3

Solutions to Stacks and Queues

3.1 Three in One: Describe how you could use a single array to implement three stacks.

pg 98

SOLUTION

Like many problems, this one somewhat depends on how well we'd like to support these stacks. If we're okay 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   class FixedMultiStack {
2     private int numberOfStacks = 3;
3     private int stackCapacity;
4     private int[] values;
5     private int[] sizes;
6
7     public FixedMultiStack(int stackSize) {
8       stackCapacity = stackSize;
9       values = new int[stackSize * numberOfStacks];
10      sizes = new int[numberOfStacks];
11    }
12
13    /* Push value onto stack. */
14    public void push(int stackNum, int value) throws FullStackException {
15      /* Check that we have space for the next element */
16      if (isFull(stackNum)) {
17        throw new FullStackException();
```

```
18      }
19
20      /* Increment stack pointer and then update top value. */
21      sizes[stackNum]++;
22      values[indexOfTop(stackNum)] = value;
23    }
24
25    /* Pop item from top stack. */
26    public int pop(int stackNum) {
27      if (isEmpty(stackNum)) {
28        throw new EmptyStackException();
29      }
30
31      int topIndex = indexOfTop(stackNum);
32      int value = values[topIndex]; // Get top
33      values[topIndex] = 0; // Clear
34      sizes[stackNum]--; // Shrink
35      return value;
36    }
37
38    /* Return top element. */
39    public int peek(int stackNum) {
40      if (isEmpty(stackNum)) {
41        throw new EmptyStackException();
42      }
43      return values[indexOfTop(stackNum)];
44    }
45
46    /* Return if stack is empty. */
47    public boolean isEmpty(int stackNum) {
48      return sizes[stackNum] == 0;
49    }
50
51    /* Return if stack is full. */
52    public boolean isFull(int stackNum) {
53      return sizes[stackNum] == stackCapacity;
54    }
55
56    /* Returns index of the top of the stack. */
57    private int indexOfTop(int stackNum) {
58      int offset = stackNum * stackCapacity;
59      int size = sizes[stackNum];
60      return offset + size - 1;
61    }
62  }
```

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 much work.

```
1  public class MultiStack {
2    /* StackInfo is a simple class that holds a set of data about each stack. It
3     * does not hold the actual items in the stack. We could have done this with
4     * just a bunch of individual variables, but that's messy and doesn't gain us
5     * much. */
6    private class StackInfo {
7      public int start, size, capacity;
8      public StackInfo(int start, int capacity) {
9        this.start = start;
10       this.capacity = capacity;
11     }
12
13     /* Check if an index on the full array is within the stack boundaries. The
14      * stack can wrap around to the start of the array. */
15     public boolean isWithinStackCapacity(int index) {
16       /* If outside of bounds of array, return false. */
17       if (index < 0 || index >= values.length) {
18         return false;
19       }
20
21       /* If index wraps around, adjust it. */
22       int contiguousIndex = index < start ? index + values.length : index;
23       int end = start + capacity;
24       return start <= contiguousIndex && contiguousIndex < end;
25     }
26
27     public int lastCapacityIndex() {
28       return adjustIndex(start + capacity - 1);
29     }
30
31     public int lastElementIndex() {
32       return adjustIndex(start + size - 1);
33     }
34
35     public boolean isFull() { return size == capacity; }
36     public boolean isEmpty() { return size == 0; }
37   }
38
39   private StackInfo[] info;
40   private int[] values;
41
42   public MultiStack(int numberOfStacks, int defaultSize) {
43     /* Create metadata for all the stacks. */
44     info = new StackInfo[numberOfStacks];
45     for (int i = 0; i < numberOfStacks; i++) {
46       info[i] = new StackInfo(defaultSize * i, defaultSize);
47     }
48     values = new int[numberOfStacks * defaultSize];
49   }
50
51   /* Push value onto stack num, shifting/expanding stacks as necessary. Throws
52    * exception if all stacks are full. */
53   public void push(int stackNum, int value) throws FullStackException {
```

