the software as-is and need to test it. With white box testing, we have additional programmatic access to test individual functions. We can also automate some black box testing, although it's certainly much harder.

Let's walk through an approach from start to end.

Step 1: Are we doing Black Box Testing or White Box Testing?

Though this question can often be delayed to a later step, I like to get it out of the way

early on. Check with your interviewer as to whether you're doing black box testing or white box testing—or both.

Step 2: Who will use it? And why?

Software typically has one or more target users, and the features are designed with this in mind. For example, if you're asked to test software for parental controls on a web browser, your target users include both parents (who are implementing the blocking) and children (who are the recipients of blocking). You may also have "guests" (people who should neither be implementing nor receiving blocking).

Step 3: What are the use cases?

In the software blocking scenario, the use cases of the parents include installing the software, updating controls, removing controls, and of course their own personal internet usage. For the children, the use cases include accessing legal content as well as "illegal" content.

Remember that it's not up to you to just magically decide the use cases. This is a conversation to have with your interviewer.

Step 4: What are the bounds of use?

Now that we have the vague use cases defined, we need to figure out what exactly this means. What does it mean for a website to be blocked? Should just the "illegal" page be blocked, or the entire website? Is the application supposed to "learn" what is bad content, or is it based on a white list or black list? If it's supposed to learn what inappropriate content is, what degree of false positives or false negatives is acceptable?

Step 5: What are the stress conditions / failure conditions?

When the software fails—which it inevitably will—what should the failure look like? Clearly, the software failure shouldn't crash the computer. Instead, it's likely that the software should just permit a blocked site, or ban an allowable site. In the latter case, you might want to discuss the possibility of a selective override with a password from the parents.

Step 6: What are the test cases? How would you perform the testing?

Here is where the distinctions between manual and automated testing, and between black box and white box testing, really come into play.

Steps 3 and 4 should have roughly defined the use cases. In step 6, we further define them and discuss how to perform the testing. What exact situations are you testing? Which of these steps can be automated? Which require human intervention?

Remember that while automation allows you to do some very powerful testing, it also has some significant drawbacks. Manual testing should usually be part of your test procedures.

When you go through this list, don't just rattle off every scenario you can think of. It's disorganized, and you're sure to miss major categories. Instead, approach this in a structured manner. Break down your testing into the main components, and go from there. Not only will you give a more complete list of test cases, but you'll also show that you're a structured, methodical person.

Testing a Function

In many ways, testing a function is the easiest type of testing. The conversation is typically briefer and less vague, as the testing is usually limited to validating input and output.

However, don't overlook the value of some conversation with your interviewer. You should discuss any assumptions with your interviewer, particularly with respect to how to handle specific situations.

Suppose you were asked to write code to test `sort(int[] array)`, which sorts an array of integers. You might proceed as follows.

Step 1: Define the test cases

In general, you should think about the following types of test cases:

- *The normal case:* Does it generate the correct output for typical inputs? Remember to think about potential issues here. For example, because sorting often requires some sort of partitioning, it's reasonable to think that the algorithm might fail on arrays with an odd number of elements, since they can't be evenly partitioned. Your test case should list both examples.
- *The extremes:* What happens when you pass in an empty array? Or a very small (one element) array? What if you pass in a very large one?
- *Nulls and "illegal" input:* It is worthwhile to think about how the code should behave when given illegal input. For example, if you're testing a function to generate the nth Fibonacci number, your test cases should probably include the situation where n is negative.
- *Strange input:* A fourth kind of input sometimes comes up: strange input. What happens when you pass in an already sorted array? Or an array that's sorted in reverse order?

Generating these tests does require knowledge of the function you are writing. If you are unclear as to the constraints, you will need to ask your interviewer about this first.

Step 2: Define the expected result

Often, the expected result is obvious: the right output. However, in some cases, you might want to validate additional aspects. For instance, if the `sort` method returns a new sorted copy of the array, you should probably validate that the original array has

not been touched.

Step 3: Write test code

Once you have the test cases and results defined, writing the code to implement the test cases should be fairly straightforward. Your code might look something like:

```
1  void testAddThreeSorted() {
2     MyList list = new MyList();
3     list.addThreeSorted(3, 1, 2); // Adds 3 items in sorted order
4     assertEquals(list.getElement(0), 1);
5     assertEquals(list.getElement(1), 2);
6     assertEquals(list.getElement(2), 3);
7  }
```

Troubleshooting Questions

A final type of question is explaining how you would debug or troubleshoot an existing issue. Many candidates balk at a question like this, giving unrealistic answers like "reinstall the software." You can approach these questions in a structured manner, like anything else.

Let's walk through this problem with an example: You're working on the Google Chrome team when you receive a bug report: Chrome crashes on launch. What would you do?

Reinstalling the browser might solve this user's problem, but it wouldn't help the other users who might be experiencing the same issue. Your goal is to understand what's *really* happening, so that the developers can fix it.

Step 1: Understand the Scenario

The first thing you should do is ask questions to understand as much about the situation as possible.

- How long has the user been experiencing this issue?
- What version of the browser is it? What operating system?
- Does the issue happen consistently, or how often does it happen? When does it happen?
- Is there an error report that launches?

Step 2: Break Down the Problem

Now that you understand the details of the scenario, you want to break down the problem into testable units. In this case, you can imagine the flow of the situation as follows:

1. Go to Windows Start menu.
2. Click on Chrome icon.

3. Browser instance starts.
4. Browser loads settings.
5. Browser issues HTTP request for homepage.
6. Browser gets HTTP response.
7. Browser parses webpage.
8. Browser displays content.

At some point in this process, something fails and it causes the browser to crash. A strong tester would iterate through the elements of this scenario to diagnose the problem.

Step 3: Create Specific, Manageable Tests

Each of the above components should have realistic instructions—things that you can ask the user to do, or things that you can do yourself (such as replicating steps on your own machine). In the real world, you will be dealing with customers, and you can't give them instructions that they can't or won't do.

Interview Questions

12.1 Find the mistake(s) in the following code:

```
1 unsigned int i;
2 for (i = 100; i >= 0; --i)
3   printf("%d\n", i);
```

pg 378

12.2 You are given the source to an application which crashes when it is run. After running it ten times in a debugger, you find it never crashes in the same place. The application is single threaded, and uses only the C standard library. What programming errors could be causing this crash? How would you test each one?

pg 378

12.3 We have the following method used in a chess game: `boolean canMoveTo(int x, int y)`. This method is part of the `Piece` class and returns whether or not the piece can move to position `(x, y)`. Explain how you would test this method.

pg 379

12.4 How would you load test a webpage without using any test tools?

pg 380

12.5 How would you test a pen?

pg 381

12.6 How would you test an ATM in a distributed banking system?

pg 382

Knowledge Based

Interview Questions and Advice

13

C and C++

A good interviewer won't demand that you code in a language you don't profess to know. Hopefully, if you're asked to code in C++, it's listed on your resume. If you don't remember all the APIs, don't worry—most interviewers (though not all) don't care that much. We do recommend, however, studying up on basic C++ syntax so that you can approach these questions with ease.

Classes and Inheritance

Though C++ classes have similar characteristics to those of other languages, we'll review some of the syntax below.

The code below demonstrates the implementation of a basic class with inheritance.

```
1  #include <iostream>
2  using namespace std;
3
4  #define NAME_SIZE 50 // Defines a macro
5
6  class Person {
7     int id; // all members are private by default
8     char name[NAME_SIZE];
9
10  public:
11    void aboutMe() {
12       cout << "I am a person.";
13    }
14 };
15
16 class Student : public Person {
17  public:
18    void aboutMe() {
19       cout << "I am a student.";
20    }
21 };
22
```

```
23 int main() {
24    Student * p = new Student();
25    p->aboutMe(); // prints "I am a student."
26    delete p; // Important! Make sure to delete allocated memory.
27    return 0;
28 }
```

All data members and methods are private by default in C++. One can modify this by introducing the keyword `public`.

Constructors and Destructors

The constructor of a class is automatically called upon an object's creation. If no constructor is defined, the compiler automatically generates one called the Default Constructor. Alternatively, we can define our own constructor.

```
1  Person(int a) {
2     id = a;
3  }
```

Fields within the class can also be initialized as follows:

```
1  Person(int a) : id(a) {
2     ...
3  }
```

The data member `id` is assigned before the actual object is created and before the remainder of the constructor code is called. It is particularly useful when we have constant fields that can only be assigned a value once.

The destructor cleans up upon object deletion and is automatically called when an object is destroyed. It cannot take an argument as we don't explicitly call a destructor.

```
1  ~Person() {
2     delete obj; // free any memory allocated within class
3  }
```

Virtual Functions

In an earlier example, we defined p to be of type `Student`:

```
1  Student * p = new Student();
2  p->aboutMe();
```

What would happen if we defined p to be a `Person*`, like so?

```
1  Person * p = new Student();
2  p->aboutMe();
```

In this case, "`I am a person`" would be printed instead. This is because the function aboutMe is resolved at compile-time, in a mechanism known as *static binding*.

If we want to ensure that the `Student`'s implementation of aboutMe is called, we can define aboutMe in the `Person` class to be `virtual`.

```
1  class Person {
```

```
    ...
    virtual void aboutMe() {
      cout << "I am a person.";
    }
};

class Student : public Person {
 public:
   void aboutMe() {
     cout << "I am a student.";
   }
};
```

Another usage for virtual functions is when we can't (or don't want to) implement a method for the parent class. Imagine, for example, that we want `Student` and `Teacher` to inherit from `Person` so that we can implement a common method such as `addCourse(string s)`. Calling `addCourse` on `Person`, however, wouldn't make much sense since the implementation depends on whether the object is actually a `Student` or `Teacher`.

In this case, we might want `addCourse` to be a virtual function defined within `Person`, with the implementation being left to the subclass.

```
class Person {
   int id; // all members are private by default
   char name[NAME_SIZE];
  public:
   virtual void aboutMe() {
     cout << "I am a person." << endl;
   }
   virtual bool addCourse(string s) = 0;
};

class Student : public Person {
  public:
   void aboutMe() {
     cout << "I am a student." << endl;
   }

   bool addCourse(string s) {
     cout << "Added course " << s << " to student." << endl;
     return true;
   }
};

int main() {
   Person * p = new Student();
   p->aboutMe(); // prints "I am a student."
   p->addCourse("History");
   delete p;
```

```
28 }
```

Note that by defining addCourse to be a "pure virtual function," Person is now an abstract class and we cannot instantiate it.

Virtual Destructor

The virtual function naturally introduces the concept of a "virtual destructor." Suppose we wanted to implement a destructor method for Person and Student. A naive solution might look like this:

```
1  class Person {
2   public:
3     ~Person() {
4        cout << "Deleting a person." << endl;
5     }
6  };
7
8  class Student : public Person {
9   public:
10    ~Student() {
11       cout << "Deleting a student." << endl;
12    }
13 };
14
15 int main() {
16    Person * p = new Student();
17    delete p; // prints "Deleting a person."
18 }
```

As in the earlier example, since p is a Person, the destructor for the Person class is called. This is problematic because the memory for Student may not be cleaned up.

To fix this, we simply define the destructor for Person to be virtual.

```
1  class Person {
2   public:
3     virtual ~Person() {
4        cout << "Deleting a person." << endl;
5     }
6  };
7
8  class Student : public Person {
9   public:
10    ~Student() {
11       cout << "Deleting a student." << endl;
12    }
13 };
14
15 int main() {
16    Person * p = new Student();
17    delete p;
```

```
18 }
```

This will output the following:

```
Deleting a student.
Deleting a person.
```

Default Values

Functions can specify default values, as shown below. Note that all default parameters must be on the right side of the function declaration, as there would be no other way to specify how the parameters line up.

```
1  int func(int a, int b = 3) {
2     x = a;
3     y = b;
4     return a + b;
5  }
6
7  w = func(4);
8  z = func(4, 5);
```

Operator Overloading

Operator overloading enables us to apply operators like + to objects that would otherwise not support these operations. For example, if we wanted to merge two `BookShelves` into one, we could overload the + operator as follows.

```
1  BookShelf BookShelf::operator+(BookShelf &other) { ... }
```

Pointers and References

A pointer holds the address of a variable and can be used to perform any operation that could be directly done on the variable, such as accessing and modifying it.

Two pointers can equal each other, such that changing one's value also changes the other's value (since they, in fact, point to the same address).

```
1  int * p = new int;
2  *p = 7;
3  int * q = p;
4  *p = 8;
5  cout << *q; // prints 8
```

Note that the size of a pointer varies depending on the architecture: 32 bits on a 32-bit machine and 64 bits on a 64-bit machine. Pay attention to this difference, as it's common for interviewers to ask exactly how much space a data structure takes up.

References

A reference is another name (an alias) for a pre-existing object and it does not have memory of its own. For example:

```
1  int a = 5;
```

```
2  int & b = a;
3  b = 7;
4  cout << a; // prints 7
```

In line 2 above, b is a reference to a; modifying b will also modify a.

You cannot create a reference without specifying where in memory it refers to. However, you can create a free-standing reference as shown below:

```
1  /* allocates memory to store 12 and makes b a reference to this
2   * piece of memory. */
3  int & b = 12;
```

Unlike pointers, references cannot be null and cannot be reassigned to another piece of memory.

Pointer Arithmetic

One will often see programmers perform addition on a pointer, such as what you see below:

```
1  int * p = new int[2];
2  p[0] = 0;
3  p[1] = 1;
4  p++;
5  cout << *p; // Outputs 1
```

Performing p++ will skip ahead by `sizeof(int)` bytes, such that the code outputs 1. Had p been of different type, it would skip ahead as many bytes as the size of the data structure.

Templates

Templates are a way of reusing code to apply the same class to different data types. For example, we might have a list-like data structure which we would like to use for lists of various types. The code below implements this with the `ShiftedList` class.

```
1  template <class T>
2  class ShiftedList {
3    T* array;
4    int offset, size;
5   public:
6    ShiftedList(int sz) : offset(0), size(sz) {
7      array = new T[size];
8    }
9
10   ~ShiftedList() {
11     delete [] array;
12   }
13
14   void shiftBy(int n) {
15     offset = (offset + n) % size;
16   }
```

```
17
18    T getAt(int i) {
19       return array[convertIndex(i)];
20    }
21
22    void setAt(T item, int i) {
23       array[convertIndex(i)] = item;
24    }
25
26 private:
27    int convertIndex(int i) {
28       int index = (i - offset) % size;
29       while (index < 0) index += size;
30       return index;
31    }
32 };
33
34 int main() {
35    int size = 4;
36    ShiftedList<int> * list = new ShiftedList<int>(size);
37    for (int i = 0; i < size; i++) {
38       list->setAt(i, i);
39    }
40    cout << list->getAt(0) << endl;
41    cout << list->getAt(1) << endl;
42    list->shiftBy(1);
43    cout << list->getAt(0) << endl;
44    cout << list->getAt(1) << endl;
45    delete list;
46 }
```

Interview Questions

13.1 Write a method to print the last K lines of an input file using C++.

pg 386

13.2 Compare and contrast a hash table and an STL map. How is a hash table implemented? If the number of inputs is small, which data structure options can be used instead of a hash table?

pg 387

13.3 How do virtual functions work in C++?

pg 388

13.4 What is the difference between deep copy and shallow copy? Explain how you would use each.

pg 389

13.5 What is the significance of the keyword "volatile" in C?

pg 389

13.6 Why does a destructor in base class need to be declared `virtual`?

pg 391

13.7 Write a method that takes a pointer to a `Node` structure as a parameter and returns a complete copy of the passed in data structure. The `Node` data structure contains two pointers to other `Node`s.

pg 391

13.8 Write a smart pointer class. A smart pointer is a data type, usually implemented with templates, that simulates a pointer while also providing automatic garbage collection. It automatically counts the number of references to a `SmartPointer<T*>` object and frees the object of type T when the reference count hits zero.

pg 392

13.9 Write an aligned malloc and free function that supports allocating memory such that the memory address returned is divisible by a specific power of two.

EXAMPLE

`align_malloc(1000,128)` will return a memory address that is a multiple of 128 and that points to memory of size 1000 bytes.

`aligned_free()` will free memory allocated by `align_malloc`.

pg 395

13.10 Write a function in C called `my2DAlloc` which allocates a two-dimensional array. Minimize the number of calls to `malloc` and make sure that the memory is accessible by the notation `arr[i][j]`.

pg 396

Additional Questions: Arrays and Strings (#1.2), Linked Lists (#2.7), Testing (#12.1), Java (#14.4), Threads and Locks (#16.3)

14

Java

While Java-related questions are found throughout this book, this chapter deals with questions about the language and syntax. Such questions are more unusual at bigger companies, which believe more in testing a candidate's aptitude than a candidate's knowledge (and which have the time and resources to train a candidate in a particular language). However, at other companies, these pesky questions can be quite common.

How to Approach

As these questions focus so much on knowledge, it may seem silly to talk about an approach to these problems. After all, isn't it just about knowing the right answer?

Yes and no. Of course, the best thing you can do to master these questions is to learn Java inside and out. But, if you do get stumped, you can try to tackle it with the following approach:

1. Create an example of the scenario, and ask yourself how things should play out.
2. Ask yourself how other languages would handle this scenario.
3. Consider how you would design this situation if you were the language designer. What would the implications of each choice be?

Your interviewer may be equally—or more—impressed if you can derive the answer than if you automatically knew it. Don't try to bluff though. Tell the interviewer, "I'm not sure I can recall the answer, but let me see if I can figure it out. Suppose we have this code..."

final keyword

The `final` keyword in Java has a different meaning depending on whether it is applied to a variable, class or method.

- *Variable:* The value cannot be changed once initialized.
- *Method:* The method cannot be overridden by a subclass.
- *Class:* The class cannot be subclassed.

finally keyword

The `finally` keyword is used in association with a `try/catch` block and guarantees that a section of code will be executed, even if an exception is thrown. The `finally` block will be executed after the `try` and `catch` blocks, but before control transfers back to its origin.

Watch how this plays out in the example below.

```
1  public static String lem() {
2     System.out.println("lem");
3     return "return from lem";
4  }
5
6  public static String foo() {
7     int x = 0;
8     int y = 5;
9     try {
10       System.out.println("start try");
11       int b = y / x;
12       System.out.println("end try");
13       return "returned from try";
14    } catch (Exception ex) {
15       System.out.println("catch");
16       return lem() + " | returned from catch";
17    } finally {
18       System.out.println("finally");
19    }
20 }
21
22 public static void bar() {
23    System.out.println("start bar");
24    String v = foo();
25    System.out.println(v);
26    System.out.println("end bar");
27 }
28
29 public static void main(String[] args) {
30    bar();
31 }
```

The output for this code is the following:

```
1  start bar
2  start try
3  catch
```

```
4  lem
5  finally
6  return from lem | returned from catch
7  end bar
```

Look carefully at lines 3 to 5 in the output. The `catch` block is fully executed (including the function call in the `return` statement), then the `finally` block, and then the function actually returns.

finalize method

The automatic garbage collector calls the `finalize()` method just before actually destroying the object. A class can therefore override the `finalize()` method from the `Object` class in order to define custom behavior during garbage collection.

```
1  protected void finalize() throws Throwable {
2     /* Close open files, release resources, etc */
3  }
```

Overloading vs. Overriding

Overloading is a term used to describe when two methods have the same name but differ in the type or number of arguments.

```
1  public double computeArea(Circle c) { ... }
2  public double computeArea(Square s) { ... }
```

Overriding, however, occurs when a method shares the same name and function signature as another method in its super class.

```
1  public abstract class Shape {
2     public void printMe() {
3        System.out.println("I am a shape.");
4     }
5     public abstract double computeArea();
6  }
7
8  public class Circle extends Shape {
9     private double rad = 5;
10    public void printMe() {
11       System.out.println("I am a circle.");
12    }
13
14    public double computeArea() {
15       return rad * rad * 3.15;
16    }
17 }
18
19 public class Ambiguous extends Shape {
20    private double area = 10;
21    public double computeArea() {
22       return area;
```

```
23    }
24 }
25
26 public class IntroductionOverriding {
27    public static void main(String[] args) {
28       Shape[] shapes = new Shape[2];
29       Circle circle = new Circle();
30       Ambiguous ambiguous = new Ambiguous();
31
32       shapes[0] = circle;
33       shapes[1] = ambiguous;
34
35       for (Shape s : shapes) {
36          s.printMe();
37          System.out.println(s.computeArea());
38       }
39    }
40 }
```

The above code will print:

```
1  I am a circle.
2  78.75
3  I am a shape.
4  10.0
```

Observe that `Circle` overrode `printMe()`, whereas `Ambiguous` just left this method as-is.

Collection Framework

Java's collection framework is incredibly useful, and you will see it used throughout this book. Here are some of the most useful items:

`ArrayList`: An `ArrayList` is a dynamically resizing array, which grows as you insert elements.

```
1  ArrayList<String> myArr = new ArrayList<String>();
2  myArr.add("one");
3  myArr.add("two");
4  System.out.println(myArr.get(0)); /* prints <one> */
```

`Vector`: A `vector` is very similar to an `ArrayList`, except that it is synchronized. Its syntax is almost identical as well.

```
1  Vector<String> myVect = new Vector<String>();
2  myVect.add("one");
3  myVect.add("two");
4  System.out.println(myVect.get(0));
```

`LinkedList`: `LinkedList` is, of course, Java's built-in `LinkedList` class. Though it rarely comes up in an interview, it's useful to study because it demonstrates some of the syntax for an iterator.

```
1  LinkedList<String> myLinkedList = new LinkedList<String>();
2  myLinkedList.add("two");
3  myLinkedList.addFirst("one");
4  Iterator<String> iter = myLinkedList.iterator();
5  while (iter.hasNext()) {
6     System.out.println(iter.next());
7  }
```

HashMap: The HashMap collection is widely used, both in interviews and in the real world. We've provided a snippet of the syntax below.

```
1  HashMap<String, String> map = new HashMap<String, String>();
2  map.put("one", "uno");
3  map.put("two", "dos");
4  System.out.println(map.get("one"));
```

Before your interview, make sure you're very comfortable with the above syntax. You'll need it.

Interview Questions

Please note that because virtually all the solutions in this book are implemented with Java, we have selected only a small number of questions for this chapter. Moreover, most of these questions deal with the "trivia" of the languages, since the rest of the book is filled with Java programming questions.

14.1 In terms of inheritance, what is the effect of keeping a constructor private?

pg 400

14.2 In Java, does the `finally` block get executed if we insert a `return` statement inside the try block of a `try-catch-finally`?

pg 400

14.3 What is the difference between `final`, `finally`, and `finalize`?

pg 400

14.4 Explain the difference between templates in C++ and generics in Java.

pg 401

14.5 Explain what object reflection is in Java and why it is useful.

pg 403

14.6 Implement a `CircularArray` class that supports an array-like data structure which can be efficiently rotated. The class should use a generic type, and should support iteration via the standard `for (Obj o : circularArray)` notation.

pg 404

Additional Questions: Arrays and Strings (#1.4), Object-Oriented Design (#8.10), Threads and Locks (#16.3)

15

Databases

Candidates who profess experience with databases may be asked to demonstrate this knowledge by implementing SQL queries or designing a database for an application. We'll review some of the key concepts and offer an overview of how to approach these problems.

As you read these queries, don't be surprised by minor variations in syntax. There are a variety of flavors of SQL, and you might have worked with a slightly different one. The examples in this book have been tested against Microsoft SQL Server.

SQL Syntax and Variations

Developers commonly use both the implicit join and the explicit join in SQL queries. Both syntaxes are shown below.

```
1  /* Explicit Join */
2  SELECT CourseName, TeacherName
3  FROM   Courses INNER JOIN Teachers
4  ON     Courses.TeacherID = Teachers.TeacherID
5
6  /* Implicit Join */
7  SELECT CourseName, TeacherName
8  FROM   Courses, Teachers
9  WHERE     Courses.TeacherID = Teachers.TeacherID
```

The two statements above are equivalent, and it's a matter of personal preference which one you choose. For consistency, we will stick to the explicit join.

Denormalized vs. Normalized Databases

Normalized databases are designed to minimize redundancy, while denormalized databases are designed to optimize read time.

In a traditional normalized database with data like `Courses` and `Teachers`, `Courses` might contain a column called `TeacherID`, which is a foreign key to `Teacher`. One benefit of this is that information about the teacher (name, address, etc.) is only stored

once in the database. The drawback is that many common queries will require expensive joins.

Instead, we can denormalize the database by storing redundant data. For example, if we knew that we would have to repeat this query often, we might store the teacher's name in the `Courses` table. Denormalization is commonly used to create highly scalable systems.

SQL Statements

Let's walk through a review of basic SQL syntax, using as an example the database that was mentioned earlier. This database has the following simple structure (* indicates a primary key):

```
Courses: CourseID*, CourseName, TeacherID
Teachers: TeacherID*, TeacherName
Students: StudentID*, StudentName
StudentCourses: CourseID*, StudentID*
```

Using the above table, implement the following queries.

