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 $O(x + 2x + ... + nx)$. This reduces to $O(xn^2)$. (Why isn't it $O(xn^n)$? Because $1 + 2 + ... + n$ 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

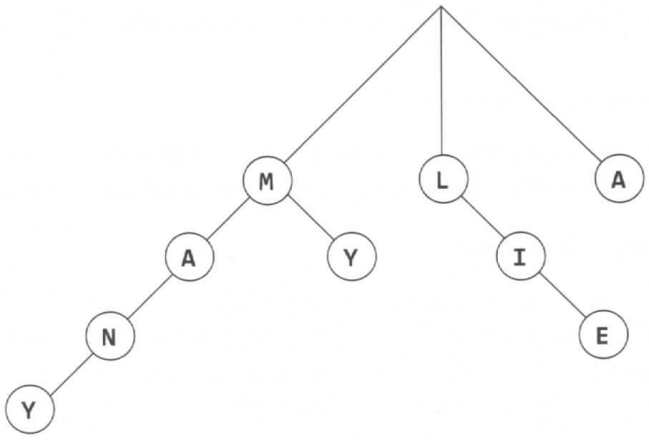
Prior to your interview, you should be comfortable implementing in-order, post-order, and pre-order traversal. The most common of these, in-order traversal, works by visiting the left side, then the current node, then the right.

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.

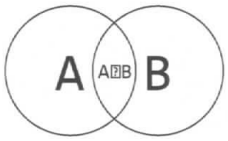
```
1  boolean[] sieveOfEratosthenes(int max) {
2     boolean[] flags = new boolean[max + 1];
3     int count = 0;
4
5     init(flags); // Set all flags to true other than 0 and 1
6     int prime = 2;
7
8     while (prime <= Math.sqrt(max)) {
9        /* Cross off remaining multiples of prime */
10       crossOff(flags, prime);
11
12       /* Find next value which is true */
13       prime = getNextPrime(flags, prime);
14
15       if (prime >= flags.length) {
16          break;
17       }
18    }
19
20    return flags;
21 }
22
23 void crossOff(boolean[] flags, int prime) {
24    /* Cross off remaining multiples of prime. We can start with
25     * (prime*prime), because if we have a k * prime, where
26     * k < prime, this value would have already been crossed off in
27     * a prior iteration. */
28    for (int i = prime * prime; i < flags.length; i += prime) {
29       flags[i] = false;
30    }
31 }
32
33 int getNextPrime(boolean[] flags, int prime) {
34    int next = prime + 1;
35    while (next < flags.length && !flags[next]) {
36       next++;
37    }
38    return next;
39 }
```

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:

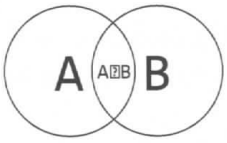
```
P(x is even and x <= 5)
    = P(x is even given x <= 5) P(x <= 5)
    = (2/5) * (1/2)
    = 1/5
```

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.

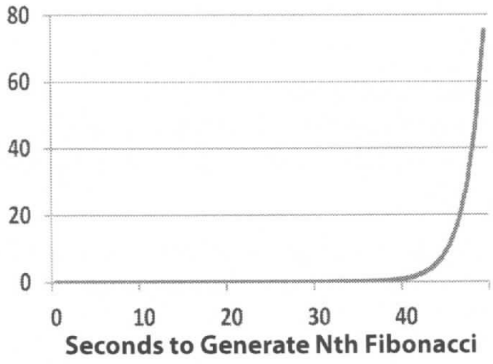
If you're unlucky enough to get a DP problem, you can approach it much the same way as a recursion problem. The difference is that intermediate results are "cached" for future calls.

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.

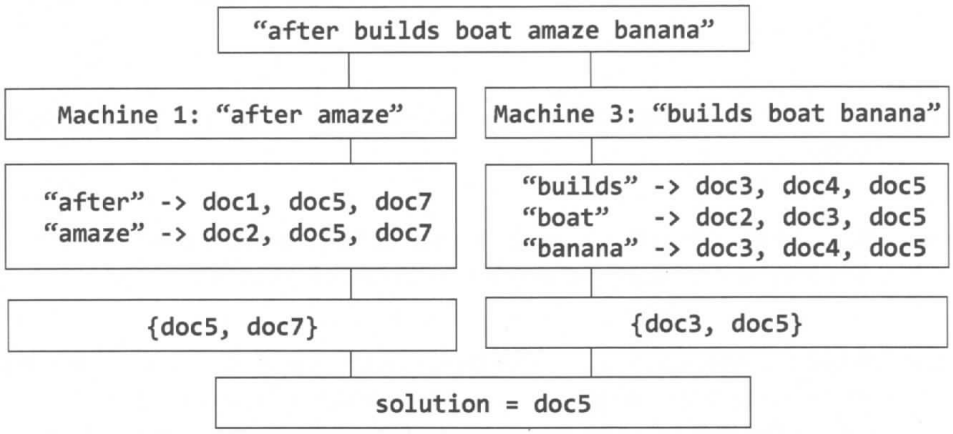
The advantage of this approach is that the lookup table is small and simple (since it must only specify a range of values), and each machine can store a copy of the lookup table. The disadvantage, however, is that if new documents or words are added, we may need to perform an expensive shift of keywords.

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)