```
54      if (allStacksAreFull()) {
55        throw new FullStackException();
56      }
57
58      /* If this stack is full, expand it. */
59      StackInfo stack = info[stackNum];
60      if (stack.isFull()) {
61        expand(stackNum);
62      }
63
64      /* Find the index of the top element in the array + 1, and increment the
65       * stack pointer */
66      stack.size++;
67      values[stack.lastElementIndex()] = value;
68    }
69
70    /* Remove value from stack. */
71    public int pop(int stackNum) throws Exception {
72      StackInfo stack = info[stackNum];
73      if (stack.isEmpty()) {
74        throw new EmptyStackException();
75      }
76
77      /* Remove last element. */
78      int value = values[stack.lastElementIndex()];
79      values[stack.lastElementIndex()] = 0; // Clear item
80      stack.size--; // Shrink size
81      return value;
82    }
83
84    /* Get top element of stack.*/
85    public int peek(int stackNum) {
86      StackInfo stack = info[stackNum];
87      return values[stack.lastElementIndex()];
88    }
89    /* Shift items in stack over by one element. If we have available capacity, then
90     * we'll end up shrinking the stack by one element. If we don't have available
91     * capacity, then we'll need to shift the next stack over too. */
92    private void shift(int stackNum) {
93      System.out.println("/// Shifting " + stackNum);
94      StackInfo stack = info[stackNum];
95
96      /* If this stack is at its full capacity, then you need to move the next
97       * stack over by one element. This stack can now claim the freed index. */
98      if (stack.size >= stack.capacity) {
99        int nextStack = (stackNum + 1) % info.length;
100       shift(nextStack);
101       stack.capacity++; // claim index that next stack lost
102     }
103
104     /* Shift all elements in stack over by one. */
105     int index = stack.lastCapacityIndex();
106     while (stack.isWithinStackCapacity(index)) {
107       values[index] = values[previousIndex(index)];
108       index = previousIndex(index);
109     }
```

```
110
111     /* Adjust stack data. */
112     values[stack.start] = 0; // Clear item
113     stack.start = nextIndex(stack.start); // move start
114     stack.capacity--; // Shrink capacity
115   }
116
117   /* Expand stack by shifting over other stacks */
118   private void expand(int stackNum) {
119     shift((stackNum + 1) % info.length);
120     info[stackNum].capacity++;
121   }
122
123   /* Returns the number of items actually present in stack. */
124   public int numberOfElements() {
125     int size = 0;
126     for (StackInfo sd : info) {
127       size += sd.size;
128     }
129     return size;
130   }
131
132   /* Returns true is all the stacks are full. */
133   public boolean allStacksAreFull() {
134     return numberOfElements() == values.length;
135   }
136
137   /* Adjust index to be within the range of 0 -> length - 1. */
138   private int adjustIndex(int index) {
139     /* Java's mod operator can return neg values. For example, (-11 % 5) will
140      * return -1, not 4. We actually want the value to be 4 (since we're wrapping
141      * around the index). */
142     int max = values.length;
143     return ((index % max) + max) % max;
144   }
145
146   /* Get index after this index, adjusted for wrap around. */
147   private int nextIndex(int index) {
148     return adjustIndex(index + 1);
149   }
150
151   /* Get index before this index, adjusted for wrap around. */
152   private int previousIndex(int index) {
153     return adjustIndex(index - 1);
154   }
155 }
```

In problems like this, it's important to focus on writing clean, maintainable code. You should use additional classes, as we did with `StackInfo`, and pull chunks of code into separate methods. Of course, this advice applies to the "real world" as well.

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

pg 98

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 integers, 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 **Stack of Plates:** 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 99

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   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   int pop() {
2     Stack last = getLastStack();
3     if (last == null) throw new EmptyStackException();
4     int v = last.pop();
5     if (last.size == 0) stacks.remove(stacks.size() - 1);
6     return v;
7   }
```

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 okay 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 Queue via Stacks: Implement a MyQueue class which implements a queue using two stacks.

pg 99

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 elements on top */
15       stackNewest.push(value);
16    }
17
18    /* Move elements from stackNewest into stackOldest. This is usually done so that
19     * we can do operations on stackOldest. */
```

```
20    private void shiftStacks() {
21       if (stackOldest.isEmpty()) {
22          while (!stackNewest.isEmpty()) {
23             stackOldest.push(stackNewest.pop());
24          }
25       }
26    }
27
28    public T peek() {
29       shiftStacks(); // Ensure stackOldest has the current elements
30       return stackOldest.peek(); // retrieve the oldest item.
31    }
32
33    public T remove() {
34       shiftStacks(); // Ensure stackOldest has the current elements
35       return stackOldest.pop(); // pop the oldest item.
36    }
37 }
```

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.

3.5 Sort Stack: Write a program to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, 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 99

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`.

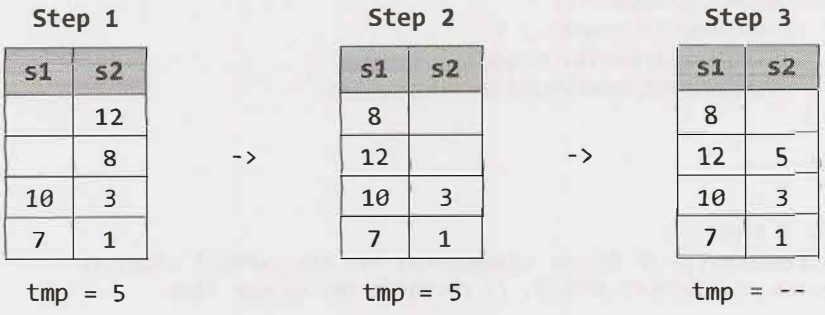
Unfortunately, this requires two additional stacks, and we can only use one. Can we do better? Yes.

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  void sort(Stack<Integer> s) {
2    Stack<Integer> r = new Stack<Integer>();
3    while(!s.isEmpty()) {
4      /* Insert each element in s in sorted order into r. */
5      int tmp = s.pop();
6      while(!r.isEmpty() && r.peek() > tmp) {
7        s.push(r.pop());
8      }
9      r.push(tmp);
10   }
11
12   /* Copy the elements from r back into s. */
13   while (!r.isEmpty()) {
14     s.push(r.pop());
15   }
16 }
```