Query 1: Student Enrollment

Implement a query to get a list of all students and how many courses each student is enrolled in.

At first, we might try something like this:

```
1  /* Incorrect Code */
2  SELECT Students.StudentName, count(*)
3  FROM Students INNER JOIN StudentCourses
4  ON Students.StudentID = StudentCourses.StudentID
5  GROUP BY Students.StudentID
```

This has three problems:

1. We have excluded students who are not enrolled in any courses, since `StudentCourses` only includes enrolled students. We need to change this to a `LEFT JOIN`.
2. Even if we changed it to a `LEFT JOIN`, the query is still not quite right. Doing `count(*)` would return how many items there are in a given group of `StudentIDs`. Students enrolled in zero courses would still have one item in their group. We need to change this to count the number of `CourseIDs` in each group: `count(StudentCourses.CourseID)`.
3. We've grouped by `Students.StudentID`, but there are still multiple `StudentNames` in each group. How will the database know which `StudentName` to return? Sure, they may all have the same value, but the database doesn't understand that. We need to apply an *aggregate* function to this, such as `first(Students.StudentName)`.

Fixing these issues gets us to this query:

```
1  /* Solution 1: Wrap with another query */
2  SELECT StudentName, Students.StudentID, Cnt
3  FROM (
4     SELECT  Students.StudentID,
5             count(StudentCourses.CourseID) as [Cnt]
6     FROM Students LEFT JOIN StudentCourses
7     ON Students.StudentID = StudentCourses.StudentID
8     GROUP BY Students.StudentID
9  ) T INNER JOIN Students on T.studentID = Students.StudentID
```

Looking at this code, one might ask why we don't just select the student name on line 3 to avoid having to wrap lines 3 through 6 with another query. This (incorrect) solution is shown below.

```
1  /* Incorrect Code */
1  SELECT StudentName, Students.StudentID,
2         count(StudentCourses.CourseID) as [Cnt]
3  FROM Students LEFT JOIN StudentCourses
4  ON Students.StudentID = StudentCourses.StudentID
5  GROUP BY Students.StudentID
```

The answer is that we *can't* do that - at least not exactly as shown. We can only select values that are in an aggregate function or in the GROUP BY clause.

Alternatively, we could resolve the above issues with either of the following statements:

```
1  /* Solution 2: Add StudentName to GROUP BY clause. */
2  SELECT StudentName, Students.StudentID,
3         count(StudentCourses.CourseID) as [Cnt]
4  FROM Students LEFT JOIN StudentCourses
5  ON Students.StudentID = StudentCourses.StudentID
6  GROUP BY Students.StudentID, Students.StudentName
```

OR

```
1  /* Solution 3: Wrap with aggregate function. */
2  SELECT  max(StudentName) as [StudentName], Students.StudentID,
3          count(StudentCourses.CourseID) as [Count]
4  FROM Students LEFT JOIN StudentCourses
5  ON Students.StudentID = StudentCourses.StudentID
6  GROUP BY Students.StudentID
```

Query 2: Teacher Class Size

Implement a query to get a list of all teachers and how many students they each teach. If a teacher teaches the same student in two courses, you should double count the student. Sort the list in descending order of the number of students a teacher teaches.

We can construct this query step by step. First, let's get a list of `TeacherIDs` and how many students are associated with each `TeacherID`. This is very similar to the earlier query.

```
1  SELECT TeacherID, count(StudentCourses.CourseID) AS [Number]
```

```
2  FROM Courses INNER JOIN StudentCourses
3  ON Courses.CourseID = StudentCourses.CourseID
4  GROUP BY Courses.TeacherID
```

Note that this `INNER JOIN` will not select teachers who aren't teaching classes. We'll handle that in the below query when we join it with the list of all teachers.

```
1  SELECT TeacherName, isnull(StudentSize.Number, 0)
2  FROM Teachers LEFT JOIN
3       (SELECT TeacherID, count(StudentCourses.CourseID) AS [Number]
4        FROM Courses INNER JOIN StudentCourses
5        ON Courses.CourseID = StudentCourses.CourseID
6        GROUP BY Courses.TeacherID) StudentSize
7  ON Teachers.TeacherID = StudentSize.TeacherID
8  ORDER BY StudentSize.Number DESC
```

Note how we handled the `NULL` values in the `SELECT` statement to convert the `NULL` values to zeros.

Small Database Design

Additionally, you might be asked to design your own database. We'll walk you through an approach for this. You might notice the similarities between this approach and the approach for object-oriented design.

Step 1: Handle Ambiguity

Database questions often have some ambiguity, intentionally or unintentionally. Before you proceed with your design, you must understand exactly what you need to design.

Imagine you are asked to design a system to represent an apartment rental agency. You will need to know whether this agency has multiple locations or just one. You should also discuss with your interviewer how general you should be. For example, it would be extremely rare for a person to rent two apartments in the same building. But does that mean you shouldn't be able to handle that? Maybe, maybe not. Some very rare conditions might be best handled through a work around (like duplicating the person's contact information in the database).

Step 2: Define the Core Objects

Next, we should look at the core objects of our system. Each of these core objects typically translates into a table. In this case, our core objects might be `Property`, `Building`, `Apartment`, `Tenant` and `Manager`.

Step 3: Analyze Relationships

Outlining the core objects should give us a good sense of what the tables should be. How do these tables relate to each other? Are they many-to-many? One-to-many?

If `Buildings` has a one-to-many relationship with `Apartments` (one `Building` has many `Apartments`), then we might represent this as follows:

Buildings		
BuildingID	BuildingName	BuildingAddress

Apartments		
ApartmentID	ApartmentAddress	BuildingID

Note that the `Apartments` table links back to `Buildings` with a `BuildingID` column.

If we want to allow for the possibility that one person rents more than one apartment, we might want to implement a many-to-many relationship as follows:

Tenants		
TenantID	TenantName	TenantAddress

Apartments		
ApartmentID	ApartmentAddress	BuildingID

TenantApartments	
TenantID	ApartmentID

The `TenantApartments` table stores a relationship between `Tenants` and `Apartments`.

Step 4: Investigate Actions

Finally, we fill in the details. Walk through the common actions that will be taken and understand how to store and retrieve the relevant data. We'll need to handle lease terms, moving out, rent payments, etc. Each of these actions requires new tables and columns.

Large Database Design

When designing a large, scalable database, joins (which are required in the above examples) are generally very slow. Thus, you must *denormalize* your data. Think carefully about how data will be used—you'll probably need to duplicate the data in multiple tables.

Interview Questions

Questions 1 through 3 refer to the below database schema:

Apartments		Buildings		Tenants	
AptID	int	BuildingID	int	TenantID	int
UnitNumber	varchar	ComplexID	int	TenantName	varchar

Apartments		Buildings		Tenants	
BuildingID	int	BuildingName	varchar		
		Address	varchar		

Complexes		AptTenants		Requests	
ComplexID	int	TenantID	int	RequestID	int
ComplexName	varchar	AptID	int	Status	varchar
				AptID	int
				Description	varchar

Note that each apartment can have multiple tenants, and each tenant can have multiple apartments. Each apartment belongs to one building, and each building belongs to one complex.

15.1 Write a SQL query to get a list of tenants who are renting more than one apartment.

pg 408

15.2 Write a SQL query to get a list of all buildings and the number of open requests (`Requests` in which `status` equals `'Open'`).

pg 408

15.3 Building #11 is undergoing a major renovation. Implement a query to close all requests from apartments in this building.

pg 409

15.4 What are the different types of joins? Please explain how they differ and why certain types are better in certain situations.

pg 409

15.5 What is denormalization? Explain the pros and cons.

pg 411

15.6 Draw an entity-relationship diagram for a database with companies, people, and professionals (people who work for companies).

pg 412

15.7 Imagine a simple database storing information for students' grades. Design what this database might look like and provide a SQL query to return a list of the honor roll students (top 10%), sorted by their grade point average.

pg 412

Additional Questions: Object-Oriented Design (#8.6)

16

Threads and Locks

In a Microsoft, Google or Amazon interview, it's not terribly common to be asked to implement an algorithm with threads (unless you're working in a team for which this is a particularly important skill). It is, however, relatively common for interviewers at any company to assess your general understanding of threads, particularly your understanding of deadlocks.

This chapter will provide an introduction to this topic.

Threads in Java

Every thread in Java is created and controlled by a unique object of the `java.lang.Thread` class. When a standalone application is run, a user thread is automatically created to execute the `main()` method. This thread is called the main thread.

In Java, we can implement threads in one of two ways:

- By implementing the `java.lang.Runnable` interface
- By extending the `java.lang.Thread` class

We will cover both of these below.

Implementing the Runnable Interface

The `Runnable` interface has the following very simple structure.

```
1  public interface Runnable {
2    void run();
3  }
```

To create and use a thread using this interface, we do the following:

1. Create a class which implements the `Runnable` interface. An object of this class is a `Runnable` object.
2. Create an object of type `Thread` by passing a `Runnable` object as argument to the `Thread` constructor. The `Thread` object now has a `Runnable` object that implements the `run()` method.

3. The `start()` method is invoked on the `Thread` object created in the previous step.

For example:

```
1  public class RunnableThreadExample implements Runnable {
2     public int count = 0;
3
4     public void run() {
5        System.out.println("RunnableThread starting.");
6        try {
7           while (count < 5) {
8              Thread.sleep(500);
9              count++;
10          }
11       } catch (InterruptedException exc) {
12          System.out.println("RunnableThread interrupted.");
13       }
14       System.out.println("RunnableThread terminating.");
15    }
16 }
17
18 public static void main(String[] args) {
19    RunnableThreadExample instance = new RunnableThreadExample();
20    Thread thread = new Thread(instance);
21    thread.start();
22
23    /* waits until above thread counts to 5 (slowly) */
24    while (instance.count != 5) {
25       try {
26          Thread.sleep(250);
27       } catch (InterruptedException exc) {
28          exc.printStackTrace();
29       }
30    }
31 }
```

In the above code, observe that all we really needed to do is have our class implement the `run()` method (line 4). Another method can then pass an instance of the class to `new Thread(obj)` (lines 19 - 20) and call `start()` on the thread (line 21).

Extending the Thread Class

Alternatively, we can create a thread by extending the `Thread` class. This will almost always mean that we override the `run()` method, and the subclass may also call the thread constructor explicitly in its constructor.

The below code provides an example of this.

```
1  public class ThreadExample extends Thread {
2     int count = 0;
3
4     public void run() {
```

```
5         System.out.println("Thread starting.");
6         try {
7            while (count < 5) {
8               Thread.sleep(500);
9               System.out.println("In Thread, count is " + count);
10              count++;
11           }
12        } catch (InterruptedException exc) {
13           System.out.println("Thread interrupted.");
14        }
15        System.out.println("Thread terminating.");
16     }
17  }
18
19  public class ExampleB {
20     public static void main(String args[]) {
21        ThreadExample instance = new ThreadExample();
22        instance.start();
23
24        while (instance.count != 5) {
25           try {
26              Thread.sleep(250);
27           } catch (InterruptedException exc) {
28              exc.printStackTrace();
29           }
30        }
31     }
32  }
```

This code is very similar to the first approach. The difference is that since we are extending the `Thread` class, rather than just implementing an interface, we can call `start()` on the instance of the class itself.

Extending the Thread Class vs. Implementing the Runnable Interface

When creating threads, there are two reasons why implementing the `Runnable` interface may be preferable to extending the `Thread` class:

- Java does not support multiple inheritance. Therefore, extending the `Thread` class means that the subclass cannot extend any other class. A class implementing the `Runnable` interface will be able to extend another class.
- A class might only be interested in being runnable, and therefore, inheriting the full overhead of the `Thread` class would be excessive.

Synchronization and Locks

Threads within a given process share the same memory space, which is both a positive and a negative. It enables threads to share data, which can be valuable. However, it also creates the opportunity for issues when two threads modify a resource at the same

time. Java provides synchronization in order to control access to shared resources.

The keyword `synchronized` and the `lock` form the basis for implementing synchronized execution of code.

Synchronized Methods

Most commonly, we restrict access to shared resources through the use of the `synchronized` keyword. It can be applied to methods and code blocks, and restricts multiple threads from executing the code simultaneously *on the same object*.

To clarify the last point, consider the following code:

```
1  public class MyClass extends Thread  {
2     private String name;
3     private MyObject myObj;
4
5     public MyClass(MyObject obj, String n) {
6        name = n;
7        myObj = obj;
8     }
9
10    public void run() {
11       myObj.foo(name);
12    }
13 }
14
15 public class MyObject {
16    public synchronized void foo(String name) {
17       try {
18          System.out.println("Thread " + name + ".foo(): starting");
19          Thread.sleep(3000);
20          System.out.println("Thread " + name + ".foo(): ending");
21       } catch (InterruptedException exc) {
22          System.out.println("Thread " + name + ": interrupted.");
23       }
24    }
25 }
```

Can two instances of `MyClass` call `foo` at the same time? It depends. If they have the same instance of `MyObject`, then no. But, if they hold different references, then the answer is yes.

```
1  /* Difference references - both threads can call MyObject.foo() */
2  MyObject obj1 = new MyObject();
3  MyObject obj2 = new MyObject();
4  MyClass thread1 = new MyClass(obj1, "1");
5  MyClass thread2 = new MyClass(obj2, "2");
6  thread1.start();
7  thread2.start()
8
9  /* Same reference to obj. Only one will be allowed to call foo,
```

```
10   * and the other will be forced to wait. */
11  MyObject obj = new MyObject();
12  MyClass thread1 = new MyClass(obj, "1");
13  MyClass thread2 = new MyClass(obj, "2");
14  thread1.start()
15  thread2.start()
```

Static methods synchronize on the *class lock*. The two threads above could not simultaneously execute synchronized static methods on the same class, even if one is calling foo and the other is calling bar.

```
1   public class MyClass extends Thread  {
2      ...
3      public void run() {
4         if (name.equals("1")) MyObject.foo(name);
5         else if (name.equals("2")) MyObject.bar(name);
6      }
7   }
8
9   public class MyObject {
10     public static synchronized void foo(String name) {
11        /* same as before */
12     }
13
14     public static synchronized void bar(String name) {
15        /* same as foo */
16     }
17  }
```

If you run this code, you will see the following printed:

```
Thread 1.foo(): starting
Thread 1.foo(): ending
Thread 2.bar(): starting
Thread 2.bar(): ending
```

Synchronized Blocks

Similarly, a block of code can be synchronized. This operates very similarly to synchronizing a method.

```
1   public class MyClass extends Thread  {
2      ...
3      public void run() {
4         myObj.foo(name);
5      }
6   }
7   public class MyObject {
8      public void foo(String name) {
9         synchronized(this) {
10           ...
11        }
12     }
```

```
13 }
```

Like synchronizing a method, only one thread per instance of `MyObject` can execute the code within the `synchronized` block. That means that, if `thread1` and `thread2` have the same instance of `MyObject`, only one will be allowed to execute the code block at a time.

Locks

For more granular control, we can utilize a lock. A lock (or monitor) is used to synchronize access to a shared resource by associating the resource with the lock. A thread gets access to a shared resource by first acquiring the lock associated with the resource. At any given time, at most one thread can hold the lock and, therefore, only one thread can access the shared resource.

A common use case for locks is when a resource is accessed from multiple places, but should be only accessed by one thread *at a time*. This case is demonstrated in the code below.

```
1  public class LockedATM {
2     private Lock lock;
3     private int balance = 100;
4
5     public LockedATM() {
6        lock = new ReentrantLock();
7     }
8
9     public int withdraw(int value) {
10       lock.lock();
11       int temp = balance;
12       try {
13          Thread.sleep(100);
14          temp = temp - value;
15          Thread.sleep(100);
16          balance = temp;
17       } catch (InterruptedException e) {      }
18       lock.unlock();
19       return temp;
20    }
21
22    public int deposit(int value) {
23       lock.lock();
24       int temp = balance;
25       try {
26          Thread.sleep(100);
27          temp = temp + value;
28          Thread.sleep(300);
29          balance = temp;
30       } catch (InterruptedException e) {      }
31       lock.unlock();
```

```
32        return temp;
33    }
34 }
```

Of course, we've added code to intentionally slow down the execution of `withdraw` and `deposit`, as it helps to illustrate the potential problems that can occur. You may not write code exactly like this, but the situation it mirrors is very, very real. Using a lock will help protect a shared resource from being modified in unexpected ways.

Deadlocks and Deadlock Prevention

A deadlock is a situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds (or an equivalent situation with several threads). Since each thread is waiting for the other thread to relinquish a lock, they both remain waiting forever. The threads are said to be deadlocked.

In order for a deadlock to occur, you must have all four of the following conditions met:

1. *Mutual Exclusion:* Only one process can access a resource at a given time. (Or, more accurately, there is limited access to a resource. A deadlock could also occur if a resource has limited quantity.)
2. *Hold and Wait:* Processes already holding a resource can request additional resources, without relinquishing their current resources.
3. *No Preemption:* One process cannot forcibly remove another process' resource.
4. *Circular Wait:* Two or more processes form a circular chain where each process is waiting on another resource in the chain.

Deadlock prevention entails removing any of the above conditions, but it gets tricky because many of these conditions are difficult to satisfy. For instance, removing #1 is difficult because many resources can only be used by one process at a time (e.g., printers). Most deadlock prevention algorithms focus on avoiding condition #4: circular wait.

Interview Questions

16.1 What's the difference between a thread and a process?

pg 416

16.2 How would you measure the time spent in a context switch?

pg 416

16.3 In the famous dining philosophers problem, a bunch of philosophers are sitting around a circular table with one chopstick between each of them. A philosopher needs both chopsticks to eat, and always picks up the left chopstick before the right one. A deadlock could potentially occur if all the philosophers reached for the left chopstick at the same time. Using threads and locks, implement a simulation of the dining philosophers problem that prevents deadlocks.

pg 418

16.4 Design a class which provides a lock only if there are no possible deadlocks.

pg 420

16.5 Suppose we have the following code:

```
public class Foo {
   public Foo() { ... }
   public void first() { ... }
   public void second() { ... }
   public void third() { ... }
}
```

The same instance of `Foo` will be passed to three different threads. `ThreadA` will call `first`, `threadB` will call `second`, and `threadC` will call `third`. Design a mechanism to ensure that `first` is called before `second` and `second` is called before `third`.

pg 425

16.6 You are given a class with synchronized method A and a normal method B. If you have two threads in one instance of a program, can they both execute A at the same time? Can they execute A and B at the same time?

pg 427

Additional Review Problems

Interview Questions and Advice

17

Moderate

17.1 Write a function to swap a number in place (that is, without temporary variables).

pg 430

17.2 Design an algorithm to figure out if someone has won a game of tic-tac-toe.

pg 431

17.3 Write an algorithm which computes the number of trailing zeros in n factorial.

pg 434

17.4 Write a method which finds the maximum of two numbers. You should not use if-else or any other comparison operator.

pg 436

17.5 The Game of Master Mind is played as follows:

The computer has four slots, and each slot will contain a ball that is red (R), yellow (Y), green (G) or blue (B). For example, the computer might have RGGB (Slot #1 is red, Slots #2 and #3 are green, Slot #4 is blue).

You, the user, are trying to guess the solution. You might, for example, guess YRGB.

When you guess the correct color for the correct slot, you get a "hit." If you guess a color that exists but is in the wrong slot, you get a "pseudo-hit." Note that a slot that is a hit can never count as a pseudo-hit.

For example, if the actual solution is RGBY and you guess GGRR, you have one hit and one pseudo-hit.

Write a method that, given a guess and a solution, returns the number of hits and pseudo-hits.

pg 438

17.6 Given an array of integers, write a method to find indices `m` and `n` such that if you sorted elements `m` through `n`, the entire array would be sorted. Minimize `n - m` (that is, find the smallest such sequence).

EXAMPLE

Input: `1, 2, 4, 7, 10, 11, 7, 12, 6, 7, 16, 18, 19`

Output: `(3, 9)`

pg 439

17.7 Given any integer, print an English phrase that describes the integer (e.g., "One Thousand, Two Hundred Thirty Four").

pg 442

17.8 You are given an array of integers (both positive and negative). Find the contiguous sequence with the largest sum. Return the sum.

EXAMPLE

Input: `2, -8, 3, -2, 4, -10`

Output: `5 (i.e., {3, -2, 4})`

pg 443

17.9 Design a method to find the frequency of occurrences of any given word in a book.

pg 445

17.10 Since XML is very verbose, you are given a way of encoding it where each tag gets mapped to a pre-defined integer value. The language/grammar is as follows:

```
Element   --> Tag Attributes END Children END
Attribute --> Tag Value
END       --> 0
Tag       --> some predefined mapping to int
Value     --> string value END
```

For example, the following XML might be converted into the compressed string below (assuming a mapping of `family -> 1, person ->2, firstName -> 3, lastName -> 4, state -> 5`).

```
<family lastName="McDowell" state="CA">
   <person firstName="Gayle">Some Message</person>
</family>
```

Becomes:

```
1 4 McDowell 5 CA 0 2 3 Gayle 0 Some Message 0 0.
```

Write code to print the encoded version of an XML element (passed in `Element` and `Attribute` objects).

pg 446

17.11 Implement a method `rand7()` given `rand5()`. That is, given a method that generates a random number between 0 and 4 (inclusive), write a method that generates a random number between 0 and 6 (inclusive).

pg 447

17.12 Design an algorithm to find all pairs of integers within an array which sum to a specified value.

pg 450

17.13 Consider a simple node-like data structure called `BiNode`, which has pointers to two other nodes.

```
1 public class BiNode {
2    public BiNode node1, node2;
3    public int data;
4 }
```

The data structure `BiNode` could be used to represent both a binary tree (where `node1` is the left node and `node2` is the right node) or a doubly linked list (where `node1` is the previous node and `node2` is the next node). Implement a method to convert a binary search tree (implemented with `BiNode`) into a doubly linked list. The values should be kept in order and the operation should be performed in place (that is, on the original data structure).

pg 451

17.14 Oh, no! You have just completed a lengthy document when you have an unfortunate Find/Replace mishap. You have accidentally removed all spaces, punctuation, and capitalization in the document. A sentence like "`I reset the computer. It still didn't boot!`" would become "`iresetthecomputeritstilldidntboot`". You figure that you can add back in the punctation and capitalization later, once you get the individual words properly separated. Most of the words will be in a dictionary, but some strings, like proper names, will not.

Given a dictionary (a list of words), design an algorithm to find the optimal way of "unconcatenating" a sequence of words. In this case, "optimal" is defined to be the parsing which minimizes the number of unrecognized sequences of characters.

For example, the string "`jesslookedjustliketimherbrother`" would be optimally parsed as "`JESS looked just like TIM her brother`". This parsing has seven unrecognized characters, which we have capitalized for clarity.

pg 455

18

Hard

18.1 Write a function that adds two numbers. You should not use + or any arithmetic operators.

pg 462

18.2 Write a method to shuffle a deck of cards. It must be a perfect shuffle—in other words, each of the 52! permutations of the deck has to be equally likely. Assume that you are given a random number generator which is perfect.

pg 463

18.3 Write a method to randomly generate a set of `m` integers from an array of size n. Each element must have equal probability of being chosen.

pg 464

18.4 Write a method to count the number of 2s that appear in all the numbers between 0 and n (inclusive).

EXAMPLE

Input: 25

Output: 9 (2, 12, 20, 21, 22, 23, 24 and 25. Note that 22 counts for two 2s.)

pg 465

18.5 You have a large text file containing words. Given any two words, find the shortest distance (in terms of number of words) between them in the file. If the operation will be repeated many times for the same file (but different pairs of words), can you optimize your solution?

pg 468

18.6 Describe an algorithm to find the smallest one million numbers in one billion numbers. Assume that the computer memory can hold all one billion numbers.

pg 469

18.7 Given a list of words, write a program to find the longest word made of other words in the list.

EXAMPLE

Input: `cat, banana, dog, nana, walk, walker, dogwalker`

Output: `dogwalker`

pg 471

18.8 Given a string s and an array of smaller strings T, design a method to search s for each small string in T.

pg 473

18.9 Numbers are randomly generated and passed to a method. Write a program to find and maintain the median value as new values are generated.

pg 474

18.10 Given two words of equal length that are in a dictionary, write a method to transform one word into another word by changing only one letter at a time. The new word you get in each step must be in the dictionary.

EXAMPLE

Input: DAMP, LIKE

Output: DAMP -> LAMP -> LIMP -> LIME -> LIKE

pg 476

18.11 Imagine you have a square matrix, where each cell (pixel) is either black or white Design an algorithm to find the maximum subsquare such that all four borders are filled with black pixels.

pg 477

18.12 Given an NxN matrix of positive and negative integers, write code to find the submatrix with the largest possible sum.

pg 481

18.13 Given a list of millions of words, design an algorithm to create the largest possible rectangle of letters such that every row forms a word (reading left to right) and every column forms a word (reading top to bottom). The words need not be chosen consecutively from the list, but all rows must be the same length and all columns must be the same height.

pg 485

Solutions

IX

*Join us at **www.CrackingTheCodingInterview.com** to download full, compilable Java / Eclipse solutions, discuss problems from this book with other readers, report issues, view this book's errata, post your resume, and seek additional advice.*

Arrays and Strings

1.1 *Implement an algorithm to determine if a string has all unique characters. What if you cannot use additional data structures?*

pg 73

SOLUTION

You may want to start off with asking your interviewer if the string is an ASCII string or a Unicode string. This is an important question, and asking it will show an eye for detail and a deep understanding of Computer Science.