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.

3.6 Animal Shelter: An animal shelter, which holds only dogs and cats, 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 99

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  abstract class Animal {
2    private int order;
3    protected String name;
4    public Animal(String n) { name = n; }
5    public void setOrder(int ord) { order = ord; }
6    public int getOrder() { return order; }
7
8    /* Compare orders of animals to return the older item. */
9    public boolean isOlderThan(Animal a) {
10     return this.order < a.getOrder();
11   }
12 }
13
14 class AnimalQueue {
15   LinkedList<Dog> dogs = new LinkedList<Dog>();
16   LinkedList<Cat> cats = new LinkedList<Cat>();
17   private int order = 0; // acts as timestamp
18
19   public void enqueue(Animal a) {
20     /* Order is used as a sort of timestamp, so that we can compare the insertion
21      * order of a dog to a cat. */
22     a.setOrder(order);
23     order++;
24
25     if (a instanceof Dog) dogs.addLast((Dog) a);
26     else if (a instanceof Cat) cats.addLast((Cat)a);
27   }
28
29   public Animal dequeueAny() {
30     /* Look at tops of dog and cat queues, and pop the queue with the oldest
31      * value. */
32     if (dogs.size() == 0) {
33       return dequeueCats();
34     } else if (cats.size() == 0) {
35       return dequeueDogs();
36     }
```

```
37
38        Dog dog = dogs.peek();
39        Cat cat = cats.peek();
40        if (dog.isOlderThan(cat)) {
41           return dequeueDogs();
42        } else {
43           return dequeueCats();
44        }
45     }
46
47     public Dog dequeueDogs() {
48        return dogs.poll();
49     }
50
51     public Cat dequeueCats() {
52        return cats.poll();
53     }
54  }
55
56  public class Dog extends Animal {
57     public Dog(String n) { super(n); }
58  }
59
60  public class Cat extends Animal {
61     public Cat(String n) { super(n); }
62  }
```

It is important that `Dog` and `Cat` both inherit from an `Animal` class since `dequeueAny()` needs to be able to support returning both `Dog` and `Cat` objects.

If we wanted, `order` could be a true timestamp with the actual date and time. The advantage of this is that we wouldn't have to set and maintain the numerical order. If we somehow wound up with two animals with the same timestamp, then (by definition) we don't have an older animal and we could return either one.

4

Solutions to Trees and Graphs

4.1 Route Between Nodes: Given a directed graph, design an algorithm to find out whether there is a route between two nodes.

pg 109

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  enum State { Unvisited, Visited, Visiting; }
2
3  boolean search(Graph g, Node start, Node end) {
4    if (start == end) return true;
5
6    // operates as Queue
7    LinkedList<Node> q = new LinkedList<Node>();
8
9    for (Node u : g.getNodes()) {
10     u.state = State.Unvisited;
11   }
12   start.state = State.Visiting;
13   q.add(start);
14   Node u;
15   while (!q.isEmpty()) {
16     u = q.removeFirst(); // i.e., dequeue()
17     if (u != null) {
18       for (Node v : u.getAdjacent()) {
19         if (v.state == State.Unvisited) {
20           if (v == end) {
21             return true;
22           } else {
23             v.state = State.Visiting;
24             q.add(v);
25           }
26         }
27       }
28       u.state = State.Visited;
29     }
```

```
30    }
31    return false;
32  }
```

It may be worth discussing with your interviewer the tradeoffs 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.

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

pg 109

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 array[]) {
2     return createMinimalBST(array, 0, array.length - 1);
3   }
4
5   TreeNode createMinimalBST(int arr[], int start, int end) {
6     if (end < start) {
7       return null;
8     }
9     int mid = (start + end) / 2;
10    TreeNode n = new TreeNode(arr[mid]);
11    n.left = createMinimalBST(arr, start, mid - 1);
12    n.right = createMinimalBST(arr, mid + 1, end);
13    return n;
```

```
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.