We'll assume for simplicity that the character set is ASCII. If not, we would need to increase the storage size, but the rest of the logic would be the same.

Given this, one simple optimization we can make to this problem is to automatically return `false` if the length of the string is greater than the number of unique characters in the alphabet. After all, you can't have a string with 280 unique characters if there are only 256 possible unique characters.

Our first solution is to create an array of boolean values, where the flag at index `i` indicates whether character `i` in the alphabet is contained in the string. If you run across this character a second time, you can immediately return `false`.

The code below implements this algorithm.

```
1  public boolean isUniqueChars2(String str) {
2     if (str.length() > 256) return false;
3
4     boolean[] char_set = new boolean[256];
5     for (int i = 0; i < str.length(); i++) {
6        int val = str.charAt(i);
7        if (char_set[val]) { // Already found this char in string
8           return false;
9        }
10       char_set[val] = true;
11    }
12    return true;
13 }
```

The time complexity for this code is `O(n)`, where n is the length of the string. The space complexity is `O(1)`.

We can reduce our space usage by a factor of eight by using a bit vector. We will assume, in the below code, that the string only uses the lower case letters a through z. This will allow us to use just a single `int`.

```
1  public boolean isUniqueChars(String str) {
2
3
4     int checker = 0;
5     for (int i = 0; i < str.length(); i++) {
6        int val = str.charAt(i) - 'a';
```

```
7         if ((checker & (1 << val)) > 0) {
8            return false;
9         }
10        checker |= (1 << val);
11     }
12     return true;
13  }
```

Alternatively, we could do the following:

1. Compare every character of the string to every other character of the string. This will take $O(n^2)$ time and $O(1)$ space.
2. If we are allowed to modify the input string, we could sort the string in $O(n \log(n))$ time and then linearly check the string for neighboring characters that are identical. Careful, though: many sorting algorithms take up extra space.

These solutions are not as optimal in some respects, but might be better depending on the constraints of the problem.

1.2 *Implement a function void reverse(char* str) in C or C++ which reverses a null-terminated string.*

pg 73

SOLUTION

This is a classic interview question. The only "gotcha" is to try to do it in place, and to be careful for the `null` character.

We will implement this in C.

```
1  void reverse(char *str) {
2     char* end = str;
3     char tmp;
4     if (str) {
5        while (*end) { /* find end of the string */
6           ++end;
7        }
8        --end; /* set one char back, since last char is null */
9
10       /* swap characters from start of string with the end of the
11        * string, until the pointers meet in middle. */
12       while (str < end) {
13          tmp = *str;
14          *str++ = *end;
15          *end-- = tmp;
16       }
17    }
18 }
```

This is just one of many ways to implement this solution. We could even implement this

code recursively (but we wouldn't recommend it).

1.3 *Given two strings, write a method to decide if one is a permutation of the other.*

pg 73

SOLUTION

Like in many questions, we should confirm some details with our interviewer. We should understand if the anagram comparison is case sensitive. That is, is God an anagram of dog? Additionally, we should ask if whitespace is significant.

We will assume for this problem that the comparison is case sensitive and whitespace is significant. So, "god " is different from "dog".

Whenever we compare two strings, we know that if they are different lengths then they cannot be anagrams.

There are two easy ways to solve this problem, both of which use this optimization.

Solution #1: Sort the strings.

If two strings are anagrams, then we know they have the same characters, but in different orders. Therefore, sorting the strings will put the characters from two anagrams in the same order. We just need to compare the sorted versions of the strings.

```
1  public String sort(String s) {
2     char[] content = s.toCharArray();
3     java.util.Arrays.sort(content);
4     return new String(content);
5  }
6
7  public boolean permutation(String s, String t) {
8     if (s.length() != t.length()) {
9        return false;
10    }
11    return sort(s).equals(sort(t));
12 }
```

Though this algorithm is not as optimal in some senses, it may be preferable in one sense: it's clean, simple and easy to understand. In a practical sense, this may very well be a superior way to implement the problem.

However, if efficiency is very important, we can implement it a different way.

Solution #2: Check if the two strings have identical character counts.

We can also use the definition of an anagram—two words with the same character counts—to implement this algorithm. We simply iterate through this code, counting how many times each character appears. Then, afterwards, we compare the two arrays.

```
1  public boolean permutation(String s, String t) {
2     if (s.length() != t.length()) {
3        return false;
4     }
5
6     int[] letters = new int[256]; // Assumption
7
8     char[] s_array = s.toCharArray();
9     for (char c : s_array) { // count number of each char in s.
10       letters[c]++;
11    }
12
13    for (int i = 0; i < t.length(); i++) {
14       int c = (int) t.charAt(i);
15       if (--letters[c] < 0) {
16          return false;
17       }
18    }
19
20    return true;
21 }
```

Note the assumption on line 6. In your interview, you should always check with your interviewer about the size of the character set. We assumed that the character set was ASCII.

1.4 *Write a method to replace all spaces in a string with '%20'. You may assume that the string has sufficient space at the end of the string to hold the additional characters, and that you are given the "true" length of the string. (Note: if implementing in Java, please use a character array so that you can perform this operation in place.)*

pg 73

SOLUTION

A common approach in string manipulation problems is to edit the string starting from the end and work backwards. This is useful because we have extra buffer at the end, which allows us to change characters without worrying about what we're overwriting.

We will use this approach in this problem. The algorithm works through a two scan approach. In the first scan, we count how many spaces there are in the string. This is used to compute how long the final string should be. In the second pass, which is done in reverse order, we actually edit the string. When we see a space, we copy %20 into the next spots. If there is no space, then we copy the original character.

The code below implements this algorithm.

```
1  public void replaceSpaces(char[] str, int length) {
2     int spaceCount = 0, newLength, i;
3     for (i = 0; i < length; i++) {
```

```
4         if (str[i] == ' ') {
5           spaceCount++;
6         }
7       }
8       newLength = length + spaceCount * 2;
9       str[newLength] = '\0';
10      for (i = length - 1; i >= 0; i--) {
11        if (str[i] == ' ') {
12          str[newLength - 1] = '0';
13          str[newLength - 2] = '2';
14          str[newLength - 3] = '%';
15          newLength = newLength - 3;
16        } else {
17          str[newLength - 1] = str[i];
18          newLength = newLength - 1;
19        }
20      }
21  }
```

We have implemented this problem using character arrays, since Java strings are immutable. If we used strings directly, this would require returning a new copy of the string, but it would allow us to implement this in just one pass.

1.5 *Implement a method to perform basic string compression using the counts of repeated characters. For example, the string* `aabcccccaaa` *would become* `a2b1c5a3`. *If the "compressed" string would not become smaller than the original string, your method should return the original string.*

pg 73

SOLUTION

At first glance, implementing this method seems fairly straightforward, but perhaps a bit tedious. We iterate through the string, copying characters to a new string and counting the repeats. How hard could it be?

```
1   public String compressBad(String str) {
2     String mystr = "";
3     char last = str.charAt(0);
4     int count = 1;
5     for (int i = 1; i < str.length(); i++) {
6       if (str.charAt(i) == last) { // Found repeat char
7         count++;
8       } else { // Insert char count, and update last char
9         mystr += last + "" + count;
10        last = str.charAt(i);
11        count = 1;
12      }
13    }
14    return mystr + last + count;
```

```
15 }
```

This code doesn't handle the case when the compressed string is longer than the original string, but it otherwise works. Is it efficient though? Take a look at the runtime of this code.

The runtime is $O(p + k^2)$, where p is the size of the original string and k is the number of character sequences. For example, if the string is aabccdeeaa, then there are six character sequences. It's slow because string concatenation operates in $O(n^2)$ time (see **StringBuffer** in Chapter 1).

We can make this somewhat better by using a StringBuffer.

```
1  String compressBetter(String str) {
2     /* Check if compression would create a longer string */
3     int size = countCompression(str);
4     if (size >= str.length()) {
5        return str;
6     }
7
8     StringBuffer mystr = new StringBuffer();
9     char last = str.charAt(0);
10    int count = 1;
11    for (int i = 1; i < str.length(); i++) {
12       if (str.charAt(i) == last) { // Found repeated char
13          count++;
14       } else { // Insert char count, and update last char
15          mystr.append(last); // Insert char
16          mystr.append(count); // Insert count
17          last = str.charAt(i);
18          count = 1;
19       }
20    }
21
22    /* In lines 15 - 16 above, characters are inserted when the
23     * repeated character changes. We need to update the string at
24     * the end of the method as well, since the very last set of
25     * repeated characters wouldn't be set in the compressed string
26     * yet. */
27    mystr.append(last);
28    mystr.append(count);
29    return mystr.toString();
30 }
31
32 int countCompression(String str) {
33    if (str == null || str.isEmpty()) return 0;
34    char last = str.charAt(0);
35    int size = 0;
36    int count = 1;
37    for (int i = 1; i < str.length(); i++) {
38       if (str.charAt(i) == last) {
```

```
39          count++;
40        } else {
41          last = str.charAt(i);
42          size += 1 + String.valueOf(count).length();
43          count = 1;
44        }
45      }
46      size += 1 + String.valueOf(count).length();
47      return size;
48  }
```

This algorithm is much better. Note that we have added the size check in lines 2 through 5.

If we don't want to (or aren't allowed to) use a `StringBuffer`, we can still solve this problem efficiently. In line 2, we compute the end size of the string. This allows us to create a `char` array of the correct size, so we can implement the code as follows:

```
1   String compressAlternate(String str) {
2     /* Check if compression would create a longer string */
3     int size = countCompression(str);
4     if (size >= str.length()) {
5       return str;
6     }
7
8     char[] array = new char[size];
9     int index = 0;
10    char last = str.charAt(0);
11    int count = 1;
12    for (int i = 1; i < str.length(); i++) {
13      if (str.charAt(i) == last) { // Found repeated character
14        count++;
15      } else {
16        /* Update the repeated character count */
17        index = setChar(array, last, index, count);
18        last = str.charAt(i);
19        count = 1;
20      }
21    }
22
23    /* Update string with the last set of repeated characters. */
24    index = setChar(array, last, index, count);
25    return String.valueOf(array);
26  }
27
28  int setChar(char[] array, char c, int index, int count) {
29    array[index] = c;
30    index++;
31
32    /* Convert the count to a string, then to an array of chars */
33    char[] cnt = String.valueOf(count).toCharArray();
```

```
34
35      /* Copy characters from biggest digit to smallest */
36      for (char x : cnt) {
37          array[index] = x;
38          index++;
39      }
40      return index;
41  }
42
43  int countCompression(String str) { /* same as earlier */ }
```

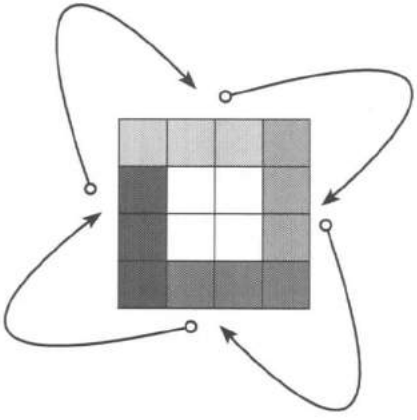
Like the second solution, the above code runs in O(N) time and O(N) space.

1.6 *Given an image represented by an NxN matrix, where each pixel in the image is 4 bytes, write a method to rotate the image by 90 degrees. Can you do this in place?*

pg 73

SOLUTION

Because we're rotating the matrix by 90 degrees, the easiest way to do this is to implement the rotation in layers. We perform a circular rotation on each layer, moving the top edge to the right edge, the right edge to the bottom edge, the bottom edge to the left edge, and the left edge to the top edge.



How do we perform this four-way edge swap? One option is to copy the top edge to an array, and then move the left to the top, the bottom to the left, and so on. This requires O(N) memory, which is actually unnecessary.

A better way to do this is to implement the swap index by index. In this case, we do the following:

```
1  for i = 0 to n
2      temp = top[i];
3      top[i] = left[i]
```

```
4       left[i] = bottom[i]
5       bottom[i] = right[i]
6       right[i] = temp
```

We perform such a swap on each layer, starting from the outermost layer and working our way inwards. (Alternatively, we could start from the inner layer and work outwards.)

The code for this algorithm is below.

```
1   public void rotate(int[][] matrix, int n) {
2     for (int layer = 0; layer < n / 2; ++layer) {
3       int first = layer;
4       int last = n - 1 - layer;
5       for(int i = first; i < last; ++i) {
6         int offset = i - first;
7         // save top
8         int top = matrix[first][i];
9
10        // left -> top
11        matrix[first][i] = matrix[last-offset][first];
12
13        // bottom -> left
14        matrix[last-offset][first] = matrix[last][last - offset];
15
16        // right -> bottom
17        matrix[last][last - offset] = matrix[i][last];
18
19        // top -> right
20        matrix[i][last] = top;
21      }
22    }
23  }
```

This algorithm is $O(N^2)$, which is the best we can do since any algorithm must touch all N^2 elements.

1.7 *Write an algorithm such that if an element in an MxN matrix is 0, its entire row and column are set to 0.*

pg 73

SOLUTION

At first glance, this problem seems easy: just iterate through the matrix and every time we see a cell with value zero, set its row and column to 0. There's one problem with that solution though: when we come across other cells in that row or column, we'll see the zeros and change their row and column to zero. Pretty soon, our entire matrix will be set to zeros.

One way around this is to keep a second matrix which flags the zero locations. We would then do a second pass through the matrix to set the zeros. This would take $O(MN)$ space.

Do we really need O(MN) space? No. Since we're going to set the entire row and column to zero, we don't need to track that it was exactly `cell[2][4]` (row 2, column 4). We only need to know that row 2 has a zero somewhere, and column 4 has a zero somewhere. We'll set the entire row and column to zero anyway, so why would we care to keep track of the exact location of the zero?

The code below implements this algorithm. We use two arrays to keep track of all the rows with zeros and all the columns with zeros. We then make a second pass of the matrix and set a cell to zero if its row or column is zero.

```
1  public void setZeros(int[][] matrix) {
2     boolean[] row = new boolean[matrix.length];
3     boolean[] column = new boolean[matrix[0].length];
4
5     // Store the row and column index with value 0
6     for (int i = 0; i < matrix.length; i++) {
7        for (int j = 0; j < matrix[0].length;j++) {
8           if (matrix[i][j] == 0) {
9              row[i] = true;
10             column[j] = true;
11          }
12       }
13    }
14
15    // Set arr[i][j] to 0 if either row i or column j has a 0
16    for (int i = 0; i < matrix.length; i++) {
17       for (int j = 0; j < matrix[0].length; j++) {
18          if (row[i] || column[j]) {
19             matrix[i][j] = 0;
20          }
21       }
22    }
23 }
```

To make this somewhat more space efficient, we could use a bit vector instead of a boolean array.

1.8 *Assume you have a method `isSubstring` which checks if one word is a substring of another. Given two strings, `s1` and `s2`, write code to check if `s2` is a rotation of `s1` using only one call to `isSubstring` (e.g., "`waterbottle`" is a rotation of "`erbottlewat`").*

pg 74

SOLUTION

If we imagine that `s2` is a rotation of `s1`, then we can ask what the rotation point is. For example, if you rotate `waterbottle` after `wat`, you get `erbottlewat`. In a rotation, we cut `s1` into two parts, `x` and `y`, and rearrange them to get `s2`.

```
s1 = xy = waterbottle
x = wat
y = erbottle
s2 = yx = erbottlewat
```

So, we need to check if there's a way to split s1 into x and y such that xy = s1 and yx = s2. Regardless of where the division between x and y is, we can see that yx will always be a substring of xyxy. That is, s2 will always be a substring of s1s1.

And this is precisely how we solve the problem: simply do `isSubstring(s1s1, s2)`.

The code below implements this algorithm.

```
1  public boolean isRotation(String s1, String s2) {
2     int len = s1.length();
3     /* check that s1 and s2 are equal length and not empty */
4     if (len == s2.length() && len > 0) {
5        /* concatenate s1 and s1 within new buffer */
6        String s1s1 = s1 + s1;
7        return isSubstring(s1s1, s2);
8     }
9     return false;
10 }
```

Linked Lists

Data Structures: Solutions **Chapter 2**

2.1 *Write code to remove duplicates from an unsorted linked list.*

FOLLOW UP

How would you solve this problem if a temporary buffer is not allowed?

pg 77

SOLUTION

In order to remove duplicates from a linked list, we need to be able to track duplicates. A simple hash table will work well here.

In the below solution, we simply iterate through the linked list, adding each element to a hash table. When we discover a duplicate element, we remove the element and continue iterating. We can do this all in one pass since we are using a linked list.

```
1  public static void deleteDups(LinkedListNode n) {
2     Hashtable table = new Hashtable();
3     LinkedListNode previous = null;
4     while (n != null) {
5        if (table.containsKey(n.data)) {
6           previous.next = n.next;
7        } else {
8           table.put(n.data, true);
9           previous = n;
10       }
11       n = n.next;
12    }
13 }
```

The above solution takes O(N) time, where N is the number of elements in the linked list.

Follow Up: No Buffer Allowed

If we don't have a buffer, we can iterate with two pointers: current which iterates through the linked list, and runner which checks all subsequent nodes for duplicates.

```
1  public static void deleteDups(LinkedListNode head) {
2     if (head == null) return;
3
4     LinkedListNode current = head;
5     while (current != null) {
6        /* Remove all future nodes that have the same value */
7        LinkedListNode runner = current;
8        while (runner.next != null) {
9           if (runner.next.data == current.data) {
10             runner.next = runner.next.next;
11          } else {
12             runner = runner.next;
13          }
14       }
```

```
15          current = current.next;
16      }
17  }
```

This code runs in O(1) space, but $O(N^2)$ time.

2.2 *Implement an algorithm to find the kth to last element of a singly linked list.*

pg 77

SOLUTION

We will approach this problem both recursively and non-recursively. Remember that recursive solutions are often cleaner but less optimal. For example, in this problem, the recursive implementation is about half the length of the iterative solution but also takes O(n) space, where n is the number of elements in the linked list.

Note that for this solution, we have defined k such that passing in `k = 1` would return the last element, `k = 2` would return to the second to last element, and so on. It is equally acceptable to define k such that k = 0 would return the last element.

Solution #1: If linked list size is known

If the size of the linked list is known, then the kth to last element is the `(length - k)` th element. We can just iterate through the linked list to find this element. Because this solution is so trivial, we can almost be sure that this is not what the interviewer intended.

Solution #2: Recursive

This algorithm recurses through the linked list. When it hits the end, the method passes back a counter set to 0. Each parent call adds 1 to this counter. When the counter equals k, we know we have reached the kth to last element of the linked list.

Implementing this is short and sweet—provided we have a way of "passing back" an integer value through the stack. Unfortunately, we can't pass back a node and a counter using normal return statements. So how do we handle this?

Approach A: Don't Return the Element.

One way to do this is to change the problem to simply printing the kth to last element. Then, we can pass back the value of the counter simply through return values.

```
1  public static int nthToLast(LinkedListNode head, int k) {
2    if (head == null) {
3      return 0;
4    }
5    int i = nthToLast(head.next, k) + 1;
6    if (i == k) {
7      System.out.println(head.data);
8    }
```

```
9      return i;
10 }
```

Of course, this is only a valid solution if the interviewer says it is valid.

Approach B: Use C++.

A second way to solve this is to use C++ and to pass values by reference. This allows us to return the node value, but also update the counter by passing a pointer to it.

```
1  node* nthToLast(node* head, int k, int& i) {
2    if (head == NULL) {
3      return NULL;
4    }
5    node * nd = nthToLast(head->next, k, i);
6    i = i + 1;
7    if (i == k) {
8      return head;
9    }
10   return nd;
11 }
```

Approach C: Create a Wrapper Class.

We described earlier that the issue was that we couldn't simultaneously return a counter and an index. If we wrap the `counter` value with simple class (or even a single element array), we can mimic passing by reference.

```
1  public class IntWrapper {
2    public int value = 0;
3  }
4
5  LinkedListNode nthToLastR2(LinkedListNode head, int k,
6                             IntWrapper i) {
7    if (head == null) {
8      return null;
9    }
10   LinkedListNode node = nthToLastR2(head.next, k, i);
11   i.value = i.value + 1;
12   if (i.value == k) { // We've found the kth element
13     return head;
14   }
15   return node;
16 }
17
```

Each of these recursive solutions takes `O(n)` space due to the recursive calls.

There are a number of other solutions that we haven't addressed. We could store the counter in a static variable. Or, we could create a class that stores both the node and the counter, and return an instance of that class. Regardless of which solution we pick, we need a way to update both the node and the counter in a way that all levels of the recursive stack will see.

Solution #3: Iterative

A more optimal, but less straightforward, solution is to implement this iteratively. We can use two pointers, p1 and p2. We place them k nodes apart in the linked list by putting p1 at the beginning and moving p2 k nodes into the list. Then, when we move them at the same pace, p2 will hit the end of the linked list after LENGTH - k steps. At that point, p1 will be LENGTH - k nodes into the list, or k nodes from the end.

The code below implements this algorithm.

```
1  LinkedListNode nthToLast(LinkedListNode head, int k) {
2     if (k <= 0) return null;
3
4     LinkedListNode p1 = head;
5     LinkedListNode p2 = head;
6
7     // Move p2 forward k nodes into the list.
8     for (int i = 0; i < k - 1; i++) {
9        if (p2 == null) return null; // Error check
10       p2 = p2.next;
11    }
12    if (p2 == null) return null;
13
14    /* Now, move p1 and p2 at the same speed. When p2 hits the end,
15     * p1 will be at the right element. */
16    while (p2.next != null) {
17       p1 = p1.next;
18       p2 = p2.next;
19    }
20    return p1;
21 }
```

This algorithm takes O(n) time and O(1) space.

2.3 *Implement an algorithm to delete a node in the middle of a singly linked list, given only access to that node.*

pg 77

SOLUTION

In this problem, you are not given access to the head of the linked list. You only have access to that node. The solution is simply to copy the data from the next node over to the current node, and then to delete the next node.

The code below implements this algorithm.

```
1  public static boolean deleteNode(LinkedListNode n) {
2     if (n == null || n.next == null) {
3        return false; // Failure
4     }
```

```
    LinkedListNode next = n.next;
    n.data = next.data;
    n.next = next.next;
    return true;
}
```

Note that this problem cannot be solved if the node to be deleted is the last node in the linked list. That's ok—your interviewer wants you to point that out, and to discuss how to handle this case. You could, for example, consider marking the node as dummy.

2.4 *Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x.*

pg 77

SOLUTION

If this were an array, we would need to be careful about how we shifted elements. Array shifts are very expensive.

However, in a linked list, the situation is much easier. Rather than shifting and swapping elements, we can actually create two different linked lists: one for elements less than x, and one for elements greater than or equal to x.

We iterate through the linked list, inserting elements into our `before` list or our `after` list. Once we reach the end of the linked list and have completed this splitting, we merge the two lists.

The code below implements this approach.

```
/* Pass in the head of the linked list and the value to partition
 * around */
public LinkedListNode partition(LinkedListNode node, int x) {
   LinkedListNode beforeStart = null;
   LinkedListNode beforeEnd = null;
   LinkedListNode afterStart = null;
   LinkedListNode afterEnd = null;

   /* Partition list */
   while (node != null) {
      LinkedListNode next = node.next;
      node.next = null;
      if (node.data < x) {
         /* Insert node into end of before list */
         if (beforeStart == null) {
            beforeStart = node;
            beforeEnd = beforeStart;
         } else {
            beforeEnd.next = node;
            beforeEnd = node;
```

```
21          }
22        } else {
23          /* Insert node into end of after list */
24          if (afterStart == null) {
25            afterStart = node;
26            afterEnd = afterStart;
27          } else {
28            afterEnd.next = node;
29            afterEnd = node;
30          }
31        }
32        node = next;
33      }
34
35      if (beforeStart == null) {
36        return afterStart;
37      }
38
39      /* Merge before list and after list */
40      beforeEnd.next = afterStart;
41      return beforeStart;
42  }
```

If it bugs you to keep around four different variables for tracking two linked lists, you're not alone. We can get rid of some of these, with just a minor hit to the efficiency. This drop in efficiency comes because we have to traverse the linked list an extra time. The big-O time will remain the same though, and we get shorter, cleaner code.