4.3 List of Depths: 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 109

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, ArrayList<LinkedList<TreeNode>> lists,
2                             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 first time we've
9       * visited level i, we must have seen levels 0 through i - 1. We can
10      * therefore safely add the level at the end. */
11     lists.add(list);
12   } else {
13     list = lists.get(level);
14   }
15   list.add(root);
16   createLevelLinkedList(root.left, lists, level + 1);
17   createLevelLinkedList(root.right, lists, level + 1);
18 }
19
20 ArrayList<LinkedList<TreeNode>> createLevelLinkedList(TreeNode root) {
21   ArrayList<LinkedList<TreeNode>> lists = new ArrayList<LinkedList<TreeNode>>();
22   createLevelLinkedList(root, lists, 0);
23   return lists;
24 }
```

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(TreeNode root) {
2    ArrayList<LinkedList<TreeNode>> result = new ArrayList<LinkedList<TreeNode>>();
3    /* "Visit" the root */
4    LinkedList<TreeNode> current = new LinkedList<TreeNode>();
5    if (root != null) {
6      current.add(root);
7    }
```

```
8
9     while (current.size() > 0) {
10       result.add(current); // Add previous level
11       LinkedList<TreeNode> parents = current; // Go to next level
12       current = new LinkedList<TreeNode>();
13       for (TreeNode parent : parents) {
14          /* Visit the children */
15          if (parent.left != null) {
16             current.add(parent.left);
17          }
18          if (parent.right != null) {
19             current.add(parent.right);
20          }
21       }
22    }
23    return result;
24 }
```

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 (in a balanced tree), 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."

4.4 **Check Balanced:** 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 110

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  int getHeight(TreeNode root) {
2     if (root == null) return -1; // Base case
3     return Math.max(getHeight(root.left), getHeight(root.right)) + 1;
4  }
5
6  boolean isBalanced(TreeNode root) {
7     if (root == null) return true; // Base case
8
9     int heightDiff = getHeight(root.left) - getHeight(root.right);
10    if (Math.abs(heightDiff) > 1) {
11       return false;
12    } else { // Recurse
13       return isBalanced(root.left) && isBalanced(root.right);
14    }
15 }
```

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 `O(N log N)` since each node is "touched" once per node above it.

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 at the same time as it's checking heights. What do we do when we discover that the subtree isn't balanced? Just return an error code.

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 an error code. We will immediately break and return an error code from the current call.

> What do we use for an error code? The height of a null tree is generally defined to be -1, so that's not a great idea for an error code. Instead, we'll use `Integer.MIN_VALUE`.

The code below implements this algorithm.

```
1  int checkHeight(TreeNode root) {
2    if (root == null) return -1;
3
4    int leftHeight = checkHeight(root.left);
5    if (leftHeight == Integer.MIN_VALUE) return Integer.MIN_VALUE; // Pass error up
6
7    int rightHeight = checkHeight(root.right);
8    if (rightHeight == Integer.MIN_VALUE) return Integer.MIN_VALUE; // Pass error up
9
10   int heightDiff = leftHeight - rightHeight;
11   if (Math.abs(heightDiff) > 1) {
12     return Integer.MIN_VALUE; // Found error -> pass it back
13   } else {
14     return Math.max(leftHeight, rightHeight) + 1;
15   }
16 }
17
18 boolean isBalanced(TreeNode root) {
19   return checkHeight(root) != Integer.MIN_VALUE;
20 }
```

This code runs in `O(N)` time and `O(H)` space, where H is the height of the tree.

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

pg 110

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 pseudo-code for this method looks something like:

```
1  int index = 0;
2  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 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  Integer last_printed = null;
2  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 (last_printed != null && n.data <= last_printed) {
10     return false;
11   }
12   last_printed = n.data;
13
14   // Check / recurse right
```

```
15    if (!checkBST(n.right)) return false;
16
17    return true; // All good!
18  }
```

We've used an `Integer` instead of `int` so that we can know when `last_printed` has been set to a value.

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.

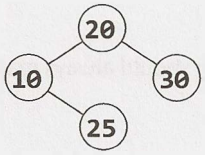
```
1   class WrapInt {
2     public int value;
3   }
```

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:

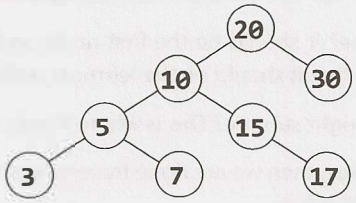


Although each node is bigger than its left node and smaller than its right node, this is clearly not a binary search tree since 25 is in the wrong place.

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 = NULL, max = NULL)`, which the root obviously meets. (`NULL` indicates that there is no min or max.) We then branch left, checking that these nodes are within the range `(min = NULL, max = 20)`. Then, we branch right, checking that the nodes are within the range `(min = 20, max = NULL)`.

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, null, null);
3  }
4
5  boolean checkBST(TreeNode n, Integer min, Integer max) {
6    if (n == null) {
7      return true;
8    }
9    if ((min != null && n.data <= min) || (max != null && n.data > max)) {
10     return false;
11   }
12
13   if (!checkBST(n.left, min, n.data) || !checkBST(n.right, n.data, max)) {
14     return false;
15   }
16   return true;
17 }
```