The second solution operates in a slightly different way. Instead of inserting nodes into the end of the before list and the after list, it inserts nodes into the front of them.

```
1   public LinkedListNode partition(LinkedListNode node, int x) {
2     LinkedListNode beforeStart = null;
3     LinkedListNode afterStart = null;
4
5     /* Partition list */
6     while (node != null) {
7       LinkedListNode next = node.next;
8       if (node.data < x) {
9         /* Insert node into start of before list */
10        node.next = beforeStart;
11        beforeStart = node;
12      } else {
13        /* Insert node into front of after list */
14        node.next = afterStart;
15        afterStart = node;
16      }
17      node = next;
18    }
19
20    /* Merge before list and after list */
```

```
21    if (beforeStart == null) {
22      return afterStart;
23    }
24
25    /* Find end of before list, and merge the lists */
26    LinkedListNode head = beforeStart;
27    while (beforeStart.next != null) {
28      beforeStart = beforeStart.next;
29    }
30    beforeStart.next = afterStart;
31
32    return head;
33  }
```

Note that in this problem, we need to be very careful about null values. Check out line 7 in the above solution. The line is here because we are modifying the linked list as we're looping through it. We need to store the next node in a temporary variable so that we remember which node should be next in our iteration.

2.5 *You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in reverse order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.*

FOLLOW UP

Suppose the digits are stored in forward order. Repeat the above problem.

pg 77

SOLUTION

It's useful to remember in this problem how exactly addition works. Imagine the problem:

```
    6 1 7
  + 2 9 5
```

First, we add 7 and 5 to get 12. The digit 2 becomes the last digit of the number, and 1 gets carried over to the next step. Second, we add 1, 1, and 9 to get 11. The 1 becomes the second digit, and the other 1 gets carried over the final step. Third and finally, we add 1, 6 and 2 to get 9. So, our value becomes 912.

We can mimic this process recursively by adding node by node, carrying over any "excess" data to the next node. Let's walk through this for the below linked list:

```
    7 -> 1 -> 6
+   5 -> 9 -> 2
```

We do the following:

1. We add 7 and 5 first, getting a result of 12. 2 becomes the first node in our linked list,

and we "carry" the 1 to the next sum.

```
List: 2 -> ?
```

2. We then add 1 and 9, as well as the "carry," getting a result of 11. 1 becomes the second element of our linked list, and we carry the 1 to the next sum.

```
List: 2 -> 1 -> ?
```

3. Finally, we add 6, 2 and our "carry," to get 9. This become the final element of our linked list.

```
List: 2 -> 1 -> 9.
```

The code below implements this algorithm.

```
1  LinkedListNode addLists(LinkedListNode l1, LinkedListNode l2,
2                          int carry) {
3     /* We're done if both lists are null AND the carry value is 0 */
4     if (l1 == null && l2 == null && carry == 0) {
5        return null;
6     }
7
8     LinkedListNode result = new LinkedListNode(carry, null, null);
9
10    /* Add value, and the data from l1 and l2 */
11    int value = carry;
12    if (l1 != null) {
13       value += l1.data;
14    }
15    if (l2 != null) {
16       value += l2.data;
17    }
18
19    result.data = value % 10; /* Second digit of number */
20
21    /* Recurse */
22    if (l1 != null || l2 != null) {
23       LinkedListNode more = addLists(l1 == null ? null : l1.next,
24                                      l2 == null ? null : l2.next,
25                                      value >= 10 ? 1 : 0);
26       result.setNext(more);
27    }
28    return result;
29 }
```

In implementing this code, we must be careful to handle the condition when one linked list is shorter than another. We don't want to get a null pointer exception.

Follow Up

Part B is conceptually the same (recurse, carry the excess), but has some additional complications when it comes to implementation:

1. One list may be shorter than the other, and we cannot handle this "on the fly." For example, suppose we were adding (1 -> 2 -> 3 -> 4) and (5 -> 6 -> 7). We need to know that the 5 should be "matched" with the 2, not the 1. We can accomplish this by comparing the lengths of the lists in the beginning and padding the shorter list with zeros.

2. In the first part, successive results were added to the tail (i.e., passed forward). This meant that the recursive call would be *passed* the carry, and would return the result (which is then appended to the tail). In this case, however, results are added to the head (i.e., passed backward). The recursive call must return the result, as before, as well as the carry. This is not terribly challenging to implement, but it is more cumbersome. We can solve this issue by creating a wrapper class called Partial Sum.

The code below implements this algorithm.

```
1  public class PartialSum {
2     public LinkedListNode sum = null;
3     public int carry = 0;
4  }
5
6  LinkedListNode addLists(LinkedListNode l1, LinkedListNode l2) {
7     int len1 = length(l1);
8     int len2 = length(l2);
9
10    /* Pad the shorter list with zeros - see note (1) */
11    if (len1 < len2) {
12       l1 = padList(l1, len2 - len1);
13    } else {
14       l2 = padList(l2, len1 - len2);
15    }
16
17    /* Add lists */
18    PartialSum sum = addListsHelper(l1, l2);
19
20    /* If there was a carry value left over, insert this at the
21     * front of the list. Otherwise, just return the linked list. */
22    if (sum.carry == 0) {
23       return sum.sum;
24    } else {
25       LinkedListNode result = insertBefore(sum.sum, sum.carry);
26       return result;
27    }
28 }
29
30 PartialSum addListsHelper(LinkedListNode l1, LinkedListNode l2) {
31    if (l1 == null && l2 == null) {
32       PartialSum sum = new PartialSum();
33       return sum;
34    }
35    /* Add smaller digits recursively */
```

```
36    PartialSum sum = addListsHelper(l1.next, l2.next);
37
38    /* Add carry to current data */
39    int val = sum.carry + l1.data + l2.data;
40
41    /* Insert sum of current digits */
42    LinkedListNode full_result = insertBefore(sum.sum, val % 10);
43
44    /* Return sum so far, and the carry value */
45    sum.sum = full_result;
46    sum.carry = val / 10;
47    return sum;
48 }
49
50 /* Pad the list with zeros */
51 LinkedListNode padList(LinkedListNode l, int padding) {
52    LinkedListNode head = l;
53    for (int i = 0; i < padding; i++) {
54       LinkedListNode n = new LinkedListNode(0, null, null);
55       head.prev = n;
56       n.next = head;
57       head = n;
58    }
59    return head;
60 }
61
62 /* Helper function to insert node in the front of a linked list */
63 LinkedListNode insertBefore(LinkedListNode list, int data) {
64    LinkedListNode node = new LinkedListNode(data, null, null);
65    if (list != null) {
66       list.prev = node;
67       node.next = list;
68    }
69    return node;
70 }
```

Note how we have pulled `insertBefore()`, `padList()`, and `length()` (not listed) into their own methods. This makes the code cleaner and easier to read—a wise thing to do in your interviews!

2.6 *Given a circular linked list, implement an algorithm which returns the node at the beginning of the loop.*

pg 78

SOLUTION

This is a modification of a classic interview problem: detect if a linked list has a loop. Let's apply the Pattern Matching approach.

Part 1: Detect If Linked List Has A Loop

An easy way to detect if a linked list has a loop is through the `FastRunner` / `SlowRunner` approach. `FastRunner` moves two steps at a time, while `SlowRunner` moves one step. Much like two cars racing around a track at different steps, they must eventually meet.

An astute reader may wonder if `FastRunner` might "hop over" `SlowRunner` completely, without ever colliding. That's not possible. Suppose that `FastRunner` *did* hop over `SlowRunner`, such that SlowRunner is at spot `i` and `FastRunner` is at spot `i + 1`. In the previous step, SlowRunner would be at spot `i - 1` and FastRunner would at spot `((i + 1) - 2)`, or spot `i - 1`. That is, they would have collided.

Part 2: When Do They Collide?

Let's assume that the linked list has a "non-looped" part of size k.

If we apply our algorithm from part 1, when will `FastRunner` and `SlowRunner` collide?

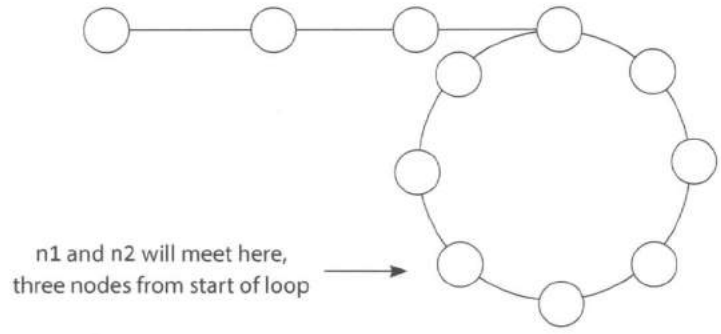
We know that for every p steps that `SlowRunner` takes, `FastRunner` has taken 2p steps. Therefore, when `SlowRunner` enters the looped portion after k steps, `FastRunner` has taken 2k steps total and must be `2k - k` steps, or k steps, into the looped portion. Since k might be much larger than the loop length, we should actually write this as `mod(k, LOOP_SIZE)` steps, which we will denote as K.

At each subsequent step, `FastRunner` and `SlowRunner` get either one step farther away or one step closer, depending on your perspective. That is, because we are in a circle, when A moves q steps away from B, it is also moving q steps closer to B.

So now we know the following facts:

1. `SlowRunner` is 0 steps into the loop.
2. `FastRunner` is K steps into the loop.
3. `SlowRunner` is K steps behind `FastRunner`.
4. `FastRunner` is `LOOP_SIZE - K` steps behind `SlowRunner`.
5. `FastRunner` catches up to `SlowRunner` at a rate of 1 step per unit of time.

So, when do they meet? Well, if `FastRunner` is `LOOP_SIZE - K` steps behind `SlowRunner`, and `FastRunner` catches up at a rate of 1 step per unit of time, then they meet after `LOOP_SIZE - K` steps. At this point, they will be K steps before the head of the loop. Let's call this point `CollisionSpot`.



Part 3: How Do You Find The Start of the Loop?

We now know that `CollisionSpot` is K nodes before the start of the loop. Because K = `mod(k, LOOP_SIZE)` (or, in other words, k = K + M * `LOOP_SIZE`, for any integer M), it is also correct to say that it is k nodes from the loop start. For example, if node N is 2 nodes into a 5 node loop, it is also correct to say that it is 7, 12, or even 397 nodes into the loop.

Therefore, both `CollisionSpot` and `LinkedListHead` are k nodes from the start of the loop.

Now, if we keep one pointer at `CollisionSpot` and move the other one to `LinkedListHead`, they will each be k nodes from `LoopStart`. Moving the two pointers at the same speed will cause them to collide again—this time after k steps, at which point they will both be at `LoopStart`. All we have to do is return this node.

Part 4: Putting It All Together

To summarize, we move `FastPointer` twice as fast as `SlowPointer`. When `SlowPointer` enters the loop, after k nodes, `FastPointer` is k nodes into the linked list. This means that `FastPointer` and `SlowPointer` are `LOOP_SIZE - k` nodes away from each other.

Next, if `FastPointer` moves two nodes for each node that `SlowPointer` moves, they move one node closer to each other on each turn. Therefore, they will meet after `LOOP_SIZE - k` turns. Both will be k nodes from the front of the loop.

The head of the linked list is also k nodes from the front of the loop. So, if we keep one pointer where it is, and move the other pointer to the head of the linked list, then they will meet at the front of the loop.

Our algorithm is derived directly from parts 1, 2 and 3.

1. Create two pointers, `FastPointer` and `SlowPointer`.
2. Move `FastPointer` at a rate of 2 steps and `SlowPointer` at a rate of 1 step.
3. When they collide, move `SlowPointer` to `LinkedListHead`. Keep `FastPointer`

where it is.

4. Move SlowPointer and FastPointer at a rate of one step. Return the new collision point.

The code below implements this algorithm.

```
1  LinkedListNode FindBeginning(LinkedListNode head) {
2     LinkedListNode slow = head;
3     LinkedListNode fast = head;
4
5     /* Find meeting point. This will be LOOP_SIZE - k steps into the
6      * linked list. */
7     while (fast != null && fast.next != null) {
8        slow = slow.next;
9        fast = fast.next.next;
10       if (slow == fast) { // Collision
11          break;
12       }
13    }
14
15    /* Error check - no meeting point, and therefore no loop */
16    if (fast == null || fast.next == null) {
17       return null;
18    }
19
20    /* Move slow to Head. Keep fast at Meeting Point. Each are k
21     * steps from the Loop Start. If they move at the same pace,
22     * they must meet at Loop Start. */
23    slow = head;
24    while (slow != fast) {
25       slow = slow.next;
26       fast = fast.next;
27    }
28
29    /* Both now point to the start of the loop. */
30    return fast;
31 }
```

2.7 *Implement a function to check if a linked list is a palindrome.*

pg 78

SOLUTION

To approach this problem, we can picture a palindrome like `0 -> 1 -> 2 -> 1 -> 0`. We know that, since it's a palindrome, the list must be the same backwards and forwards. This leads us to our first solution.

Solution #1: Reverse and Compare

Our first solution is to reverse the linked list and compare the reversed list to the original list. If they're the same, the lists are identical.

Note that when we compare the linked list to the reversed list, we only actually need to compare the first half of the list. If the first half of the normal list matches the first half of the reversed list, then the second half of the normal list must match the second half of the reversed list.

Solution #2: Iterative Approach

We want to detect linked lists where the front half of the list is the reverse of the second half. How would we do that? By reversing the front half of the list. A stack can accomplish this.

We need to push the first half of the elements onto a stack. We can do this in two different ways, depending on whether or not we know the size of the linked list.

If we know the size of the linked list, we can iterate through the first half of the elements in a standard for loop, pushing each element onto a stack. We must be careful, of course, to handle the case where the length of the linked list is odd.

If we don't know the size of the linked list, we can iterate through the linked list, using the fast runner / slow runner technique described in the beginning of the chapter. At each step in the loop, we push the data from the slow runner onto a stack. When the fast runner hits the end of the list, the slow runner will have reached the middle of the linked list. By this point, the stack will have all the elements from the front of the linked list, but in reverse order.

Now, we simply iterate through the rest of the linked list. At each iteration, we compare the node to the top of the stack. If we complete the iteration without finding a difference, then the linked list is a palindrome.

```
1  boolean isPalindrome(LinkedListNode head) {
2    LinkedListNode fast = head;
3    LinkedListNode slow = head;
4
5    Stack<Integer> stack = new Stack<Integer>();
6
7    /* Push elements from first half of linked list onto stack. When
8     * fast runner (which is moving at 2x speed) reaches the end of
9     * the linked list, then we know we're at the middle */
10   while (fast != null && fast.next != null) {
11     stack.push(slow.data);
12     slow = slow.next;
13     fast = fast.next.next;
14   }
15
16   /* Has odd number of elements, so skip the middle element */
17   if (fast != null) {
```

```
18        slow = slow.next;
19      }
20
21      while (slow != null) {
22        int top = stack.pop().intValue();
23
24        /* If values are different, then it's not a palindrome */
25        if (top != slow.data) {
26          return false;
27        }
28        slow = slow.next;
29      }
30      return true;
31  }
```

Solution #3: Recursive Approach

First, a word on notation: in the below solution, when we use the notation node Kx, the variable K indicates the value of the node data, and x (which is either f or b) indicates whether we are referring to the front node with that value or the back node. For example, in the below linked list, node 3b would refer to the second (back) node with value 3.

Now, like many linked list problems, you can approach this problem recursively. We may have some intuitive idea that we want to compare element 0 and element n, element 1 and element n-1, element 2 and element n-2, and so on, until the middle element(s). For example:

```
0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0
```

In order to apply this approach, we first need to know when we've reached the middle element, as this will form our base case. We can do this by passing in `length - 2` for the length each time. When the length equals 0 or 1, we're at the center of the linked list.

```
1  recurse(Node n, int length) {
2    if (length == 0 || length == 1) {
3      return [something]; // At middle
4    }
5    recurse(n.next, length - 2);
6    ...
7  }
```

This method will form the outline of the `isPalindrome` method. The "meat" of the algorithm though is comparing node `i` to node `n - i` to check if the linked list is a palindrome. How do we do that?

Let's examine what the call stack looks like:

```
1  v1 = isPalindrome: list = 0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0. length = 7
2    v2 = isPalindrome: list = 1 ( 2 ( 3 ) 2 ) 1 ) 0. length = 5
3      v3 = isPalindrome: list = 2 ( 3 ) 2 ) 1 ) 0. length = 3
```

```
4          v4 = isPalindrome: list = 3 ) 2 ) 1 ) 0. length = 1
5          returns v3
6        returns v2
7      returns v1
8    returns ?
```

In the above call stack, each call wants to check if the list is a palindrome by comparing its head node with the corresponding node from the back of the list. That is:

- Line 1 needs to compare node `0f` with node `0b`
- Line 2 needs to compare node `1f` with node `1b`
- Line 3 needs to compare node `2f` with node `2b`
- Line 4 needs to compare node `3f` with node `3b`.

If we rewind the stack, passing nodes back as described below, we can do just that:

- Line 4 sees that it is the middle node (since `length = 1`), and passes back `head.next`. The value head equals node `3`, so `head.next` is node `2b`.
- Line 3 compares its head, node `2f`, to `returned_node` (the value from the previous recursive call), which is node `2b`. If the values match, it passes a reference to node `1b` (`returned_node.next`) up to line 2.
- Line 2 compares its head (node `1f`) to `returned_node` (node `1b`). If the values match, it passes a reference to node `0b` (or, `returned_node.next`) up to line 1.
- Line 1 compares its head, node `0f`, to `returned_node`, which is node `0b`. If the values match, it returns true.

To generalize, each call compares its head to `returned_node`, and then passes `returned_node.next` up the stack. In this way, every node `i` gets compared to node `n - i`. If at any point the values do not match, we return `false`, and every call up the stack checks for that value.

But wait, you might ask, sometimes we said we'll return a `boolean` value, and sometimes we're returning a node. Which is it?

It's both. We create a simple class with two members, a `boolean` and a node, and return an instance of that class.

```
1  class Result {
2    public LinkedListNode node;
3    public boolean result;
4  }
```

The example below illustrates the parameters and return values from this sample list.

```
1  isPalindrome: list = 0 ( 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 9
2    isPalindrome: list = 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 7
3      isPalindrome: list = 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 5
4        isPalindrome: list = 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 3
```

```
            isPalindrome: list = 4 ) 3 ) 2 ) 1 ) 0. len = 1
            returns node 3b, true
         returns node 2b, true
      returns node 1b, true
   returns node 0b, true
returns nobe 0b, true
```

Implementing this code is now just a matter of filling in the details.

```
Result isPalindromeRecurse(LinkedListNode head, int length) {
   if (head == null || length == 0) {
      return new Result(null, true);
   } else if (length == 1) {
      return new Result(head.next, true);
   } else if (length == 2) {
      return new Result(head.next.next,
                        head.data == head.next.data);
   }
   Result res = isPalindromeRecurse(head.next, length - 2);
   if (!res.result || res.node == null) {
      return res;
   } else {
      res.result = head.data == res.node.data;
      res.node = res.node.next;
      return res;
   }
}

boolean isPalindrome(LinkedListNode head) {
   Result p = isPalindromeRecurse(head, listSize(head));
   return p.result;
}
```

Some of you might be wondering why we went through all this effort to create a special `Result` class. Isn't there a better way? Not really—at least not in Java.

However, if we were implementing this in C or C++, we could have passed in a double pointer.

```
bool isPalindromeRecurse(Node head, int length, Node** next) {
   ...
}
```

It's ugly, but it works.

Stacks and Queues

Data Structures: Solutions **Chapter 3**

3.1 *Describe how you could use a single array to implement three stacks.*

pg 80

SOLUTION

Like many problems, this one somewhat depends on how well we'd like to support these stacks. If we're ok with simply allocating a fixed amount of space for each stack, we can do that. This may mean though that one stack runs out of space, while the others are nearly empty.

Alternatively, we can be flexible in our space allocation, but this significantly increases the complexity of the problem.

Approach 1: Fixed Division

We can divide the array in three equal parts and allow the individual stack to grow in that limited space. Note: we will use the notation "[" to mean inclusive of an end point and "(" to mean exclusive of an end point.

- For stack 1, we will use `[0, n/3)`.
- For stack 2, we will use `[n/3, 2n/3)`.
- For stack 3, we will use `[2n/3, n)`.

The code for this solution is below.

```
1  int stackSize = 100;
2  int[] buffer = new int [stackSize * 3];
3  int[] stackPointer = {-1, -1, -1}; // pointers to track top element
4
5  void push(int stackNum, int value) throws Exception {
6     /* Check if we have space */
7     if (stackPointer[stackNum] + 1 >= stackSize) { // Last element
8        throw new Exception("Out of space.");
9     }
10    /* Increment stack pointer and then update top value */
11    stackPointer[stackNum]++;
12    buffer[absTopOfStack(stackNum)] = value;
13 }
14
15 int pop(int stackNum) throws Exception {
16    if (stackPointer[stackNum] == -1) {
17       throw new Exception("Trying to pop an empty stack.");
18    }
19    int value = buffer[absTopOfStack(stackNum)]; // Get top
20    buffer[absTopOfStack(stackNum)] = 0; // Clear index
21    stackPointer[stackNum]--; // Decrement pointer
22    return value;
23 }
24
```

```
25 int peek(int stackNum) {
26    int index = absTopOfStack(stackNum);
27    return buffer[index];
28 }
29
30 boolean isEmpty(int stackNum) {
31    return stackPointer[stackNum] == -1;
32 }
33
34 /* returns index of top of stack "stackNum", in absolute terms */
35 int absTopOfStack(int stackNum) {
36    return stackNum * stackSize + stackPointer[stackNum];
37 }
```

If we had additional information about the expected usages of the stacks, then we could modify this algorithm accordingly. For example, if we expected Stack 1 to have many more elements than Stack 2, we could allocate more space to Stack 1 and less space to Stack 2.

Approach 2: Flexible Divisions

A second approach is to allow the stack blocks to be flexible in size. When one stack exceeds its initial capacity, we grow the allowable capacity and shift elements as necessary.

We will also design our array to be circular, such that the final stack may start at the end of the array and wrap around to the beginning.

Please note that the code for this solution is far more complex than would be appropriate for an interview. You could be responsible for pseudocode, or perhaps the code of individual components, but the entire implementation would be far too challenging.

```
1  /* StackData is a simple class that holds a set of data about each
2   * stack. It does not hold the actual items in the stack. */
3  public class StackData {
4     public int start;
5     public int pointer;
6     public int size = 0;
7     public int capacity;
8     public StackData(int _start, int _capacity) {
9        start = _start;
10       pointer = _start - 1;
11       capacity = _capacity;
12    }
13
14    public boolean isWithinStack(int index, int total_size) {
15       /* Note: if stack wraps, the head (right side) wraps around
16        * to the left. */
17       if (start <= index && index < start + capacity) {
18          // non-wrapping, or "head" (right side) of wrapping case
```

```
19          return true;
20        } else if (start + capacity > total_size &&
21                   index < (start + capacity) % total_size) {
22          // tail (left side) of wrapping case
23          return true;
24        }
25        return false;
26     }
27  }
28
29  public class QuestionB {
30     static int number_of_stacks = 3;
31     static int default_size = 4;
32     static int total_size = default_size * number_of_stacks;
33     static StackData [] stacks = {new StackData(0, default_size),
34        new StackData(default_size, default_size),
35        new StackData(default_size * 2, default_size)};
36     static int [] buffer = new int [total_size];
37
38     public static void main(String [] args) throws Exception  {
39        push(0, 10);
40        push(1, 20);
41        push(2, 30);
42        int v = pop(0);
43        ...
44     }
45
46     public static int numberOfElements() {
47        return stacks[0].size + stacks[1].size + stacks[2].size;
48     }
49
50     public static int nextElement(int index) {
51        if (index + 1 == total_size) return 0;
52        else return index + 1;
53     }
54
55     public static int previousElement(int index) {
56        if (index == 0) return total_size - 1;
57        else return index - 1;
58     }
59
60     public static void shift(int stackNum) {
61        StackData stack = stacks[stackNum];
62        if (stack.size >= stack.capacity) {
63           int nextStack = (stackNum + 1) % number_of_stacks;
64           shift(nextStack); // make some room
65           stack.capacity++;
66        }
67
68        // Shift elements in reverse order
```

```
69      for (int i = (stack.start + stack.capacity - 1) % total_size;
70           stack.isWithinStack(i, total_size);
71           i = previousElement(i)) {
72         buffer[i] = buffer[previousElement(i)];
73      }
74
75      buffer[stack.start] = 0;
76      stack.start = nextElement(stack.start); // move stack start
77      stack.pointer = nextElement(stack.pointer); // move pointer
78      stack.capacity--; // return capacity to original
79   }
80
81   /* Expand stack by shifting over other stacks */
82   public static void expand(int stackNum) {
83      shift((stackNum + 1) % number_of_stacks);
84      stacks[stackNum].capacity++;
85   }
86
87   public static void push(int stackNum, int value)
88         throws Exception {
89      StackData stack = stacks[stackNum];
90      /* Check that we have space */
91      if (stack.size >= stack.capacity) {
92         if (numberOfElements() >= total_size) { // Totally full
93            throw new Exception("Out of space.");
94         } else { // just need to shift things around
95            expand(stackNum);
96         }
97      }
98      /* Find the index of the top element in the array + 1,
99       * and increment the stack pointer */
100     stack.size++;
101     stack.pointer = nextElement(stack.pointer);
102     buffer[stack.pointer] = value;
103  }
104
105  public static int pop(int stackNum) throws Exception {
106     StackData stack = stacks[stackNum];
107     if (stack.size == 0) {
108        throw new Exception("Trying to pop an empty stack.");
109     }
110     int value = buffer[stack.pointer];
111     buffer[stack.pointer] = 0;
112     stack.pointer = previousElement(stack.pointer);
113     stack.size--;
114     return value;
115  }
116
117  public static int peek(int stackNum) {
118     StackData stack = stacks[stackNum];
```

```
119       return buffer[stack.pointer];
120    }
121
122    public static boolean isEmpty(int stackNum) {
123       StackData stack = stacks[stackNum];
124       return stack.size == 0;
125    }
126 }
```

In problems like this, it's important to focus on writing clean, maintainable code. You should use additional classes, as we did with `StackData`, and pull chunks of code into separate methods. Of course, this advice applies to the "real world" as well.