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 Successor: 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 110

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:

```
Node inorderSucc(Node n) {
   if (n has a right subtree) {
      return leftmost child of right subtree
   } else {
      while (n is a right child of n.parent) {
         n = n.parent; // Go up
      }
      return n.parent; // Parent has not been traversed
   }
}
```

But wait—what if we traverse all the way up the tree before finding a left child? This will happen only when we 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).

```
TreeNode inorderSucc(TreeNode n) {
   if (n == null) return null;

   /* Found right children -> return leftmost node of right subtree. */
   if (n.right != null) {
      return leftMostChild(n.right);
   } else {
      TreeNode q = n;
      TreeNode x = q.parent;
      // Go up until we're on left instead of right
      while (x != null && x.left != q) {
         q = x;
         x = x.parent;
      }
      return x;
   }
}

TreeNode leftMostChild(TreeNode n) {
   if (n == null) {
      return null;
   }
   while (n.left != null) {
      n = n.left;
   }
   return n;
}
```

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 **Build Order:** You are given a list of projects and a list of dependencies (which is a list of pairs of projects, where the second project is dependent on the first project). All of a project's dependencies must be built before the project is. Find a build order that will allow the projects to be built. If there is no valid build order, return an error.

EXAMPLE

Input:

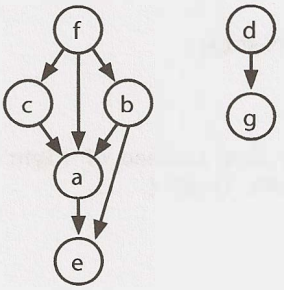
```
projects: a, b, c, d, e, f
dependencies: (a, d), (f, b), (b, d), (f, a), (d, c)
```

Output: `f, e, a, b, d, c`

pg 110

SOLUTION

Visualizing the information as a graph probably works best. Be careful with the direction of the arrows. In the graph below, an arrow from d to g means that d must be compiled before g. You can also draw them in the opposite direction, but you need to consistent and clear about what you mean. Let's draw a fresh example.



In drawing this example (which is *not* the example from the problem description), I looked for a few things.

- I wanted the nodes labeled somewhat randomly. If I had instead put a at the top, with b and c as children, then d and e, it could be misleading. The alphabetical order would match the compile order.
- I wanted a graph with multiple parts/components, since a connected graph is a bit of a special case.
- I wanted a graph where a node links to a node that cannot immediately follow it. For example, f links to a but a cannot immediately follow it (since b and c must come before a and after f).
- I wanted a larger graph since I need to figure out the pattern.
- I wanted nodes with multiple dependencies.

Now that we have a good example, let's get started with an algorithm.

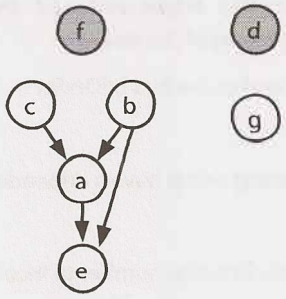
Solution #1

Where do we start? Are there any nodes that we can definitely compile immediately?

Yes. Nodes with no incoming edges can be built immediately since they don't depend on anything. Let's add all such nodes to the build order. In the earlier example, this means we have an order of f, d (or d, f).

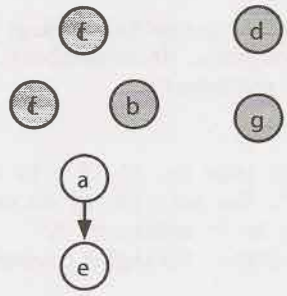
Once we've done that, it's irrelevant that some nodes are dependent on d and f since d and f have already been built. We can reflect this new state by removing d and f's outgoing edges.

```
build order: f, d
```



Next, we know that c, b, and g are free to build since they have no incoming edges. Let's build those and then remove their outgoing edges.

```
build order: f, d, c, b, g
```



Project a can be built next, so let's do that and remove its outgoing edges. This leaves just e. We build that next, giving us a complete build order.

```
build order: f, d, c, b, g, a, e
```

Did this algorithm work, or did we just get lucky? Let's think about the logic.

1. We first added the nodes with no incoming edges. If the set of projects can be built, there must be some "first" project, and that project can't have any dependencies. If a project has no dependencies (incoming edges), then we certainly can't break anything by building it first.
2. We removed all outgoing edges from these roots. This is reasonable. Once those root projects were built, it doesn't matter if another project depends on them.
3. After that, we found the nodes that *now* have no incoming edges. Using the same logic from steps 1 and 2, it's okay if we build these. Now we just repeat the same steps: find the nodes with no dependencies, add them to the build order, remove their outgoing edges, and repeat.
4. What if there are nodes remaining, but all have dependencies (incoming edges)? This means there's no way to build the system. We should return an error.

The implementation follows this approach very closely.

Initialization and setup:

1. Build a graph where each project is a node and its outgoing edges represent the projects that depend on it. That is, if A has an edge to B (A -> B), it means B has a dependency on A and therefore A must be built before B. Each node also tracks the number of *incoming* edges.
2. Initialize a `buildOrder` array. Once we determine a project's build order, we add it to the array. We also continue to iterate through the array, using a `toBeProcessed` pointer to point to the next node to be fully processed.

3. Find all the nodes with zero incoming edges and add those to a `buildOrder` array. Set a `toBeProcessed` pointer to the beginning of the array.

Repeat until `toBeProcessed` is at the end of the `buildOrder`:

1. Read node at `toBeProcessed`.
    - If `node` is `null`, then all remaining nodes have a dependency and we have detected a cycle.
2. For each child of node:
    - Decrement `child.dependencies` (the number of incoming edges).
    - If `child.dependencies` is zero, add `child` to end of `buildOrder`.
3. Increment `toBeProcessed`.

The code below implements this algorithm.

```
1  /* Find a correct build order. */
2  Project[] findBuildOrder(String[] projects, String[][] dependencies) {
3    Graph graph = buildGraph(projects, dependencies);
4    return orderProjects(graph.getNodes());
5  }
6
7  /* Build the graph, adding the edge (a, b) if b is dependent on a. Assumes a pair
8   * is listed in "build order". The pair (a, b) in dependencies indicates that b
9   * depends on a and a must be built before b. */
10 Graph buildGraph(String[] projects, String[][] dependencies) {
11   Graph graph = new Graph();
12   for (String project : projects) {
13     graph.createNode(project);
14   }
15
16   for (String[] dependency : dependencies) {
17     String first = dependency[0];
18     String second = dependency[1];
19     graph.addEdge(first, second);
20   }
21
22   return graph;
23 }
24
25 /* Return a list of the projects a correct build order.*/
26 Project[] orderProjects(ArrayList<Project> projects) {
27   Project[] order = new Project[projects.size()];
28
29   /* Add "roots" to the build order first.*/
30   int endOfList = addNonDependent(order, projects, 0);
31
32   int toBeProcessed = 0;
33   while (toBeProcessed < order.length) {
34     Project current = order[toBeProcessed];
35
36     /* We have a circular dependency since there are no remaining projects with
37      * zero dependencies. */
38     if (current == null) {
39       return null;
40     }
41
```

```
      /* Remove myself as a dependency. */
      ArrayList<Project> children = current.getChildren();
      for (Project child : children) {
         child.decrementDependencies();
      }

      /* Add children that have no one depending on them. */
      endOfList = addNonDependent(order, children, endOfList);
      toBeProcessed++;
   }

   return order;
}

/* A helper function to insert projects with zero dependencies into the order
 * array, starting at index offset. */
int addNonDependent(Project[] order, ArrayList<Project> projects, int offset) {
   for (Project project : projects) {
      if (project.getNumberDependencies() == 0) {
         order[offset] = project;
         offset++;
      }
   }
   return offset;
}

public class Graph {
   private ArrayList<Project> nodes = new ArrayList<Project>();
   private HashMap<String, Project> map = new HashMap<String, Project>();

   public Project getOrCreateNode(String name) {
      if (!map.containsKey(name)) {
         Project node = new Project(name);
         nodes.add(node);
         map.put(name, node);
      }

      return map.get(name);
   }

   public void addEdge(String startName, String endName) {
      Project start = getOrCreateNode(startName);
      Project end = getOrCreateNode(endName);
      start.addNeighbor(end);
   }

   public ArrayList<Project> getNodes() { return nodes; }
}

public class Project {
   private ArrayList<Project> children = new ArrayList<Project>();
   private HashMap<String, Project> map = new HashMap<String, Project>();
   private String name;
   private int dependencies = 0;

   public Project(String n) { name = n; }
```

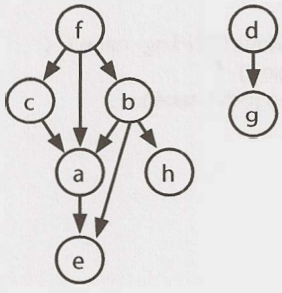
```
98
99    public void addNeighbor(Project node) {
100     if (!map.containsKey(node.getName())) {
101       children.add(node);
102       map.put(node.getName(), node);
103       node.incrementDependencies();
104     }
105   }
106
107   public void incrementDependencies() { dependencies++; }
108   public void decrementDependencies() { dependencies--; }
109
110   public String getName() { return name; }
111   public ArrayList<Project> getChildren() { return children; }
112   public int getNumberDependencies() { return dependencies; }
113 }
```

This solution takes O(P + D) time, where P is the number of projects and D is the number of dependency pairs.