3.2 *How would you design a stack which, in addition to push and pop, also has a function `min` which returns the minimum element? Push, pop and `min` should all operate in O(1) time.*

pg 80

SOLUTION

The thing with minimums is that they don't change very often. They only change when a smaller element is added.

One solution is to have just a single `int` value, `minValue`, that's a member of the `Stack` class. When `minValue` is popped from the stack, we search through the stack to find the new minimum. Unfortunately, this would break the constraint that push and pop operate in O(1) time.

To further understand this question, let's walk through it with a short example:

```
push(5); // stack is {5}, min is 5
push(6); // stack is {6, 5}, min is 5
push(3); // stack is {3, 6, 5}, min is 3
push(7); // stack is {7, 3, 6, 5}, min is 3
pop(); // pops 7. stack is {3, 6, 5}, min is 3
pop(); // pops 3. stack is {6, 5}. min is 5.
```

Observe how once the stack goes back to a prior state (`{6, 5}`), the minimum also goes back to its prior state (5). This leads us to our second solution.

If we kept track of the minimum at each state, we would be able to easily know the minimum. We can do this by having each node record what the minimum beneath itself is. Then, to find the `min`, you just look at what the top element thinks is the min.

When you push an element onto the stack, the element is given the current minimum. It sets its "`local min`" to be the `min`.

```
1  public class StackWithMin extends Stack<NodeWithMin> {
2     public void push(int value) {
3        int newMin = Math.min(value, min());
```

```
4         super.push(new NodeWithMin(value, newMin));
5      }
6
7      public int min() {
8         if (this.isEmpty()) {
9            return Integer.MAX_VALUE; // Error value
10        } else {
11           return peek().min;
12        }
13     }
14  }
15
16  class NodeWithMin {
17     public int value;
18     public int min;
19     public NodeWithMin(int v, int min){
20        value = v;
21        this.min = min;
22     }
23  }
```

There's just one issue with this: if we have a large stack, we waste a lot of space by keeping track of the `min` for every single element. Can we do better?

We can (maybe) do a bit better than this by using an additional stack which keeps track of the `mins`.

```
1   public class StackWithMin2 extends Stack<Integer> {
2      Stack<Integer> s2;
3      public StackWithMin2() {
4         s2 = new Stack<Integer>();
5      }
6
7      public void push(int value){
8         if (value <= min()) {
9            s2.push(value);
10        }
11        super.push(value);
12     }
13
14     public Integer pop() {
15        int value = super.pop();
16        if (value == min()) {
17           s2.pop();
18        }
19        return value;
20     }
21
22     public int min() {
23        if (s2.isEmpty()) {
24           return Integer.MAX_VALUE;
```

```
25        } else {
26            return s2.peek();
27        }
28    }
29 }
```

Why might this be more space efficient? Suppose we had a very large stack and the first element inserted happened to be the minimum. In the first solution, we would be keeping n ints, where n is the size of the stack. In the second solution though, we store just a few pieces of data: a second stack with one element and the members within this stack.

3.3 *Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure* `SetOfStacks` *that mimics this.* `SetOfStacks` *should be composed of several stacks and should create a new stack once the previous one exceeds capacity.* `SetOfStacks.push()` *and* `SetOfStacks.pop()` *should behave identically to a single stack (that is,* `pop()` *should return the same values as it would if there were just a single stack).*

FOLLOW UP

Implement a function `popAt(int index)` *which performs a pop operation on a specific sub-stack.*

pg 80

SOLUTION

In this problem, we've been told what our data structure should look like:

```
1  class SetOfStacks {
2     ArrayList<Stack> stacks = new ArrayList<Stack>();
3     public void push(int v) { ... }
4     public int pop() { ... }
5  }
```

We know that `push()` should behave identically to a single stack, which means that we need `push()` to call `push()` on the last stack in the array of stacks. We have to be a bit careful here though: if the last stack is at capacity, we need to create a new stack. Our code should look something like this:

```
1  public void push(int v) {
2     Stack last = getLastStack();
3     if (last != null && !last.isFull()) { // add to last stack
4        last.push(v);
5     } else { // must create new stack
6        Stack stack = new Stack(capacity);
7        stack.push(v);
8        stacks.add(stack);
9     }
```

```
10 }
```

What should pop() do? It should behave similarly to push() in that it should operate on the last stack. If the last stack is empty (after popping), then we should remove the stack from the list of stacks.

```
1  public int pop() {
2     Stack last = getLastStack();
3     int v = last.pop();
4     if (last.size == 0) stacks.remove(stacks.size() - 1);
5     return v;
6  }
```

Follow Up: Implement popAt(int index)

This is a bit trickier to implement, but we can imagine a "rollover" system. If we pop an element from stack 1, we need to remove the *bottom* of stack 2 and push it onto stack 1. We then need to rollover from stack 3 to stack 2, stack 4 to stack 3, etc.

You could make an argument that, rather than "rolling over," we should be OK with some stacks not being at full capacity. This would improve the time complexity (by a fair amount, with a large number of elements), but it might get us into tricky situations later on if someone assumes that all stacks (other than the last) operate at full capacity. There's no "right answer" here; you should discuss this trade-off with your interviewer.

```
1  public class SetOfStacks {
2     ArrayList<Stack> stacks = new ArrayList<Stack>();
3     public int capacity;
4     public SetOfStacks(int capacity) {
5        this.capacity = capacity;
6     }
7
8     public Stack getLastStack() {
9        if (stacks.size() == 0) return null;
10       return stacks.get(stacks.size() - 1);
11    }
12
13    public void push(int v) { /* see earlier code */ }
14    public int pop() { /* see earlier code */ }
15    public boolean isEmpty() {
16       Stack last = getLastStack();
17       return last == null || last.isEmpty();
18    }
19
20    public int popAt(int index) {
21       return leftShift(index, true);
22    }
23
24    public int leftShift(int index, boolean removeTop) {
25       Stack stack = stacks.get(index);
26       int removed_item;
```

```
27        if (removeTop) removed_item = stack.pop();
28        else removed_item = stack.removeBottom();
29        if (stack.isEmpty()) {
30           stacks.remove(index);
31        } else if (stacks.size() > index + 1) {
32           int v = leftShift(index + 1, false);
33           stack.push(v);
34        }
35        return removed_item;
36     }
37  }
38
39  public class Stack {
40     private int capacity;
41     public Node top, bottom;
42     public int size = 0;
43
44     public Stack(int capacity) { this.capacity = capacity; }
45     public boolean isFull() { return capacity == size; }
46
47     public void join(Node above, Node below) {
48        if (below != null) below.above = above;
49        if (above != null) above.below = below;
50     }
51
52     public boolean push(int v) {
53        if (size >= capacity) return false;
54        size++;
55        Node n = new Node(v);
56        if (size == 1) bottom = n;
57        join(n, top);
58        top = n;
59        return true;
60     }
61
62     public int pop() {
63        Node t = top;
64        top = top.below;
65        size--;
66        return t.value;
67     }
68
69     public boolean isEmpty() {
70        return size == 0;
71     }
72
73     public int removeBottom() {
74        Node b = bottom;
75        bottom = bottom.above;
76        if (bottom != null) bottom.below = null;
```

```
77          size--;
78          return b.value;
79      }
80  }
```

This problem is not conceptually that tough, but it requires a lot of code to implement it fully. Your interviewer would not ask you to implement the entire code.

A good strategy on problems like this is to separate code into other methods, like a `leftShift` method that popAt can call. This will make your code cleaner and give you the opportunity to lay down the skeleton of the code before dealing with some of the details.

3.4 *In the classic problem of the Towers of Hanoi, you have 3 towers and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (i.e., each disk sits on top of an even larger one). You have the following constraints:*

(1) Only one disk can be moved at a time.

(2) A disk is slid off the top of one tower onto the next rod.

(3) A disk can only be placed on top of a larger disk.

Write a program to move the disks from the first tower to the last using Stacks.

pg 81

SOLUTION

This problem sounds like a good candidate for the Base Case and Build approach.

Let's start with the smallest possible example: n = 1.

Case n = 1. Can we move Disk 1 from Tower 1 to Tower 3? Yes.

1. We simply move Disk 1 from Tower 1 to Tower 3.

Case n = 2. Can we move Disk 1 and Disk 2 from Tower 1 to Tower 3? Yes.

1. Move Disk 1 from Tower 1 to Tower 2
2. Move Disk 2 from Tower 1 to Tower 3
3. Move Disk 1 from Tower 2 to Tower 3

Note how in the above steps, Tower 2 acts as a buffer, holding a disk while we move

other disks to Tower 3.

Case n = 3. Can we move Disk 1, 2, and 3 from Tower 1 to Tower 3? Yes.

1. We know we can move the top two disks from one tower to another (as shown earlier), so let's assume we've already done that. But instead, let's move them to Tower 2.
2. Move Disk 3 to Tower 3.
3. Move Disk 1 and Disk 2 to Tower 3. We already know how to do this—we just repeat what we did in Step 1.

Case n = 4. Can we move Disk 1, 2, 3 and 4 from Tower 1 to Tower 3? Yes.

1. Move Disks 1, 2, and 3 to Tower 2. We know how to do that from the earlier examples.
2. Move Disk 4 to Tower 3.
3. Move Disks 1, 2 and 3 back to Tower 3.

Remember that the labels of Tower 2 and Tower 3 aren't important. They're equivalent towers. So, moving disks to Tower 3 with Tower 2 serving as a buffer is equivalent to moving disks to Tower 2 with Tower 3 serving as a buffer.

This approach leads to a natural recursive algorithm. In each part, we are doing the following steps, outlined below with pseudocode:

```
1  moveDisks(int n, Tower origin, Tower destination, Tower buffer) {
2     /* Base case */
3     if (n <= 0) return;
4
5     /* move top n - 1 disks from origin to buffer, using destination
6      * as a buffer. */
7     moveDisks(n - 1, origin, buffer, destination);
8
9     /* move top from origin to destination
10    moveTop(origin, destination);
11
12    /* move top n - 1 disks from buffer to destination, using
13     * origin as a buffer. */
14    moveDisks(n - 1, buffer, destination, origin);
15 }
```

The following code provides a more detailed implementation of this algorithm, using concepts of object-oriented design.

```
1  public static void main(String[] args) {
2     int n = 3;
3     Tower[] towers = new Tower[n];
4     for (int i = 0; i < 3; i++) {
5        towers[i] = new Tower(i);
6     }
7
```

```
8     for (int i = n - 1; i >= 0; i--) {
9        towers[0].add(i);
10    }
11    towers[0].moveDisks(n, towers[2], towers[1]);
12 }
13
14 public class Tower {
15    private Stack<Integer> disks;
16    private int index;
17    public Tower(int i) {
18       disks = new Stack<Integer>();
19       index = i;
20    }
21
22    public int index() {
23       return index;
24    }
25
26    public void add(int d) {
27       if (!disks.isEmpty() && disks.peek() <= d) {
28          System.out.println("Error placing disk " + d);
29       } else {
30          disks.push(d);
31       }
32    }
33
34    public void moveTopTo(Tower t) {
35       int top = disks.pop();
36       t.add(top);
37       System.out.println("Move disk " + top + " from " + index() +
38                          " to " + t.index());
39    }
40
41    public void moveDisks(int n, Tower destination, Tower buffer) {
42       if (n > 0) {
43          moveDisks(n - 1, buffer, destination);
44          moveTopTo(destination);
45          buffer.moveDisks(n - 1, destination, this);
46       }
47    }
48 }
```

Implementing the towers as their own object is not strictly necessary, but it does help to make the code cleaner in some respects.

3.5 *Implement a MyQueue class which implements a queue using two stacks.*

pg 81

SOLUTION

Since the major difference between a queue and a stack is the order (first-in first-out vs. last-in first-out), we know that we need to modify peek( ) and pop( ) to go in reverse order. We can use our second stack to reverse the order of the elements (by popping s1 and pushing the elements on to s2). In such an implementation, on each peek( ) and pop( ) operation, we would pop everything from s1 onto s2, perform the peek / pop operation, and then push everything back.

This will work, but if two pop / peeks are performed back-to-back, we're needlessly moving elements. We can implement a "lazy" approach where we let the elements sit in s2 until we absolutely must reverse the elements.

In this approach, stackNewest has the newest elements on top and stackOldest has the oldest elements on top. When we dequeue an element, we want to remove the oldest element first, and so we dequeue from stackOldest. If stackOldest is empty, then we want to transfer all elements from stackNewest into this stack in reverse order. To insert an element, we push onto stackNewest, since it has the newest elements on top.

The code below implements this algorithm.

```
1  public class MyQueue<T> {
2    Stack<T> stackNewest, stackOldest;
3
4    public MyQueue() {
5      stackNewest = new Stack<T>();
6      stackOldest = new Stack<T>();
7    }
8
9    public int size() {
10     return stackNewest.size() + stackOldest.size();
11   }
12
13   public void add(T value) {
14     /* Push onto stackNewest, which always has the newest
15      * elements on top */
16     stackNewest.push(value);
17   }
18
19   /* Move elements from stackNewest into stackOldest. This is
20    * usually done so that we can do operations on stackOldest. */
21   private void shiftStacks() {
22     if (stackOldest.isEmpty()) {
23       while (!stackNewest.isEmpty()) {
24         stackOldest.push(stackNewest.pop());
25       }
26     }
27   }
```

```
28
29    public T peek() {
30       shiftStacks(); // Ensure stackOldest has the current elements
31       return stackOldest.peek(); // retrieve the oldest item.
32    }
33
34    public T remove() {
35       shiftStacks(); // Ensure stackOldest has the current elements
36       return stackOldest.pop(); // pop the oldest item.
37    }
38 }
```

During your actual interview, you may find that you forget the exact API calls. Don't stress too much if that happens to you. Most interviewers are okay with your asking for them to refresh your memory on little details. They're much more concerned with your big picture understanding.

3.6 *Write a program to sort a stack in ascending order (with biggest items on top). You may use at most one additional stack to hold items, but you may not copy the elements into any other data structure (such as an array). The stack supports the following operations: push, pop, peek, and isEmpty.*

pg 81

SOLUTION

One approach is to implement a rudimentary sorting algorithm. We search through the entire stack to find the minimum element and then push that onto a new stack. Then, we find the new minimum element and push that. This will actually require a total of three stacks: s1 is the original stack, s2 is the final sorted stack, and s3 acts as a buffer during our searching of s1. To search s1 for each minimum, we need to pop elements from s1 and push them onto the buffer, s3.

Unfortunately, we're only allowed one additional stack. Can we do better? Yes.

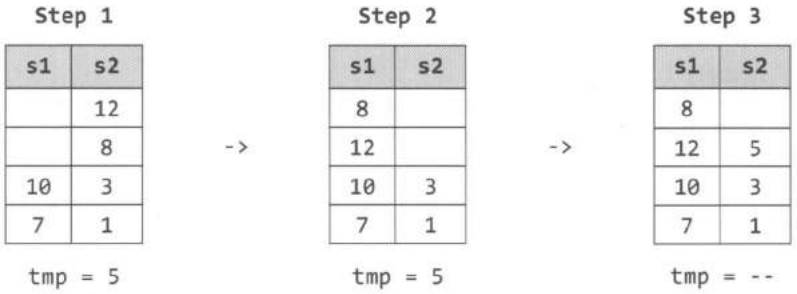
Rather than searching for the minimum repeatedly, we can sort s1 by inserting each element from s1 in order into s2. How would this work?

Imagine we have the following stacks, where s2 is "sorted" and s1 is not:

s1	s2
	12
5	8
10	3
7	1

When we pop 5 from s1, we need to find the right place in s2 to insert this number. In this case, the correct place is on s2 just above 3. How do we get it there? We can do this

by popping 5 from s1 and holding it in a temporary variable. Then, we move 12 and 8 over to s1 (by popping them from s2 and pushing them onto s1) and then push 5 onto s2.



Note that 8 and 12 are still in s1 -- and that's okay! We just repeat the same steps for those two numbers as we did for 5, each time popping off the top of s1 and putting it into the "right place" on s2. (Of course, since 8 and 12 were moved from s2 to s1 precisely *because* they were larger than 5, the "right place" for these elements will be right on top of 5. We won't need to muck around with s2's other elements, and the inside of the below `while` loop will not be run when `tmp` is 8 or 12.)

```
1  public static Stack<Integer> sort(Stack<Integer> s) {
2    Stack<Integer> r = new Stack<Integer>();
3    while (!s.isEmpty()) {
4      int tmp = s.pop(); // Step 1
5      while (!r.isEmpty() && r.peek() > tmp) { // Step 2
6        s.push(r.pop());
7      }
8      r.push(tmp); // Step 3
9    }
10   return r;
11 }
```

This algorithm is $O(N^2)$ time and $O(N)$ space.

If we were allowed to use unlimited stacks, we could implement a modified quicksort or mergesort.

With the mergesort solution, we would create two extra stacks and divide the stack into two parts. We would recursively sort each stack, and then merge them back together in sorted order into the original stack. Note that this would require the creation of two additional stacks per level of recursion.

With the quicksort solution, we would create two additional stacks and divide the stack into the two stacks based on a pivot element. The two stacks would be recursively sorted, and then merged back together into the original stack. Like the earlier solution, this one involves creating two additional stacks per level of recursion.

3.7 *An animal shelter holds only dogs and cats, and operates on a strictly "first in, first out" basis. People must adopt either the "oldest" (based on arrival time) of all animals at the shelter, or they can select whether they would prefer a dog or a cat (and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structures to maintain this system and implement operations such as enqueue, dequeueAny, dequeueDog and dequeueCat. You may use the built-in `LinkedList` data structure.*

pg 81

SOLUTION

We could explore a variety of solutions to this problem. For instance, we could maintain a single queue. This would make dequeueAny easy, but dequeueDog and dequeueCat would require iteration through the queue to find the first dog or cat. This would increase the complexity of the solution and decrease the efficiency.

An alternative approach that is simple, clean and efficient is to simply use separate queues for dogs and cats, and to place them within a wrapper class called `AnimalQueue`. We then store some sort of timestamp to mark when each animal was enqueued. When we call dequeueAny, we peek at the heads of both the dog and `cat` queue and return the oldest.

```
1  public abstract class Animal {
2     private int order;
3     protected String name;
4     public Animal(String n) {
5        name = n;
6     }
7
8     public void setOrder(int ord) {
9        order = ord;
10    }
11
12    public int getOrder() {
13       return order;
14    }
15
16    public boolean isOlderThan(Animal a) {
17       return this.order < a.getOrder();
18    }
19 }
20
21 public class AnimalQueue {
22    LinkedList<Dog> dogs = new LinkedList<Dog>();
23    LinkedList<Cat> cats = new LinkedList<Cat>();
24    private int order = 0; // acts as timestamp
25
26    public void enqueue(Animal a) {
```

```
27        /* Order is used as a sort of timestamp, so that we can
28         * compare the insertion order of a dog to a cat. */
29        a.setOrder(order);
30        order++;
31
32        if (a instanceof Dog) dogs.addLast((Dog) a);
33        else if (a instanceof Cat) cats.addLast((Cat)a);
34     }
35
36     public Animal dequeueAny() {
37        /* Look at tops of dog and cat queues, and pop the stack
38         * with the oldest value. */
39        if (dogs.size() == 0) {
40           return dequeueCats();
41        } else if (cats.size() == 0) {
42           return dequeueDogs();
43        }
44
45        Dog dog = dogs.peek();
46        Cat cat = cats.peek();
47     if (dog.isOlderThan(cat)) {
48        return dequeueDogs();
49     } else {
50        return dequeueCats();
51     }
52
53     public Dog dequeueDogs() {
54        return dogs.poll();
55     }
56
57     public Cat dequeueCats() {
58        return cats.poll();
59     }
60  }
61
62  public class Dog extends Animal {
63     public Dog(String n) {
64        super(n);
65     }
66  }
67
68  public class Cat extends Animal {
69     public Cat(String n) {
70        super(n);
71     }
72  }
```

Trees and Graphs

Data Structures: Solutions **Chapter 4**

4.1 *Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.*

pg 86

SOLUTION

In this question, we've been fortunate enough to be told exactly what balanced means: that for each node, the two subtrees differ in height by no more than one. We can implement a solution based on this definition. We can simply recurse through the entire tree, and for each node, compute the heights of each subtree.

```
1  public static int getHeight(TreeNode root) {
2     if (root == null) return 0; // Base case
3     return Math.max(getHeight(root.left),
4                     getHeight(root.right)) + 1;
5  }
6
7  public static boolean isBalanced(TreeNode root) {
8     if (root == null) return true; // Base case
9
10    int heightDiff = getHeight(root.left) - getHeight(root.right);
11    if (Math.abs(heightDiff) > 1) {
12       return false;
13    } else { // Recurse
14       return isBalanced(root.left) && isBalanced(root.right);
15    }
16 }
```

Although this works, it's not very efficient. On each node, we recurse through its entire subtree. This means that `getHeight` is called repeatedly on the same nodes. The algorithm is therefore $O(N^2)$.

We need to cut out some of the calls to `getHeight`.

If we inspect this method, we may notice that `getHeight` could actually check if the tree is balanced as the same time as it's checking heights. What do we do when we discover that the subtree isn't balanced? Just return -1.

This improved algorithm works by checking the height of each subtree as we recurse down from the root. On each node, we recursively get the heights of the left and right subtrees through the `checkHeight` method. If the subtree is balanced, then `checkHeight` will return the actual height of the subtree. If the subtree is not balanced, then `checkHeight` will return -1. We will immediately break and return -1 from the current call.

The code below implements this algorithm.

```
1  public static int checkHeight(TreeNode root) {
2     if (root == null) {
```

```
3         return 0; // Height of 0
4      }
5
6      /* Check if left is balanced. */
7      int leftHeight = checkHeight(root.left);
8      if (leftHeight == -1) {
9         return -1; // Not balanced
10     }
11     /* Check if right is balanced. */
12     int rightHeight = checkHeight(root.right);
13     if (rightHeight == -1) {
14        return -1; // Not balanced
15     }
16
17     /* Check if current node is balanced. */
18     int heightDiff = leftHeight - rightHeight;
19     if (Math.abs(heightDiff) > 1) {
20        return -1; // Not balanced
21     } else {
22        /* Return height */
23        return Math.max(leftHeight, rightHeight) + 1;
24     }
25  }
26
27  public static boolean isBalanced(TreeNode root) {
28     if (checkHeight(root) == -1) {
29        return false;
30     } else {
31        return true;
32     }
33  }
```

This code runs in O(N) time and O(H) space, where H is the height of the tree.

4.2 *Given a directed graph, design an algorithm to find out whether there is a route between two nodes.*

pg 86

SOLUTION

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.

The code below provides an iterative implementation of breadth first search.

```
1  public enum State {
2     Unvisited, Visited, Visiting;
```

```
}

public static boolean search(Graph g, Node start, Node end) {
   // operates as Queue
   LinkedList<Node> q = new LinkedList<Node>();

   for (Node u : g.getNodes()) {
      u.state = State.Unvisited;
   }
   start.state = State.Visiting;
   q.add(start);
   Node u;
   while (!q.isEmpty()) {
      u = q.removeFirst(); // i.e., dequeue()
      if (u != null) {
         for (Node v : u.getAdjacent()) {
            if (v.state == State.Unvisited) {
               if (v == end) {
                  return true;
               } else {
                  v.state = State.Visiting;
                  q.add(v);
               }
            }
         }
         u.state = State.Visited;
      }
   }
   return false;
}
```

It may be worth discussing with your interviewer the trade-offs between breadth first search and depth first search for this and other problems. For example, 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.

4.3 *Given a sorted (increasing order) array with unique integer elements, write an algorithm to create a binary search tree with minimal height.*

pg 86

SOLUTION

To create a tree of minimal height, we need to match the number of nodes in the left subtree to the number of nodes in the right subtree as much as possible. This means that we want the root to be the middle of the array, since this would mean that half the elements would be less than the root and half would be greater than it.