Note: You might recognize this as the topological sort algorithm on page 632. We've rederived this from scratch. Most people won't know this algorithm and it's reasonable for an interviewer to expect you to be able to derive it.

Solution #2

Alternatively, we can use depth-first search (DFS) to find the build path.



Suppose we picked an arbitrary node (say b) and performed a depth-first search on it. When we get to the end of a path and can't go any further (which will happen at h and e), we know that those terminating nodes can be the last projects to be built. No projects depend on them.

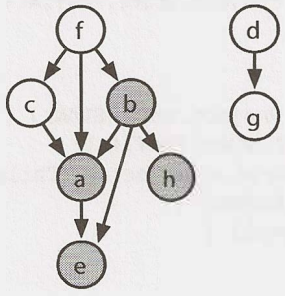
```
DFS(b)                              // Step 1
    DFS(h)                          // Step 2
        build order = ..., h        // Step 3
    DFS(a)                          // Step 4
        DFS(e)                      // Step 5
            build order = ..., e, h // Step 6
        ...                         // Step 7+
    ...
```

Now, consider what happens at node a when we return from the DFS of e. We know a's children need to appear after a in the build order. So, once we return from searching a's children (and therefore they have been added), we can choose to add a to the front of the build order.

Once we return from a, and complete the DFS of b's other children, then everything that must appear after b is in the list. Add b to the front.

```
DFS(b)                                  // Step 1
    DFS(h)                              // Step 2
        build order = ..., h            // Step 3
    DFS(a)                              // Step 4
        DFS(e)                          // Step 5
            build order = ..., e, h     // Step 6
        build order = ..., a, e, h      // Step 7
    DFS(e) -> return                    // Step 8
    build order = ..., b, a, e, h       // Step 9
```

Let's mark these nodes as having been built too, just in case someone else needs to build them.



Now what? We can start with any old node again, doing a DFS on it and then adding the node to the front of the build queue when the DFS is completed.

```
DFS(d)
    DFS(g)
        build order = ..., g, b, a, e, h
    build order = ..., d, g, b, a, e, h

DFS(f)
    DFS(c)
        build order = ..., c, d, g, b, a, e, h
    build order = f, c, d, g, b, a, e, h
```

In an algorithm like this, we should think about the issue of cycles. There is no possible build order if there is a cycle. But still, we don't want to get stuck in an infinite loop just because there's no possible solution.

A cycle will happen if, while doing a DFS on a node, we run back into the same path. What we need therefore is a signal that indicates "I'm still processing this node, so if you see the node again, we have a problem."

What we can do for this is to mark each node as a "`partial`" (or "`is visiting`") state just before we start the DFS on it. If we see any node whose state is `partial`, then we know we have a problem. When we're done with this node's DFS, we need to update the state.

We also need a state to indicate "I've already processed/built this node" so we don't re-build the node. Our state therefore can have three options: `COMPLETED`, `PARTIAL`, and `BLANK`.

The code below implements this algorithm.

```
1  Stack<Project> findBuildOrder(String[] projects, String[][] dependencies) {
2    Graph graph = buildGraph(projects, dependencies);
3    return orderProjects(graph.getNodes());
4  }
```

```
5
6  Stack<Project> orderProjects(ArrayList<Project> projects) {
7     Stack<Project> stack = new Stack<Project>();
8     for (Project project : projects) {
9        if (project.getState() == Project.State.BLANK) {
10          if (!doDFS(project, stack)) {
11             return null;
12          }
13       }
14    }
15    return stack;
16 }
17
18 boolean doDFS(Project project, Stack<Project> stack) {
19    if (project.getState() == Project.State.PARTIAL) {
20       return false; // Cycle
21    }
22
23    if (project.getState() == Project.State.BLANK) {
24       project.setState(Project.State.PARTIAL);
25       ArrayList<Project> children = project.getChildren();
26       for (Project child : children) {
27          if (!doDFS(child, stack)) {
28             return false;
29          }
30       }
31       project.setState(Project.State.COMPLETE);
32       stack.push(project);
33    }
34    return true;
35 }
36
37 /* Same as before */
38 Graph buildGraph(String[] projects, String[][] dependencies) {...}
39 public class Graph {}
40
41 /* Essentially equivalent to earlier solution, with state info added and
42  * dependency count removed. */
43 public class Project {
44    public enum State {COMPLETE, PARTIAL, BLANK};
45    private State state = State.BLANK;
46    public State getState() { return state; }
47    public void setState(State st) { state = st; }
48    /* Duplicate code removed for brevity */
49 }
```

Like the earlier algorithm, this solution is O(P+D) time, where P is the number of projects and D is the number of dependency pairs.