We proceed with constructing our tree in a similar fashion. The middle of each subsection of the array becomes the root of the node. The left half of the array will become our left subtree, and the right half of the array will become the right subtree.

One way to implement this is to use a simple `root.insertNode(int v)` method which inserts the value v through a recursive process that starts with the root node. This will indeed construct a tree with minimal height but it will not do so very efficiently. Each insertion will require traversing the tree, giving a total cost of `O(N log N)` to the tree.

Alternatively, we can cut out the extra traversals by recursively using the `createMinimalBST` method. This method is passed just a subsection of the array and returns the root of a minimal tree for that array.

The algorithm is as follows:

1. Insert into the tree the middle element of the array.
2. Insert (into the left subtree) the left subarray elements.
3. Insert (into the right subtree) the right subarray elements.
4. Recurse.

The code below implements this algorithm.

```
1  TreeNode createMinimalBST(int arr[], int start, int end) {
2    if (end < start) {
3      return null;
4    }
5    int mid = (start + end) / 2;
6    TreeNode n = new TreeNode(arr[mid]);
7    n.left = createMinimalBST(arr, start, mid - 1);
8    n.right = createMinimalBST(arr, mid + 1, end);
9    return n;
10 }
11
12 TreeNode createMinimalBST(int array[]) {
13   return createMinimalBST(array, 0, array.length - 1);
14 }
```

Although this code does not seem especially complex, it can be very easy to make little off-by-one errors. Be sure to test these parts of the code very thoroughly.

4.4 *Given a binary tree, design an algorithm which creates a linked list of all the nodes at each depth (e.g., if you have a tree with depth D, you'll have D linked lists).*

pg 86

SOLUTION

Though we might think at first glance that this problem requires a level-by-level traversal, this isn't actually necessary. We can traverse the graph any way that we'd like, provided we know which level we're on as we do so.

We can implement a simple modification of the pre-order traversal algorithm, where we pass in `level + 1` to the next recursive call. The code below provides an implementation using depth first search.

```
1  void createLevelLinkedList(TreeNode root,
2        ArrayList<LinkedList<TreeNode>> lists, int level) {
3     if (root == null) return; // base case
4
5     LinkedList<TreeNode> list = null;
6     if (lists.size() == level) { // Level not contained in list
7        list = new LinkedList<TreeNode>();
8        /* Levels are always traversed in order. So, if this is the
9         * first time we've visited level i, we must have seen levels
10        * 0 through i - 1. We can therefore safely add the level at
11        * the end. */
12       lists.add(list);
13    } else {
14       list = lists.get(level);
15    }
16    list.add(root);
17    createLevelLinkedList(root.left, lists, level + 1);
18    createLevelLinkedList(root.right, lists, level + 1);
19 }
20
21 ArrayList<LinkedList<TreeNode>> createLevelLinkedList(
22       TreeNode root) {
23    ArrayList<LinkedList<TreeNode>> lists =
24       new ArrayList<LinkedList<TreeNode>>();
25    createLevelLinkedList(root, lists, 0);
26    return lists;
27 }
```

Alternatively, we can also implement a modification of breadth first search. With this implementation, we want to iterate through the root first, then level 2, then level 3, and so on.

With each level `i`, we will have already fully visited all nodes on level `i - 1`. This means that to get which nodes are on level `i`, we can simply look at all children of the nodes of level `i - 1`.

The code below implements this algorithm.

```
1  ArrayList<LinkedList<TreeNode>> createLevelLinkedList(
2        TreeNode root) {
3     ArrayList<LinkedList<TreeNode>> result =
4        new ArrayList<LinkedList<TreeNode>>();
5     /* "Visit" the root */
6     LinkedList<TreeNode> current = new LinkedList<TreeNode>();
7     if (root != null) {
8        current.add(root);
9     }
10
11    while (current.size() > 0) {
12       result.add(current); // Add previous level
13       LinkedList<TreeNode> parents = current; // Go to next level
14       current = new LinkedList<TreeNode>();
15       for (TreeNode parent : parents) {
16          /* Visit the children */
17          if (parent.left != null) {
18             current.add(parent.left);
19          }
20          if (parent.right != null) {
21             current.add(parent.right);
22          }
23       }
24    }
25    return result;
26 }
```

One might ask which of these solutions is more efficient. Both run in `O(N)` time, but what about the space efficiency? At first, we might want to claim that the second solution is more space efficient.

In a sense, that's correct. The first solution uses `O(log N)` recursive calls, each of which adds a new level to the stack. The second solution, which is iterative, does not require this extra space.

However, both solutions require returning `O(N)` data. The extra `O(log N)` space usage from the recursive implementation is dwarfed by the `O(N)` data that must be returned. So while the first solution may actually use more data, they are equally efficient when it comes to "big O."

4.5 *Implement a function to check if a binary tree is a binary search tree.*

pg 86

SOLUTION

We can implement this solution in two different ways. The first leverages the in-order traversal, and the second builds off the property that `left <= current < right`.

Solution #1: In-Order Traversal

Our first thought might be to do an in-order traversal, copy the elements to an array, and then check to see if the array is sorted. This solution takes up a bit of extra memory, but it works -- mostly.

The only problem is that it can't handle duplicate values in the tree properly. For example, the algorithm cannot distinguish between the two trees below (one of which is invalid) since they have the same in-order traversal.

However, if we assume that the tree cannot have duplicate values, then this approach works. The pseudocode for this method looks something like:

```
1  public static int index = 0;
2  public static void copyBST(TreeNode root, int[] array) {
3     if (root == null) return;
4     copyBST(root.left, array);
5     array[index] = root.data;
6     index++;
7     copyBST(root.right, array);
8  }
9
10 public static boolean checkBST(TreeNode root) {
11    int[] array = new int[root.size];
12    copyBST(root, array);
13    for (int i = 1; i < array.length; i++) {
14       if (array[i] <= array[i - 1]) return false;
15    }
16    return true;
17 }
```

Note that it is necessary to keep track of the logical "end" of the array, since it would be allocated to hold all the elements.

When we examine this solution, we find that the array is not actually necessary. We never use it other than to compare an element to the previous element. So why not just track the last element we saw and compare it as we go?

The code below implements this algorithm.

```
1  public static int last_printed = Integer.MIN_VALUE;
2  public static boolean checkBST(TreeNode n) {
3     if (n == null) return true;
```

```
4
5        // Check / recurse left
6        if (!checkBST(n.left)) return false;
7
8        // Check current
9        if (n.data <= last_printed) return false;
10       last_printed = n.data;
11
12       // Check / recurse right
13       if (!checkBST(n.right)) return false;
14
15       return true; // All good!
16    }
```

If you don't like the use of static variables, then you can tweak this code to use a wrapper class for the integer, as shown below.

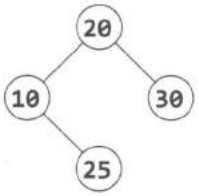
```
1  class WrapInt {
2     public int value;
3  }
```

Or, if you're implementing this in C++ or another language that supports passing integers by reference, then you can simply do that.

Solution #2: The Min / Max Solution

In the second solution, we leverage the definition of the binary search tree.

What does it mean for a tree to be a binary search tree? We know that it must, of course, satisfy the condition `left.data <= current.data < right.data` for each node, but this isn't quite sufficient. Consider the following small tree:

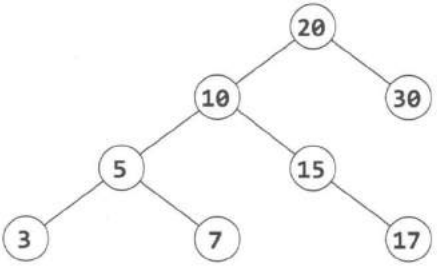


Although each node is bigger than its left node and smaller than its right node, this is clearly not a binary search tree since 25 is in the wrong place.

More precisely, the condition is that *all* left nodes must be less than or equal to the current node, which must be less than all the right nodes.

Using this thought, we can approach the problem by passing down the min and max values. As we iterate through the tree, we verify against progressively narrower ranges.

Consider the following sample tree:



We start with a range of (`min = INT_MIN, max = INT_MAX`), which the root obviously meets. We then branch left, checking that these nodes are within the range (`min = INT_MIN, max = 20`). Then, we branch right, checking that the nodes are within the range (`min = 10, max = 20`).

We proceed through the tree with this approach. When we branch left, the max gets updated. When we branch right, the min gets updated. If anything fails these checks, we stop and return false.

The time complexity for this solution is `O(N)`, where `N` is the number of nodes in the tree. We can prove that this is the best we can do, since any algorithm must touch all `N` nodes.

Due to the use of recursion, the space complexity is `O(log N)` on a balanced tree. There are up to `O(log N)` recursive calls on the stack since we may recurse up to the depth of the tree.

The recursive code for this is as follows:

```
1  boolean checkBST(TreeNode n) {
2     return checkBST(n, Integer.MIN_VALUE, Integer.MAX_VALUE);
3  }
4
5  boolean checkBST(TreeNode n, int min, int max) {
6     if (n == null) {
7        return true;
8     }
9     if (n.data <= min || n.data > max) {
10       return false;
11    }
12
13    if (!checkBST(n.left, min, n.data) ||
14        !checkBST(n.right, n.data, max)) {
15       return false;
16    }
17    return true;
18 }
```

Remember that in recursive algorithms, you should always make sure that your base cases, as well as your null cases, are well handled.

4.6 *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.*

pg 86

SOLUTION

Recall that an in-order traversal traverses the left subtree, then the current node, then the right subtree. To approach this problem, we need to think very, very carefully about what happens.

Let's suppose we have a hypothetical node. We know that the order goes left subtree, then current side, then right subtree. So, the next node we visit should be on the right side.

But which node on the right subtree? It should be the first node we'd visit if we were doing an in-order traversal of that subtree. This means that it should be the leftmost node on the right subtree. Easy enough!

But what if the node doesn't have a right subtree? This is where it gets a bit trickier.

If a node n doesn't have a right subtree, then we are done traversing n's subtree. We need to pick up where we left off with n's parent, which we'll call q.

If n was to the left of q, then the next node we should traverse should be q (again, since `left -> current -> right`).

If n were to the right of q, then we have fully traversed q's subtree as well. We need to traverse upwards from q until we find a node x that we have *not* fully traversed. How do we know that we have not fully traversed a node x? We know we have hit this case when we move from a left node to its parent. The left node is fully traversed, but its parent is not.

The pseudocode looks like this:

```
1  Node inorderSucc(Node n) {
2     if (n has a right subtree) {
3        return leftmost child of right subtree
4     } else {
5        while (n is a right child of n.parent) {
6           n = n.parent; // Go up
7        }
8        return n.parent; // Parent has not been traversed
9     }
10 }
```

But wait—what if we traverse all the way up the tree before finding a left child? This will happen only when we've hit the very end of the in-order traversal. That is, if we're *already* on the far right of the tree, then there is no in-order successor. We should return `null`.

The code below implements this algorithm (and properly handles the null case).

```
1  public TreeNode inorderSucc(TreeNode n) {
2     if (n == null) return null;
3
4     /* Found right children -> return leftmost node of right
5      * subtree. */
6     if (n.right != null) {
7        return leftMostChild(n.right);
8     } else {
9        TreeNode q = n;
10       TreeNode x = q.parent;
11       // Go up until we're on left instead of right
12       while (x != null && x.left != q) {
13          q = x;
14          x = x.parent;
15       }
16       return x;
17    }
18 }
19
20 public TreeNode leftMostChild(TreeNode n) {
21    if (n == null) {
22       return null;
23    }
24    while (n.left != null) {
25       n = n.left;
26    }
27    return n;
28 }
```

This is not the most algorithmically complex problem in the world, but it can be tricky to code perfectly. In a problem like this, it's useful to sketch out pseudocode to carefully outline the different cases.

4.7 *Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.*

pg 86

SOLUTION

If this were a binary search tree, we could modify the `find` operation for the two nodes and see where the paths diverge. Unfortunately, this is not a binary search tree, so we must try other approaches.

Let's assume we're looking for the common ancestor of nodes p and q. One question to ask here is if the nodes of our tree have links to its parents.

Solution #1: With Links to Parents

If each node has a link to its parent, we could trace p and q's paths up until they intersect. However, this may violate some assumptions of the problem as it would require either (a) being able to mark nodes as `isVisited` or (b) being able to store some data in an additional data structure, such as a hash table.

Solution #2: Without Links to Parents

Alternatively, you could follow a chain in which p and q are on the same side. That is, if p and q are both on the left of the node, branch left to look for the common ancestor. If they are both on the right, branch right to look for the common ancestor. When p and q are no longer on the same side, you must have found the first common ancestor.

The code below implements this approach.

```
1  /* Returns true if p is a descendent of root */
2  boolean covers(TreeNode root, TreeNode p) {
3     if (root == null) return false;
4     if (root == p) return true;
5     return covers(root.left, p) || covers(root.right, p);
6  }
7
8  TreeNode commonAncestorHelper(TreeNode root, TreeNode p,
9                                TreeNode q) {
10    if (root == null) return null;
11    if (root == p || root == q) return root;
12
13    boolean is_p_on_left = covers(root.left, p);
14    boolean is_q_on_left = covers(root.left, q);
15
16    /* If p and q are on different sides, return root. */
17    if (is_p_on_left != is_q_on_left) return root;
18
19    /* Else, they are on the same side. Traverse this side. */
20    TreeNode child_side = is_p_on_left ? root.left : root.right;
21    return commonAncestorHelper(child_side, p, q);
22 }
23
24 TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
25    if (!covers(root, p) || !covers(root, q)) { // Error check
26       return null;
27    }
28    return commonAncestorHelper(root, p, q);
29 }
```

This algorithm runs in $O(n)$ time on a balanced tree. This is because `covers` is called on $2n$ nodes in the first call (n nodes for the left side, and n nodes for the right side). After that, the algorithm branches left or right, at which point `covers` will be called on $2n/2$ nodes, then $2n/4$, and so on. This results in a runtime of $O(n)$.

We know at this point that we cannot do better than that in terms of the asymptotic runtime since we need to potentially look at every node in the tree. However, we may be able to improve it by a constant multiple.

Solution #3: Optimized

Although the Solution #2 is optimal in its runtime, we may recognize that there is still some inefficiency in how it operates. Specifically, `covers` searches all nodes under `root` for p and q, including the nodes in each subtree (`root.left` and `root.right`). Then, it picks one of those subtrees and searches all of its nodes. Each subtree is searched over and over again.

We may recognize that we should only need to search the entire tree once to find p and q. We should then be able to "bubble up" the findings to earlier nodes in the stack. The basic logic is the same as the earlier solution.

We recurse through the entire tree with a function called `commonAncestor(TreeNode root, TreeNode p, TreeNode q)`. This function returns values as follows:

- Returns p, if `root`'s subtree includes p (and not q).
- Returns q, if `root`'s subtree includes q (and not p).
- Returns `null`, if neither p nor q are in `root`'s subtree.
- Else, returns the common ancestor of p and q.

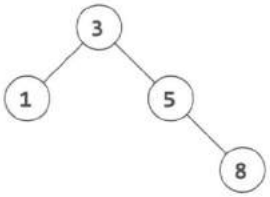
Finding the common ancestor of p and q in the final case is easy. When `commonAncestor(n.left, p, q)` and `commonAncestor(n.right, p, q)` both return non-null values (indicating that p and q were found in different subtrees), then n will be the common ancestor.

The code below offers an initial solution, but it has a bug. Can you find it?

```
1  /* The below code has a bug. */
2  TreeNode commonAncestorBad(TreeNode root, TreeNode p, TreeNode q) {
3     if (root == null) {
4        return null;
5     }
6     if (root == p && root == q) {
7        return root;
8     }
9
10    TreeNode x = commonAncestorBad(root.left, p, q);
11    if (x != null && x != p && x != q) { // Already found ancestor
12       return x;
13    }
14
15    TreeNode y = commonAncestorBad(root.right, p, q);
16    if (y != null && y != p && y != q) { // Already found ancestor
17       return y;
18    }
```

```
19
20    if (x != null && y != null) { // p and q found in diff. subtrees
21       return root; // This is the common ancestor
22    } else if (root == p || root == q) {
23       return root;
24    } else {
25       /* If either x or y is non-null, return the non-null value */
26       return x == null ? y : x;
27    }
28 }
```

The problem with this code occurs in the case where a node is not contained in the tree. For example, look at the tree below:



Suppose we call `commonAncestor(node 3, node 5, node 7)`. Of course, node 7 does not exist—and that's where the issue will come in. The calling order looks like:

```
1  commonAncestor(node 3, node 5, node 7)              // --> 5
2     calls commonAncestor(node 1, node 5, node 7)     // --> null
3     calls commonAncestor(node 5, node 5, node 7)     // --> 5
4        calls commonAncestor(node 8, node 5, node 7)  // --> null
```

In other words, when we call `commonAncestor` on the right subtree, the code will return node 5, just as it should. The problem is that, in finding the common ancestor of p and q, the calling function can't distinguish between the two cases:

- Case 1: p is a child of q (or, q is a child of p)
- Case 2: p is in the tree and q is not (or, q is in the tree and p is not)

In either of these cases, `commonAncestor` will return p. In the first case, this is the correct return value, but in the second case, the return value should be `null`.

We somehow need to distinguish between these two cases, and this is what the code below does. This code solves the problem by returning two values: the node itself and a flag indicating whether this node is actually the common ancestor.

```
1  public static class Result {
2     public TreeNode node;
3     public boolean isAncestor;
4     public Result(TreeNode n, boolean isAnc) {
5        node = n;
6        isAncestor = isAnc;
7     }
```

```
8  }
9
10 Result commonAncestorHelper(TreeNode root, TreeNode p, TreeNode q){
11    if (root == null) {
12       return new Result(null, false);
13    }
14    if (root == p && root == q) {
15       return new Result(root, true);
16    }
17
18    Result rx = commonAncestorHelper(root.left, p, q);
19    if (rx.isAncestor) { // Found common ancestor
20       return rx;
21    }
22
23    Result ry = commonAncestorHelper(root.right, p, q);
24    if (ry.isAncestor) { // Found common ancestor
25       return ry;
26    }
27
28    if (rx.node != null && ry.node != null) {
29       return new Result(root, true); // This is the common ancestor
30    } else if (root == p || root == q) {
31       /* If we're currently at p or q, and we also found one of
32        * those nodes in a subtree, then this is truly an ancestor
33        * and the flag should be true. */
34       boolean isAncestor = rx.node != null || ry.node != null ?
35                            true : false;
36       return new Result(root, isAncestor);
37    } else {
38       return new Result(rx.node!=null ? rx.node : ry.node, false);
39    }
40 }
41
42 TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
43    Result r = commonAncestorHelper(root, p, q);
44    if (r.isAncestor) {
45       return r.node;
46    }
47    return null;
48 }
```

Of course, as this issue only comes up in the case that p or q is not actually in the tree, an alternative solution would be to first search through the entire tree to make sure that both nodes exist.

4.8 *You have two very large binary trees: T1, with millions of nodes, and T2, with hundreds of nodes. Create an algorithm to decide if T2 is a subtree of T1.*

A tree T2 is a subtree of T1 if there exists a node n in T1 such that the subtree of n is identical to T2. That is, if you cut off the tree at node n, the two trees would be identical.

pg 86

SOLUTION

In problems like this, it's useful to attempt to solve the problem assuming that there is just a small amount of data. This will give us a basic idea of an approach that might work.

In this smaller, simpler problem, we could create a string representing the in-order and pre-order traversals. If T2's pre-order traversal is a substring of T1's pre-order traversal, and T2's in-order traversal is a substring of T1's in-order traversal, then T2 is a subtree of T1. Substrings can be checked with suffix trees in linear time, so this algorithm is relatively efficient in terms of the worst case time.

Note that we'll need to insert a special character into our strings to indicate when a left or right node is NULL. Otherwise, we would be unable to distinguish between the following cases:

These would have the same in-order and pre-order traversal, even though they are different trees.

```
T1, in-order:   3, 3
T1, pre-order:  3, 3
T2, in-order:   3, 3
T2, pre-order:  3, 3
```

However, if we mark the NULL values, we can distinguish between these two trees:

```
T1, in-order:   0, 3, 0, 3, 0
T1, pre-order:  3, 3, 0, 0, 0
T2, in-order:   0, 3, 0, 3, 0
T2, pre-order:  3, 0, 3, 0, 0
```

While this is a good solution for the simple case, our actual problem has much more data. Creating a copy of both trees may require too much memory given the constraints of the problem.

The Alternative Approach

An alternative approach is to search through the larger tree, T1. Each time a node in T1

matches the root of T2, call `treeMatch`. The `treeMatch` method will compare the two subtrees to see if they are identical.

Analyzing the runtime is somewhat complex. A naive answer would be to say that it is `O(nm)` time, where `n` is the number of nodes in `T1` and `m` is the number of nodes in `T2`. While this is technically correct, a little more thought can produce a tighter bound.

We do not actually call `treeMatch` on every node in `T2`. Rather, we call it k times, where k is the number of occurrences of `T2`'s root in `T1`. The runtime is closer to `O(n + km)`.

In fact, even that overstates the runtime. Even if the root were identical, we exit `treeMatch` when we find a difference between `T1` and `T2`. We therefore probably do not actually look at `m` nodes on each call of `treeMatch`.

The code below implements this algorithm.

```
1  boolean containsTree(TreeNode t1, TreeNode t2) {
2     if (t2 == null) { // The empty tree is always a subtree
3        return true;
4     }
5     return subTree(t1, t2);
6  }
7
8  boolean subTree(TreeNode r1, TreeNode r2) {
9     if (r1 == null) {
10       return false; // big tree empty & subtree still not found.
11    }
12    if (r1.data == r2.data) {
13       if (matchTree(r1,r2)) return true;
14    }
15    return (subTree(r1.left, r2) || subTree(r1.right, r2));
16 }
17
18 boolean matchTree(TreeNode r1, TreeNode r2) {
19    if (r2 == null && r1 == null) // if both are empty
20       return true; // nothing left in the subtree
21
22    // if one, but not both, are empty
23    if (r1 == null || r2 == null) {
24       return false;
25    }
26
27    if (r1.data != r2.data)
28       return false;  // data doesn't match
29    return (matchTree(r1.left, r2.left) &&
30         matchTree(r1.right, r2.right));
31    }
32 }
```

When might the simple solution be better, and when might the alternative approach be better? This is a great conversation to have with your interviewer. Here are a few

thoughts on that matter though:

1. The simple solution takes `O(n + m)` memory. The alternative solution takes `O(log(n) + log(m))` memory. Remember: memory usage can be a very big deal when it comes to scalability.
2. The simple solution is `O(n + m)` time and the alternative solution has a worst case time of `O(nm)`. However, the worst case time can be deceiving; we need to look deeper than that.
3. A slightly tighter bound on the runtime, as explained earlier, is `O(n + km)`, where `k` is the number of occurrences of `T2`'s root in `T1`. Let's suppose the node data for `T1` and `T2` were random numbers picked between 0 and p. The value of k would be approximately `n/p`. Why? Because each of n nodes in `T1` has a `1/p` chance of equaling the root, so approximately `n/p` nodes in `T1` should equal `T2.root`. So, let's say `p = 1000`, `n = 1000000` and `m = 100`. We would do somewhere around 1,100,000 node checks (`1100000 = 1000000 + 100*1000000/1000`).
4. More complex mathematics and assumptions could get us an even tighter bound. We assumed in #3 above that if we call `treeMatch`, we will end up traversing all `m` nodes of `T2`. It's far more likely though that we will find a difference very early on in the tree and will then exit early.

In summary, the alternative approach is certainly more optimal in terms of space and is likely more optimal in terms of time as well. It all depends on what assumptions you make and whether you prioritize reducing the average case runtime at the expense of the worst case runtime. This is an excellent point to make to your interviewer.

4.9 *You are given a binary tree in which each node contains a value. Design an algorithm to print all paths which sum to a given value. The path does not need to start or end at the root or a leaf.*

pg 86

SOLUTION

Let's approach this problem by using the Simplify and Generalize approach.

Part 1: Simplify—What if the path had to start at the root, but could end anywhere?

In this case, we would have a much easier problem.

We could start from the root and branch left and right, computing the sum thus far on each path. When we find the sum, we print the current path. Note that we don't stop traversing that path just because we found the sum. Why? Because the path could continue on to a +1 node and a -1 node (or any other sequence of nodes where the additional values sum to 0), and the full path would still sum to `sum`.

For example, if `sum = 5`, we could have following paths:

- p = {2, 3}
- q = {2, 3, -4, -2, 6}

If we stopped once we hit 2 + 3, we'd miss this second path and maybe some others. So, we keep going along every possible path.

Part 2: Generalize—The path can start anywhere.

Now, what if the path can start anywhere? In that case, we can make a small modification. On every node, we look "up" to see if we've found the sum. That is, rather than asking "Does this node start a path with the sum?," we ask, "Does this node complete a path with the sum?"

When we recurse through each node n, we pass the function the full path from root to n. This function then adds the nodes along the path in reverse order from n to root. When the sum of each subpath equals sum, then we print this path.

```
1  void findSum(TreeNode node, int sum, int[] path, int level) {
2     if (node == null) {
3        return;
4     }
5
6     /* Insert current node into path. */
7     path[level] = node.data;
8
9     /* Look for paths with a sum that ends at this node. */
10    int t = 0;
11    for (int i = level; i >= 0; i--){
12       t += path[i];
13       if (t == sum) {
14          print(path, i, level);
15       }
16    }
17
18    /* Search nodes beneath this one. */
19    findSum(node.left, sum, path, level + 1);
20    findSum(node.right, sum, path, level + 1);
21
22    /* Remove current node from path. Not strictly necessary, since
23     * we would ignore this value, but it's good practice. */
24    path[level] = Integer.MIN_VALUE;
25 }
26
27 public void findSum(TreeNode node, int sum) {
28    int depth = depth(node);
29    int[] path = new int[depth];
30    findSum(node, sum, path, 0);
31 }
32
33 public static void print(int[] path, int start, int end) {
```

```
34     for (int i = start; i <= end; i++) {
35        System.out.print(path[i] + " ");
36     }
37     System.out.println();
38  }
39
40  public int depth(TreeNode node) {
41     if (node == null) {
42        return 0;
43     } else {
44        return 1 + Math.max(depth(node.left), depth(node.right));
45     }
46  }
```

What is the time complexity of this algorithm (assuming a balanced binary tree)? Well, if a node is at level r, we do r amount of work (that's in the looking "up" step). We can take a guess at O(n log(n)) since there are n nodes doing an average of log(n) amount of work on each step.

If that's too fuzzy for you, we can also be very mathematical about it. Note that there are 2^r nodes at level r.

$$1 * 2^1 + 2 * 2^2 + 3 * 2^3 + 4 * 2^4 + \ldots d * 2^d$$
$$= \text{sum}(r * 2^r, \text{ r from 0 to depth})$$
$$= 2 * (d - 1) * 2^d + 2$$
$$n = 2^d$$
$$d = \log(n)$$

Observe that $2^{\log(x)} = x$.

$$O(2 * (\log(n) - 1) * 2^{\log(n)} + 2)$$
$$= O(2 (\log n - 1) * n)$$
$$= O(n \log(n))$$

The space complexity is O(log n), since the algorithm will recurse O(log n) times and the path parameter is only allocated once (at O(log n) space) during this recursion.

Bit Manipulation

Concepts and Algorithms: Solutions

Chapter 5

5.1 *You are given two 32-bit numbers, N and M, and two bit positions, i and j. Write a method to insert M into N such that M starts at bit j and ends at bit i. You can assume that the bits j through i have enough space to fit all of M. That is, if M = 10011, you can assume that there are at least 5 bits between j and i. You would not, for example, have j = 3 and i = 2, because M could not fully fit between bit 3 and bit 2.*

EXAMPLE:

Input: N = 10000000000, M = 10011, i = 2, j = 6

Output: N = 10001001100

pg 91

SOLUTION

This problem can be approached in three key steps:

1. Clear the bits j through i in N
2. Shift M so that it lines up with bits j through i
3. Merge M and N.

The trickiest part is Step 1. How do we clear the bits in N? We can do this with a mask. This mask will have all 1s, except for 0s in the bits j through i. We create this mask by creating the left half of the mask first, and then the right half.

```
1  int updateBits(int n, int m, int i, int j) {
2     /* Create a mask to clear bits i through j in n
3     /* EXAMPLE: i = 2, j = 4. Result should be 11100011.
4      * For simplicity, we'll use just 8 bits for the example.
5      */
6     int allOnes = ~0; // will equal sequence of all 1s
7
8     // 1s before position j, then 0s. left = 11100000
9     int left = allOnes << (j + 1);
10
11    // 1's after position i. right = 00000011
12    int right = ((1 << i) - 1);
13
14    // All 1s, except for 0s between i and j. mask = 11100011
15    int mask = left | right;
16
17    /* Clear bits j through i then put m in there */
18    int n_cleared = n & mask; // Clear bits j through i.
19    int m_shifted = m << i; // Move m into correct position.
20
21    return n_cleared | m_shifted; // OR them, and we're done!
22 }
```

In a problem like this (and many bit manipulation problems), you should make sure to thoroughly test your code. It's extremely easy to wind up with off-by-one errors.

5.2 *Given a real number between 0 and 1 (e.g., 0.72) that is passed in as a double, print the binary representation. If the number cannot be represented accurately in binary with at most 32 characters, print "ERROR."*

pg 92

SOLUTION

NOTE: When otherwise ambiguous, we'll use the subscripts x_2 and x_{10} to indicate whether x is in base 2 or base 10.

First, let's start off by asking ourselves what a non-integer number in binary looks like. By analogy to a decimal number, the binary number 0.101_2 would look like:

$$0.101_2 = 1 * (1/2^1) + 0 * (1/2^2) + 1 * (1/2^3).$$

To print the decimal part, we can multiply by 2 and check if 2n is greater than or equal to 1. This is essentially "shifting" the fractional sum. That is:

$$\begin{aligned} r &= 2_{10} * n \\ &= 2_{10} * 0.101_2 \\ &= 1 * (1/2^0) + 0 * (1/2^1) + 1 * (1/2^2) \\ &= 1.01_2 \end{aligned}$$

If r >= 1, then we know that n had a 1 right after the decimal point. By doing this continuously, we can check every digit.

```
1  public static String printBinary(double num) {
2     if (num >= 1 || num <= 0) {
3        return "ERROR";
4     }
5
6     StringBuilder binary = new StringBuilder();
7     binary.append(".");
8     while (num > 0) {
9        /* Setting a limit on length: 32 characters */
10       if (binary.length() >= 32) {
11          return "ERROR";
12       }
13
14       double r = num * 2;
15       if (r >= 1) {
16          binary.append(1);
17          num = r - 1;
18       } else {
19          binary.append(0);
20          num = r;
21       }
22    }
23    return binary.toString();
24 }
```

Alternatively, rather than multiplying the number by two and comparing it to 1, we can compare the number to .5, then .25, and so on. The code below demonstrates this approach.

```
1  public static String printBinary2(double num) {
2     if (num >= 1 || num <= 0) {
3        return "ERROR";
4     }
5
6     StringBuilder binary = new StringBuilder();
7     double frac = 0.5;
8     binary.append(".");
9     while (num > 0) {
10       /* Setting a limit on length: 32 characters */
11       if (binary.length() > 32) {
12          return "ERROR";
13       }
14       if (num >= frac) {
15          binary.append(1);
16          num -= frac;
17       } else {
18          binary.append(0);
19       }
20       frac /= 2;
21    }
22    return binary.toString();
23 }
```

Both approaches are equally good; it just depends on which approach you feel most comfortable with.

Either way, you should make sure to prepare thorough test cases for this problem—and to actually run through them in your interview.

5.3 *Given a positive integer, print the next smallest and the next largest number that have the same number of 1 bits in their binary representation.*

pg 92

SOLUTION

There are a number of ways to approach this problem, including using brute force, using bit manipulation, and using clever arithmetic. Note that the arithmetic approach builds on the bit manipulation approach. You'll want to understand the bit manipulation approach before going on to the arithmetic one.

The Brute Force Approach

An easy approach is simply brute force: count the number of 1s in n, and then increment

(or decrement) until you find a number with the same number of 1s. Easy—but not terribly interesting. Can we do something a bit more optimal? Yes!

Let's start with the code for `getNext`, and then move on to `getPrev`.

Bit Manipulation Approach for Get Next Number

If we think about what the next number *should* be, we can observe the following. Given the number 13948, the binary representation looks like:

1	1	0	1	1	0	0	1	1	1	1	1	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

We want to make this number bigger (but not *too* big). We also need to keep the same number of ones.

Observation: Given a number n and two bit locations `i` and `j`, suppose we flip bit `i` from a 1 to a 0, and bit `j` from a 0 to a 1. If `i > j`, then n will have decreased. If `i < j`, then n will have increased.

We know the following:

1. If we flip a zero to a one, we must flip a one to a zero.
2. When we do that, the number will be bigger if and only if the zero-to-one bit was to the left of the one-to-zero bit.
3. We want to make the number bigger, but not unnecessarily bigger. Therefore, we need to flip the rightmost zero which has ones on the right of it.

To put this in a different way, we are flipping the rightmost non-trailing zero. That is, using the above example, the trailing zeros are in the 0th and 1st spot. The rightmost non-trailing zero is at bit 7. Let's call this position p.

Step 1: Flip rightmost non-trailing zero

1	1	0	1	1	0	1	1	1	1	1	1	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

With this change, we have increased the size of n. But, we also have one too many ones, and one too few zeros. We'll need to shrink the size of our number as much as possible while keeping that in mind.

We can shrink the number by rearranging all the bits to the right of bit p such that the 0s are on the left and the 1s are on the right. As we do this, we want to replace one of the 1s with a 0.

A relatively easy way of doing this is to count how many ones are to the right of p, clear all the bits from 0 until p, and then add back in `c1-1` ones. Let `c1` be the number of ones to the right of p and `c0` be the number of zeros to the right of p.

Let's walk through this with an example.

Step 2: Clear bits to the right of p. From before, c0 = 2. c1 = 5. p = 7.

1	1	0	1	1	0	1	0	0	0	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

To clear these bits, we need to create a mask that is a sequence of ones, followed by p zeros. We can do this as follows:

```
a = 1 << p; // all zeros except for a 1 at position p.
b = a - 1;     // all zeros, followed by p ones.
mask = ~b;     // all ones, followed by p zeros.
n = n & mask; // clears rightmost p bits.
```

Or, more concisely, we do:

```
n &= ~((1 << p) - 1).
```

Step 3: Add in c1 - 1 ones.

1	1	0	1	1	0	1	0	0	0	1	1	1	1
13	12	11	10	9	8	7	6	5	4	3	2	1	0

To insert c1 - 1 ones on the right, we do the following:

```
a = 1 << (c1 - 1); // 0s with a 1 at position c1 - 1
b = a - 1;         // 0s with 1s at positions 0 through c1 - 1
n = n | b;         // inserts 1s at positions 0 through c1 - 1
```

Or, more concisely:

```
n |= (1 << (c1 - 1)) - 1;
```

We have now arrived at the smallest number bigger than n with the same number of ones.

The code for getNext is below.

```
1  public int getNext(int n) {
2     /* Compute c0 and c1 */
3     int c = n;
4     int c0 = 0;
5     int c1 = 0;
6     while (((c & 1) == 0) && (c != 0)) {
7        c0++;
8        c >>= 1;
9     }
10
11    while ((c & 1) == 1) {
12       c1++;
13       c >>= 1;
14    }
15
16    /* Error: if n == 11..1100...00, then there is no bigger number
```

```
17      * with the same number of 1s. */
18     if (c0 + c1 == 31 || c0 + c1 == 0) {
19       return -1;
20     }
21
22     int p = c0 + c1; // position of rightmost non-trailing zero
23
24     n |= (1 << p); // Flip rightmost non-trailing zero
25     n &= ~((1 << p) - 1); // Clear all bits to the right of p
26     n |= (1 << (c1 - 1)) - 1; // Insert (c1-1) ones on the right.
27     return n;
28 }
```

Bit Manipulation Approach for Get Previous Number

To implement `getPrev`, we follow a very similar approach.

1. Compute `c0` and `c1`. Note that `c1` is the number of trailing ones, and `c0` is the size of the block of zeros immediately to the left of the trailing ones.
2. Flip the rightmost non-trailing one to a zero. This will be at position `p = c1 + c0`.
3. Clear all bits to the right of bit p.
4. Insert `c1 + 1` ones immediately to the right of position *p*.

Note that Step 2 sets bit p to a zero and Step 3 sets bits 0 through `p-1` to a zero. We can merge these steps.

Let's walk through this with an example.

Step 1: Initial Number. `p = 7. c1 = 2. c0 = 5.`

1	0	0	1	1	1	1	0	0	0	0	0	1	1
13	12	11	10	9	8	7	6	5	4	3	2	1	0

Steps 2 & 3: Clear bits 0 through p.

1	0	0	1	1	1	0	0	0	0	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

We can do this as follows:

```
int a = ~0;             // Sequence of 1s
int b = a << (p + 1);   // Sequence of 1s followed by p + 1 zeros.
n &= b;                 // Clears bits 0 through p.
```

Steps 4: Insert c1 + 1 ones immediately to the right of position p.

1	0	0	1	1	1	0	1	1	1	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note that since p = c1 + c0, the (c1 + 1) ones will be followed by (c0 - 1) zeros.

We can do this as follows:

```
int a = 1 << (c1 + 1); // 0s with 1 at position (c1 + 1)
int b = a - 1;         // 0s followed by c1 + 1 ones
int c = b << (c0 - 1); // c1+1 ones followed by c0-1 zeros.
n |= c;
```

The code to implement this is below.

```
1  int getPrev(int n) {
2     int temp = n;
3     int c0 = 0;
4     int c1 = 0;
5     while (temp & 1 == 1) {
6        c1++;
7        temp >>= 1;
8     }
9
10    if (temp == 0)  return -1;
11
12    while (((temp & 1) == 0) && (temp != 0)) {
13       c0++;
14       temp >>= 1;
15    }
16
17    int p = c0 + c1; // position of rightmost non-trailing one
18    n &= ((~0) << (p + 1)); // clears from bit p onwards
19
20    int mask = (1 << (c1 + 1)) - 1; // Sequence of (c1+1) ones
21    n |= mask << (c0 - 1);
22
23    return n;
24 }
```

Arithmetic Approach to Get Next Number

If c0 is the number of trailing zeros, c1 is the size of the one block immediately following, and p = c0 + c1, we can word our solution from earlier as follows:

1. Set the pth bit to 1
2. Set all bits following p to 0
3. Set bits 0 through c1 - 1 to 1.

A quick and dirty way to perform steps 1 and 2 is to set the trailing zeros to 1 (giving us

p trailing ones), and then add 1. Adding one will flip all trailing ones, so we wind up with a 1 at bit p followed by p zeros. We can perform this arithmetically.

```
n += 2^c0 - 1;     // Sets trailing 0s to 1, giving us p trailing 1s
n += 1;            // Flips first p 1s to 0s, and puts a 1 at bit p.
```

Now, to perform Step 3 arithmetically, we just do:

```
n += 2^(c1 - 1) - 1; // Sets trailing c1 - 1 zeros to ones.
```

This math reduces to:

```
next = n + (2^c0 - 1) + 1 + (2^(c1 - 1) - 1)
     = n + 2^c0 + 2^(c1 - 1) - 1
```

The best part is that, using a little bit manipulation, it's simple to code.

```
1  int getNextArith(int n) {
2    /* ... same calculation for c0 and c1 as before ... */
3    return n + (1 << c0) + (1 << (c1 - 1)) - 1;
4  }
```

Arithmetic Approach to Get Previous Number

If c_1 is the number of trailing ones, c_0 is the size of the zero block immediately following, and $p = c_0 + c_1$, we can word the initial `getPrev` solution as follows:

1. Set the pth bit to 0
2. Set all bits following p to 1
3. Set bits 0 through $c_0 - 1$ to 0.

We can implement this arithmetically as follows. For clarity in the example, we will assume `n = 10000011`. This makes $c_1 = 2$ and $c_0 = 5$.

```
n -= 2^c1 - 1;        // Removes trailing 1s. n is now 10000000.
n -= 1;               // Flips trailing 0s. n is now 01111111.
n -= 2^(c0 - 1) - 1;  // Flips last (c0-1) 0s. n is now 01110000.
```

This reduces mathematically to:

```
next    = n - (2^c1 - 1) - 1 - (2^(c0 - 1) - 1).
        = n - 2^c1 - 2^(c0 - 1) + 1
```

Again, this is very easy to implement.

```
1  int getPrevArith(int n) {
2    /* ... same calculation for c0 and c1 as before ... */
3    return n - (1 << c1) - (1 << (c0 - 1)) + 1;
4  }
```

Whew! Don't worry, you wouldn't be expected to get all this in an interview—at least not without a lot of help from an interviewer.

5.4 *Explain what the following code does: ((n & (n-1)) == 0).*

pg 92

SOLUTION

We can work backwards to solve this question.

What does it mean if A & B == 0?

It means that A and B never have a 1 bit in the same place. So if `n & (n-1) == 0`, then n and `n-1` never share a 1.

What does n-1 look like (as compared with n)?

Try doing subtraction by hand (in base 2 or 10). What happens?

```
  1101011000 [base 2]          593100 [base 10]
-          1               -       1
= 1101010111 [base 2]        = 593099 [base 10]
```

When you subtract 1 from a number, you look at the least significant bit. If it's a 1 you change it to 0, and you are done. If it's a zero, you must "borrow" from a larger bit. So, you go to increasingly larger bits, changing each bit from a 0 to a 1, until you find a 1. You flip that 1 to a 0 and you are done.

Thus, `n-1` will look like n, except that n's initial 0s will be 1s in `n-1`, and n's least significant 1 will be a 0 in `n-1`. That is:

```
if     n = abcde1000
then n-1 = abcde0111
```

So what does n & (n-1) == 0 indicate?

n and `n-1` must have no 1s in common. Given that they look like this:

```
if     n = abcde1000
then n-1 = abcde0111
```

abcde must be all 0s, which means that n must look like this: `00001000`. The value n is therefore a power of two.

So, we have our answer: `((n & (n-1)) == 0)` checks if n is a power of 2 (or if n is 0).

5.5 *Write a function to determine the number of bits required to convert integer A to integer B.*

pg 92

SOLUTION

This seemingly complex problem is actually rather straightforward. To approach this, ask yourself how you would figure out which bits in two numbers are different. Simple: with an XOR.

Each 1 in the XOR represents a bit that is different between A and B. Therefore, to check the number of bits that are different between A and B, we simply need to count the number of bits in A^B that are 1.

```
1  int bitSwapRequired(int a, int b) {
2     int count = 0;
3     for (int c = a ^ b; c != 0; c = c >> 1) {
4        count += c & 1;
5     }
6     return count;
7  }
```

This code is good, but we can make it a bit better. Rather than simply shifting c repeatedly while checking the least significant bit, we can continuously flip the least significant bit and count how long it takes c to reach 0. The operation `c = c & (c - 1)` will clear the least significant bit in c.

The code below utilizes this approach.

```
1  public static int bitSwapRequired(int a, int b) {
2     int count = 0;
3     for (int c = a ^ b; c != 0; c = c & (c-1)) {
4        count++;
5     }
6     return count;
7  }
```

The above code is one of those bit manipulation problems that comes up sometimes in interviews. Though it'd be hard to come up with it on the spot if you've never seen it before, it is useful to remember the trick for your interviews.

5.6 *Write a program to swap odd and even bits in an integer with as few instructions as possible (e.g., bit 0 and bit 1 are swapped, bit 2 and bit 3 are swapped, and so on).*

pg 92

SOLUTION

Like many of the previous problems, it's useful to think about this problem in a different way. Operating on individual pairs of bits would be difficult, and probably not that efficient either. So how else can we think about this problem?

We can approach this as operating on the odds bits first, and then the even bits. Can we take a number n and move the odd bits over by 1? Sure. We can mask all odd bits with `10101010` in binary (which is `0xAA`), then shift them right by 1 to put them in the even spots. For the even bits, we do an equivalent operation. Finally, we merge these

two values.

This takes a total of five instructions. The code below implements this approach.

```
1  public int swapOddEvenBits(int x) {
2     return ( ((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1) );
3  }
```

We've implemented the code above for 32-bit integers in Java. If you were working with 64-bit integers, you would need to change the mask. The logic, however, would remain the same.

5.7 *An array A contains all the integers from 0 through n, except for one number which is missing. In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the jth bit of A[i]," which takes constant time. Write code to find the missing integer. Can you do it in O(n) time?*

pg 92

SOLUTION

You may have seen a very similar sounding problem: Given a list of numbers from 0 to n, with exactly one number removed, find the missing number. This problem can be solved by simply adding the list of numbers and comparing it to the actual sum of 0 through n, which is `n * (n + 1) / 2`. The difference will be the missing number.

We could solve this by computing the value of each number, based on its binary representation, and calculating the sum.

The runtime of this solution is `n * length(n)`, when `length` is the number of bits in n. Note that `length(n)` = $\log_2(n)$. So, the runtime is actually `O(n log(n))`. Not quite good enough!

So how else can we approach it?

We can actually use a similar approach, but leverage the bit values more directly.

Picture a list of binary numbers (the `-----` indicates the value that was removed):

```
00000        00100        01000        01100
00001        00101        01001        01101
00010        00110        01010
-----        00111        01011
```

Removing the number above creates an imbalance of 1s and 0s in the least significant bit, which we'll call LSB_1. In a list of numbers from 0 to n, we would expect there to be the same number of 0s as 1s (if n is odd), or an additional 0 if n is even. That is:

```
if n % 2 == 1 then count(0s) = count(1s)
if n % 2 == 0 then count(0s) = 1 + count(1s)
```

Note that this means that `count(0s)` is always greater than or equal to `count(1s)`.

When we remove a value v from the list, we'll know immediately if v is even or odd just by looking at the least significant bits of all the other values in the list.

	`n % 2 == 0` `count(0s) = 1 + count(1s)`	`n % 2 == 1` `count(0s) = count(1s)`
`v % 2 == 0` $LSB_1(v) = 0$	a 0 is removed. `count(0s) = count(1s)`	a 0 is removed. `count(0s) < count(1s)`
`v % 2 == 1` $LSB_1(v) = 1$	a 1 is removed. `count(0s) > count(1s)`	a 1 is removed. `count(0s) > count(1s)`

So, if `count(0s) <= count(1s)`, then v is even. If `count(0s) > count(1s)`, then v is odd.

Okay, but how do we figure out what the next bit in v is? If v were contained in our list, then we should expect to find the following (where $count_2$ indicates the number of 0s or 1s in the second least significant bit):

$count_2(0s) = count_2(1s)$ OR $count_2(0s) = 1 + count_2(1s)$

As in the earlier example, we can deduce the value of the second least significant bit (LSB_2) of v.

	$count_2(0s) = 1 + count_2(1s)$	$count_2(0s) = count_2(1s)$
$LSB_2(v) == 0$	a 0 is removed. $count_2(0s) = count_2(1s)$	a 0 is removed. $count_2(0s) < count_2(1s)$
$LSB_2(v) == 1$	a 1 is removed. $count_2(0s) > count_2(1s)$	a 1 is removed. $count_2(0s) > count_2(1s)$

Again, we have the same conclusion:

- If $count_2(0s) <= count_2(1s)$, then $LSB_2(v) = 0$.
- If $count_2(0s) > count_2(1s)$, then $LSB_2(v) = 1$.

We can repeat this process for each bit. On each iteration, we count the number of 0s and 1s in bit i to check if $LSB_1(v)$ is 0 or 1. Then, we discard the numbers where $LSB_1(x) != LSB_1(v)$. That is, if v is even, we discard the odd numbers, and so on.

By the end of this process, we will have computed all bits in v. In each successive iteration, we look at n, then `n / 2`, then `n / 4`, and so on, bits. This results in a runtime of `O(N)`.

If it helps, we can also move through this more visually. In the first iteration, we start with all the numbers:

```
00000          00100          01000          01100
00001          00101          01001          01101
00010          00110          01010
-----          00111          01011
```

Since $count_1(0s) > count_1(1s)$, we know that $LSB_1(v) = 1$. Now, discard all numbers x where $LSB_1(x) != LSB_1(v)$.

```
~~00000~~      ~~00100~~      ~~01000~~      ~~01100~~
00001          00101          01001          01101
~~00010~~      ~~00110~~      ~~01010~~
-----          00111          01011
```

Now, $count_2(0s) > count_2(1s)$, so we know that $LSB_2(v) = 1$. Now, discard all numbers x where $LSB_2(x) != LSB_2(v)$.

```
~~00000~~      ~~00100~~      ~~01000~~      ~~01100~~
~~00001~~      ~~00101~~      ~~01001~~      ~~01101~~
~~00010~~      ~~00110~~      ~~01010~~
-----          00111          01011
```

This time, $count_3(0s) <= count_3(1s)$, we know that $LSB_3(v) = 0$. Now, discard all numbers x where $LSB_3(x) != LSB_3(v)$.

```
~~00000~~      ~~00100~~      ~~01000~~      ~~01100~~
~~00001~~      ~~00101~~      ~~01001~~      ~~01101~~
~~00010~~      ~~00110~~      ~~01010~~
-----          ~~00111~~      01011
```

We're down to just one number. In this case, $count_4(0s) <= count_4(1s)$, so $LSB_4(v) = 0$.

When we discard all numbers where $LSB_4(v) != 0$, we'll wind up with an empty list. Once the list is empty, then $count_1(0s) <= count_1(1s)$, so $LSB_1(v) = 0$. In other words, once we have an empty list, we can fill in the rest of the bits of v with 0.

This process will compute that, for the example above, v = 00011.

The code below implements this algorithm. We've implemented the discarding aspect by partitioning the array by bit value as we go.

```
1  public int findMissing(ArrayList<BitInteger> array) {
2    /* Start from the least significant bit, and work our way up */
3    return findMissing(array, 0);
4  }
5
6  public int findMissing(ArrayList<BitInteger> input, int column) {
7    if (column >= BitInteger.INTEGER_SIZE) { // We're done!
8      return 0;
9    }
10   ArrayList<BitInteger> oneBits =
```

```
11        new ArrayList<BitInteger>(input.size()/2);
12     ArrayList<BitInteger> zeroBits =
13        new ArrayList<BitInteger>(input.size()/2);
14
15     for (BitInteger t : input) {
16        if (t.fetch(column) == 0) {
17           zeroBits.add(t);
18        } else {
19           oneBits.add(t);
20        }
21     }
22     if (zeroBits.size() <= oneBits.size()) {
23        int v = findMissing(zeroBits, column + 1);
24        return (v << 1) | 0;
25     } else {
26        int v = findMissing(oneBits, column + 1);
27        return (v << 1) | 1;
28     }
29  }
```

In lines 24 and 27, we recursively calculate the other bits of v. Then, we insert either a 0 or 1, depending on whether or not $\texttt{count}_1(\texttt{0s}) \texttt{ <= } \texttt{count}_1(\texttt{1s})$.

5.8 *A monochrome screen is stored as a single array of bytes, allowing eight consecutive pixels to be stored in one byte. The screen has width w, where w is divisible by 8 (that is, no byte will be split across rows). The height of the screen, of course, can be derived from the length of the array and the width. Implement a function drawHorizontalLine(byte[] screen, int width, int x1, int x2, int y) which draws a horizontal line from (x1, y) to (x2, y).*

pg 92

SOLUTION

A naive solution to the problem is straightforward: iterate in a for loop from x1 to x2, setting each pixel along the way. But that's hardly any fun, is it? (Nor is it very efficient.)

A better solution is to recognize that if x1 and x2 are far away from each other, several full bytes will be contained between them. These full bytes can be set one at a time by doing `screen[byte_pos] = 0xFF`. The residual start and end of the line can be set using masks.

```
1  void drawLine(byte[] screen, int width, int x1, int x2, int y) {
2     int start_offset = x1 % 8;
3     int first_full_byte = x1 / 8;
4     if (start_offset != 0) {
5        first_full_byte++;
6     }
7
8     int end_offset = x2 % 8;
```

```
9     int last_full_byte = x2 / 8;
10    if (end_offset != 7) {
11       last_full_byte--;
12    }
13
14    // Set full bytes
15    for (int b = first_full_byte; b <= last_full_byte; b++) {
16       screen[(width / 8) * y + b] = (byte) 0xFF;
17    }
18
19    // Create masks for start and end of line
20    byte start_mask = (byte) (0xFF >> start_offset);
21    byte end_mask = (byte) ~(0xFF >> (end_offset + 1));
22
23    // Set start and end of line
24    if ((x1 / 8) == (x2 / 8)) { // x1 and x2 are in the same byte
25       byte mask = (byte) (start_mask & end_mask);
26       screen[(width / 8) * y + (x1 / 8)] |= mask;
27    } else {
28       if (start_offset != 0) {
29          int byte_number = (width / 8) * y + first_full_byte - 1;
30          screen[byte_number] |= start_mask;
31       }
32       if (end_offset != 7) {
33          int byte_number = (width / 8) * y + last_full_byte + 1;
34          screen[byte_number] |= end_mask;
35       }
36    }
37 }
```

Be careful on this problem; there are a lot of "gotchas" and special cases. For example, you need to consider the case where x1 and x2 are in the same byte. Only the most careful candidates can implement this code bug-free.

Brain Teasers

Concepts and Algorithms: Solutions

Chapter 6

6.1 *You have 20 bottles of pills. 19 bottles have 1.0 gram pills, but one has pills of weight 1.1 grams. Given a scale that provides an exact measurement, how would you find the heavy bottle? You can only use the scale once.*

pg 95

SOLUTION

Sometimes, tricky constraints can be a clue. This is the case with the constraint that we can only use the scale once.

Because we can only use the scale once, we know something interesting: we must weigh multiple pills at the same time. In fact, we know we must weigh pills from at least 19 bottles at the same time. Otherwise, if we skipped two or more bottles entirely, how could we distinguish between those missed bottles? Remember that we only have *one* chance to use the scale.

So how can we weigh pills from more than one bottle and discover which bottle has the heavy pills? Let's suppose there were just two bottles, one of which had heavier pills. If we took one pill from each bottle, we would get a weight of 2.1 grams, but we wouldn't know which bottle contributed the extra 0.1 grams. We know we must treat the bottles differently somehow.

If we took one pill from Bottle #1 and two pills from Bottle #2, what would the scale show? It depends. If Bottle #1 were the heavy bottle, we would get 3.1 grams. If Bottle #2 were the heavy bottle, we would get 3.2 grams. And that is the trick to this problem.

We know the "expected" weight of a bunch of pills. The difference between the expected weight and the actual weight will indicate which bottle contributed the heavier pills, *provided* we select a different number of pills from each bottle.

We can generalize this to the full solution: take one pill from Bottle #1, two pills from Bottle #2, three pills from Bottle #3, and so on. Weigh this mix of pills. If all pills were one gram each, the scale would read 210 grams (`1 + 2 + ... + 20 = 20 * 21 / 2 = 210`). Any "overage" must come from the extra 0.1 gram pills.

This formula will tell you the bottle number: `(weight - 210 grams) / 0.1 grams`. So, if the set of pills weighed 211.3 grams, then Bottle #13 would have the heavy pills.

6.2 *There is an 8x8 chess board in which two diagonally opposite corners have been cut off. You are given 31 dominos, and a single domino can cover exactly two squares. Can you use the 31 dominos to cover the entire board? Prove your answer (by providing an example or showing why it's impossible).*

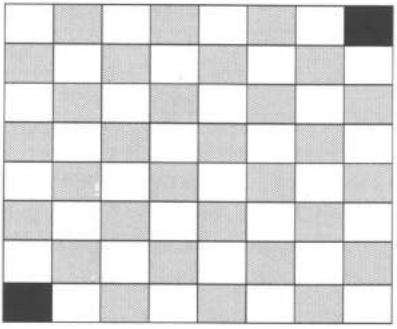
pg 95

SOLUTION

At first, it seems like this should be possible. It's an 8 x 8 board, which has 64 squares,

but two have been cut off, so we're down to 62 squares. A set of 31 dominoes should be able to fit there, right?

When we try to lay down dominoes on row 1, which only has 7 squares, we may notice that one domino must stretch into the row 2. Then, when we try to lay down dominoes onto row 2, again we need to stretch a domino into row 3.



For each row we lay down, we'll always have one domino that needs to poke into the next row. No matter how many times and ways we try to solve this issue, we won't be able to successfully lay down all the dominoes.

There's a cleaner, more solid proof for why it won't work. The chess board initially has 32 black and 32 white squares. By removing opposite corners (which must be the same color), we're left with 30 of one color and 32 of the other color. Let's say, for the sake of argument, that we have 30 black and 32 white squares.

Each domino we set on the board will always take up one white and one black square. Therefore, 31 dominos will take up 31 white squares and 31 black squares exactly. On this board, however, we must have 30 black squares and 32 white squares. Hence, it is impossible.

6.3 *You have a five-quart jug, a three-quart jug, and an unlimited supply of water (but no measuring cups). How would you come up with exactly four quarts of water? Note that the jugs are oddly shaped, such that filling up exactly "half" of the jug would be impossible.*

pg 95

SOLUTION

If we just play around with the jugs, we'll find that we can pour water back and forth between them as follows:

5 Quart	3 Quart	Note
5	0	Filled 5 quart jug.
2	3	Filled 3 quart with 5 quart's contents.
2	0	Dumped 3 quart.
0	2	Fill 3 quart with 5 quart's contents.
5	2	Filled 5 quart.
4	3	Fill remainder of 3 quart with 5 quart.
4		Done! We have four quarts.

Many brain teasers have a math or CS root to them, and this is one of them. As long as the two jug sizes are relatively prime (i.e., have no common prime factors), you can find a pour sequence for any value between one and the sum of the jug sizes.

6.4 *A bunch of people are living on an island, when a visitor comes with a strange order: all blue-eyed people must leave the island as soon as possible. There will be a flight out at 8:00pm every evening. Each person can see everyone else's eye color, but they do not know their own (nor is anyone allowed to tell them). Additionally, they do not know how many people have blue eyes, although they do know that at least one person does. How many days will it take the blue-eyed people to leave?*

pg 96

SOLUTION

Let's apply the Base Case and Build approach. Assume that there are n people on the island and c of them have blue eyes. We are explicitly told that $c > 0$.

Case c = 1: Exactly one person has blue eyes.

Assuming all the people are intelligent, the blue-eyed person should look around and realize that no one else has blue eyes. Since he knows that at least one person has blue eyes, he must conclude that it is him. Therefore, he would take the flight that evening.

Case c = 2: Exactly two people have blue eyes.

The two blue-eyed people see each other, but be unsure whether c is 1 or 2. They know, from the previous case, that if $c = 1$, the blue-eyed person would leave on the first night. Therefore, if the other blue-eyed person is still there, he must deduce that $c = 2$, which means that he himself has blue eyes. Both men would then leave on the second night.

Case c > 2: The General Case.

As we increase c, we can see that this logic continues to apply. If $c = 3$, then those three

people will immediately know that there are either 2 or 3 people with blue eyes. If there were two people, then those two people would have left on the second night. So, when the others are still around after that night, each person would conclude that c = 3 and that they, therefore, have blue eyes too. They would leave that night.

This same pattern extends up through any value of c. Therefore, if c men have blue eyes, it will take c nights for the blue-eyed men to leave. All will leave on the same night.

6.5 *There is a building of 100 floors. If an egg drops from the Nth floor or above, it will break. If it's dropped from any floor below, it will not break. You're given two eggs. Find N, while minimizing the number of drops for the worst case.*

pg 96

SOLUTION

We may observe that, regardless of how we drop Egg 1, Egg 2 must do a linear search (from lowest to highest) between the "breaking floor" and the next highest non-breaking floor. For example, if Egg 1 is dropped from floors 5 and 10 without breaking, but it breaks when it's dropped from floor 15, then Egg 2 must be dropped, in the worst case, from floors 11, 12, 13, and 14.

The Approach

As a first try, suppose we drop an egg from the 10th floor, then the 20th, ...

- If Egg 1 breaks on the first drop (floor 10), then we have at most 10 drops total.
- If Egg 1 breaks on the last drop (floor 100), then we have at most 19 drops total (floors 10, 20, ...,90, 100, then 91 through 99).

That's pretty good, but all we've considered is the absolute worst case. We should do some "load balancing" to make those two cases more even.

Our goal is to create a system for dropping Egg 1 such that the number of drops is as consistent as possible, whether Egg 1 breaks on the first drop or the last drop.

1. A perfectly load-balanced system would be one in which `Drops(Egg 1) + Drops(Egg 2)` is always the same, regardless of where Egg 1 breaks.
2. For that to be the case, since each drop of Egg 1 takes one more step, Egg 2 is allowed one fewer step.
3. We must, therefore, reduce the number of steps potentially required by Egg 2 by one drop each time. For example, if Egg 1 is dropped on floor 20 and then floor 30, Egg 2 is potentially required to take 9 steps. When we drop Egg 1 again, we must reduce potential Egg 2 steps to only 8. That is, we must drop Egg 1 at floor 39.
4. We know, therefore, Egg 1 must start at floor X, then go up by X-1 floors, then X-2, ..., until it gets to 100.

5. Solve for X in: `X + (X-1) + (X-2) + ... + 1 = 100.`
   `X (X + 1) / 2 = 100 -> X = 14.`

We go to floor 14, then 27, then 39, ... This takes 14 steps in the worse case.

As in many other maximizing / minimizing problems, the key in this problem is "worst case balancing."

6.6 *There are 100 closed lockers in a hallway. A man begins by opening all 100 lockers. Next, he closes every second locker. Then, on his third pass, he toggles every third locker (closes it if it is open or opens it if it is closed). This process continues for 100 passes, such that on each pass i, the man toggles every ith locker. After his 100th pass in the hallway, in which he toggles only locker #100, how many lockers are open?*

pg 96

SOLUTION

We can tackle this problem by thinking through what it means for a door to be toggled. This will help us deduce which doors at the very end will be left opened.

Question: For which rounds is a door toggled (open or closed)?

A door n is toggled once for each factor of n, including itself and 1. That is, door 15 is toggled on rounds 1, 3, 5, and 15.

Question: When would a door be left open?

A door is left open if the number of factors (which we will call x) is odd. You can think about this by pairing factors off as an open and a close. If there's one remaining, the door will be open.

Question: When would x be odd?

The value x is odd if n is a perfect square. Here's why: pair n's factors by their complements. For example, if n is 36, the factors are (1, 36), (2, 18), (3, 12), (4, 9), (6, 6). Note that (6, 6) only contributes one factor, thus giving n an odd number of factors.

Question: How many perfect squares are there?

There are 10 perfect squares. You could count them (1, 4, 9, 16, 25, 36, 49, 64, 81, 100), or you could simply realize that you can take the numbers 1 through 10 and square them:

```
1*1, 2*2, 3*3, ..., 10*10
```

Therefore, there are 10 lockers open at the end of this process.

Mathematics and Probability

Concepts and Algorithms: Solutions

Chapter 7

7.1 *You have a basketball hoop and someone says that you can play one of two games.*

Game 1: You get one shot to make the hoop.

Game 2: You get three shots and you have to make two of three shots.

If p is the probability of making a particular shot, for which values of p should you pick one game or the other?

pg 102

SOLUTION

To solve this problem, we can apply straightforward probability laws by comparing the probabilities of winning each game.

Probability of winning Game 1:

The probability of winning Game 1 is p, by definition.

Probability of winning Game 2:

Let `s(k,n)` be the probability of making exactly k shots out of n. The probability of winning Game 2 is the probability of making exactly two shots out of three OR making all three shots. In other words:

```
P(winning) = s(2,3) + s(3,3)
```

The probability of making all three shots is:

```
s(3,3) = p^3
```

The probability of making exactly two shots is:

```
P(making 1 and 2, and missing 3)
        + P(making 1 and 3, and missing 2)
        + P(missing 1, and making 2 and 3)
    = p * p * (1 - p) + p * (1 - p) * p + (1 - p) * p * p
    = 3 (1 - p) p^2
```

Adding these together, we get:

```
    = p^3 + 3 (1 - p) p^2
    = p^3 + 3p^2 - 3p^3
    = 3p^2 - 2p^3
```

Which game should you play?

You should play Game 1 if `P(Game 1) > P(Game 2)`:

```
p > 3p^2 - 2p^3.
1 > 3p - 2p^2
2p^2 - 3p + 1 > 0
(2p - 1)(p - 1) > 0
```

Both terms must be positive, or both must be negative. But we know `p < 1`, so `p - 1`

< 0. This means both terms must be negative.

```
2p - 1 < 0
2p < 1
p < .5
```

So, we should play Game1 if p < .5. If p = 0, 0.5, or 1, then P(Game 1) = P(Game 2), so it doesn't matter which game we play.

7.2 *There are three ants on different vertices of a triangle. What is the probability of collision (between any two or all of them) if they start walking on the sides of the triangle? Assume that each ant randomly picks a direction, with either direction being equally likely to be chosen, and that they walk at the same speed.*

Similarly, find the probability of collision with n ants on an n-vertex polygon.

pg 102

SOLUTION

The ants will collide if any of them are moving towards each other. So, the only way that they won't collide is if they are all moving in the same direction (clockwise or counter-clockwise). We can compute this probability and work backwards from there.

Since each ant can move in two directions, and there are three ants, the probability is:

$P(\text{clockwise}) = (1/2)^3$

$P(\text{counter clockwise}) = (1/2)^3$

$P(\text{same direction}) = (1/2)^3 + (1/2)^3 = 1/4$

The probability of the ants colliding is therefore the probability of the ants *not* moving in the same direction:

$P(\text{collision}) = 1 - P(\text{same direction}) = 1 - (1/4) = 3/4$

To generalize this to an n-vertex polygon: there are still only two ways in which the ants can move to avoid a collision, but there are 2^n ways they can move in total. Therefore, in general, probability of collision is:

$P(\text{clockwise}) = (1/2)^n$

$P(\text{counter}) = (1/2)^n$

$P(\text{same direction}) = 2(1/2)^n = (1/2)^{n-1}$

$P(\text{collision}) = 1 - P(\text{same direction}) = 1 - (1/2)^{n-1}$

7.3 *Given two lines on a Cartesian plane, determine whether the two lines would intersect.*

pg 102

SOLUTION

There are a lot of unknowns in this problem: What format are the lines in? What if they are the same line? You should discuss these ambiguities with your interviewer.

We'll make the following assumptions:

- If two lines are the same (same slope and y-intercept), then they are considered to intersect.
- We get to decide the data structure for the line.

If two *different* lines are not parallel, then they intersect. Thus, to check if two lines intersect, we just need to check if the slopes are different (or if the lines are identical).

We can implement the code as follows:

```
1  public class Line {
2     static double epsilon = 0.000001;
3     public double slope;
4     public double yintercept;
5
6     public Line(double s, double y) {
7        slope = s;
8        yintercept = y;
9     }
10
11    public boolean intersect(Line line2) {
12       return Math.abs(slope - line2.slope) > epsilon ||
13              Math.abs(yintercept - line2.yintercept) < epsilon;
14    }
15 }
```

In problems like these, be aware of the following:

- Ask questions. This question has a lot of unknowns—ask questions to clarify them. Many interviewers intentionally ask vague questions to see if you'll clarify your assumptions.
- When possible, design and use data structures. It shows that you understand and care about object-oriented design.
- Think through which data structures you design to represent a line. There are a lot of options, with lots of trade-offs. Pick one, and explain your choice.
- Don't assume that the slope and y-intercept are integers.
- Understand limitations of floating point representations. Never check for equality with ==. Instead, check if the difference is less than an `epsilon` value.

7.4 *Write methods to implement the multiply, subtract, and divide operations for integers. Use only the add operator.*

pg 102

SOLUTION

The only operation we have to work with is the add operator. In each of these problems, it's useful to think in depth about what these operations really do or how to phrase them in terms of other operations (either add or operations we've already completed).

Subtraction

How can we phrase subtraction in terms of addition? This one is pretty straightforward. The operation `a - b` is the same thing as `a + (-1) * b`. However, because we are not allowed to use the * (multiply) operator, we must implement a `negate` function.

```
1  /* Flip a positive sign to negative or negative sign to positive */
2  public static int negate(int a) {
3     int neg = 0;
4     int d = a < 0 ? 1 : -1;
5     while (a != 0) {
6        neg += d;
7        a += d;
8     }
9     return neg;
10 }
11
12 /* Subtract two numbers by negating b and adding them */
13 public static int minus(int a, int b) {
14    return a + negate(b);
15 }
```

The negation of the value k is implemented by adding -1 k times.

Multiplication

The connection between addition and multiplication is equally straightforward. To multiply a by b, we just add a to itself b times.

```
1  /* Multiply a by b by adding a to itself b times */
2  public static int multiply(int a, int b) {
3     if (a < b) {
4        return multiply(b, a); // algorithm is faster if b < a
5     }
6     int sum = 0;
7     for (int i = abs(b); i > 0; i--) {
8        sum += a;
9     }
10    if (b < 0) {
11       sum = negate(sum);
```

```
12    }
13    return sum;
14 }
15
16 /* Return absolute value */
17 public static int abs(int a) {
18    if (a < 0) {
19       return negate(a);
20    } else {
21       return a;
22    }
23 }
```

The one thing we need to be careful of in the above code is to properly handle multiplication of negative numbers. If b is negative, we need to flip the value of sum. So, what this code really does is:

```
multiply(a, b) <-- abs(b) * a * (-1 if b < 0).
```

We also implemented a simple abs function to help.

Division

Of the three operations, division is certainly the hardest. The good thing is that we can use the `multiply`, `subtract`, and `negate` methods now to implement `divide`.

We are trying to compute x where `x = a / b`. Or, to put this another way, find x where `a = bx`. We've now changed the problem into one that can be stated with something we know how to do: multiplication.

We could implement this by multiplying b by progressively higher values, until we reach a. That would be fairly inefficient, particularly given that our implementation of `multiply` involves a lot of adding.

Alternatively, we can look at the equation `a = xb` to see that we can compute x by adding b to itself repeatedly until we reach a. The number of times we need to do that will equal x.

Of course, a might not be evenly divisible by b, and that's okay. Integer division, which is what we've been asked to implement, is supposed to `floor` the result.

The code below implements this algorithm.

```
1  public int divide(int a, int b)
2        throws java.lang.ArithmeticException {
3     if (b == 0) {
4        throw new java.lang.ArithmeticException("ERROR");
5     }
6     int absa = abs(a);
7     int absb = abs(b);
8
9     int product = 0;
```

```
10    int x = 0;
11    while (product + absb <= absa) { /* don't go past a */
12       product += absb;
13       x++;
14    }
15
16    if ((a < 0 && b < 0) || (a > 0 && b > 0)) {
17       return x;
18    } else {
19       return negate(x);
20    }
21 }
```

In tackling this problem, you should be aware of the following:

- A logical approach of going back to what exactly multiplication and division do comes in handy. Remember that. All (good) interview problems can be approached in a logical, methodical way!
- The interviewer is looking for this sort of logical work-your-way-through-it approach.
- This is a great problem to demonstrate your ability to write clean code—specifically, to show your ability to reuse code. For example, if you were writing this solution and didn't put `negate` in its own method, you should move it into its own method once you see that you'll use it multiple times.
- Be careful about making assumptions while coding. Don't assume that the numbers are all positive or that a is bigger than b.

7.5 *Given two squares on a two-dimensional plane, find a line that would cut these two squares in half. Assume that the top and the bottom sides of the square run parallel to the x-axis.*

pg 102

SOLUTION

Before we start, we should think about what exactly this problem means by a "line." Is a line defined by a slope and a y-intercept? Or by any two points on the line? Or, should the line be really a line segment, which starts and ends at the edges of the squares?

We will assume, since it makes the problem a bit more interesting, that we mean the third option: that the line should end at the edges of the squares. In an interview situation, you should discuss this with your interviewer.

This line that cuts two squares in half must connect the two middles. We can easily calculate the slope, knowing that $\texttt{slope} = \frac{x1-y2}{x1-x2}$. Once we calculate the slope using the two middles, we can use the same equation to calculate the start and end points of the line segment.

In the below code, we will assume the origin (0, 0) is in the upper left-hand corner.

```
1  public class Square {
2     ...
3     public Point middle() {
4        return new Point((this.left + this.right) / 2.0,
5                         (this.top + this.bottom) / 2.0);
6     }
7
8     /* Return the point where the line segment connecting mid1 and
9      * mid2 intercepts the edge of square 1. That is, draw a line
10     * from mid2 to mid1, and continue it out until the edge of
11     * the square. */
12    public Point extend(Point mid1, Point mid2, double size) {
13       /* Find what direction the line mid2 -> mid1 goes. */
14       double xdir = mid1.x < mid2.x ? -1 : 1;
15       double ydir = mid1.y < mid2.y ? -1 : 1;
16
17       /* If mid1 and mid2 have the same x value, then the slope
18        * calculation will throw a divide by 0 exception. So, we
19        * compute this specially. */
20       if (mid1.x == mid2.x) {
21          return new Point(mid1.x, mid1.y + ydir * size / 2.0);
22       }
23
24       double slope = (mid1.y - mid2.y) / (mid1.x - mid2.x);
25       double x1 = 0;
26       double y1 = 0;
27
28       /* Calculate slope using the equation (y1 - y2) / (x1 - x2).
29        * Note: if the slope is "steep" (>1) then the end of the
30        * line segment will hit size / 2 units away from the middle
31        * on the y axis. If the slope is "shallow" (<1) the end of
32        * the line segment will hit size / 2 units away from the
33        * middle on the x axis. */
34       if (Math.abs(slope) == 1) {
35          x1 = mid1.x + xdir * size / 2.0;
36          y1 = mid1.y + ydir * size / 2.0;
37       } else if (Math.abs(slope) < 1) { // shallow slope
38          x1 = mid1.x + xdir * size / 2.0;
39          y1 = slope * (x1 - mid1.x) + mid1.y;
40       } else { // steep slope
41          y1 = mid1.y + ydir * size / 2.0;
42          x1 = (y1 - mid1.y) / slope + mid1.x;
43       }
44       return new Point(x1, y1);
45    }
46
47    public Line cut(Square other) {
48       /* Calculate where a line between each middle would collide
```