By the way, this problem is called **topological sort**: linearly ordering the vertices in a graph such that for every edge (a, b), a appears before b in the linear order.

4.8 First Common Ancestor: 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 110

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 each node in our tree has a link 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. This is essentially the same problem as question 2.7 which find the intersection of two linked lists. The "linked list" in this case is the path from each node up to the root. (Review this solution on page 221.)

```
1  TreeNode commonAncestor(TreeNode p, TreeNode q) {
2    int delta = depth(p) - depth(q); // get difference in depths
3    TreeNode first = delta > 0 ? q : p; // get shallower node
4    TreeNode second = delta > 0 ? p : q; // get deeper node
5    second = goUpBy(second, Math.abs(delta)); // move deeper node up
6
7    /* Find where paths intersect. */
8    while (first != second && first != null && second != null) {
9      first = first.parent;
10     second = second.parent;
11   }
12   return first == null || second == null ? null : first;
13 }
14
15 TreeNode goUpBy(TreeNode node, int delta) {
16   while (delta > 0 && node != null) {
17     node = node.parent;
18     delta--;
19   }
20   return node;
21 }
22
23 int depth(TreeNode node) {
24   int depth = 0;
25   while (node != null) {
26     node = node.parent;
27     depth++;
28   }
29   return depth;
30 }
```

This approach will take `O(d)` time, where d is the depth of the deeper node.

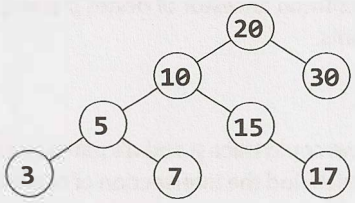
Solution #2: With Links to Parents (Better Worst-Case Runtime)

Similar to the earlier approach, we could trace p's path upwards and check if any of the nodes cover q. The first node that covers q (we already know that every node on this path will cover p) must be the first common ancestor.

Observe that we don't need to re-check the entire subtree. As we move from a node x to its parent y, all the nodes under x have already been checked for q. Therefore, we only need to check the new nodes "uncovered", which will be the nodes under x's sibling.

For example, suppose we're looking for the first common ancestor of node `p = 7` and node `q = 17`. When we go to `p.parent` (5), we uncover the subtree rooted at 3. We therefore need to search this subtree for q.

Next, we go to node 10, uncovering the subtree rooted at 15. We check this subtree for node 17 and—voila—there it is.



To implement this, we can just traverse upwards from p, storing the `parent` and the `sibling` node in a variable. (The `sibling` node is always a child of `parent` and refers to the newly uncovered subtree.) At each iteration, `sibling` gets set to the old `parent`'s sibling node and `parent` gets set to `parent.parent`.

```
1  TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
2    /* Check if either node is not in the tree, or if one covers the other. */
3    if (!covers(root, p) || !covers(root, q)) {
4      return null;
5    } else if (covers(p, q)) {
6      return p;
7    } else if (covers(q, p)) {
8      return q;
9    }
10
11   /* Traverse upwards until you find a node that covers q. */
12   TreeNode sibling = getSibling(p);
13   TreeNode parent = p.parent;
14   while (!covers(sibling, q)) {
15     sibling = getSibling(parent);
16     parent = parent.parent;
17   }
18   return parent;
19 }
20
21 boolean covers(TreeNode root, TreeNode p) {
22   if (root == null) return false;
23   if (root == p) return true;
24   return covers(root.left, p) || covers(root.right, p);
25 }
26
27 TreeNode getSibling(TreeNode node) {
28   if (node == null || node.parent == null) {
29     return null;
30   }
31
32   TreeNode parent = node.parent;
```

```
33    return parent.left == node ? parent.right : parent.left;
34  }
```

This algorithm takes O(t) time, where t is the size of the subtree for the first common ancestor. In the worst case, this will be O(n), where n is the number of nodes in the tree. We can derive this runtime by noticing that each node in that subtree is searched once.

Solution #3: 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   TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
2     /* Error check - one node is not in the tree. */
3     if (!covers(root, p) || !covers(root, q)) {
4       return null;
5     }
6     return ancestorHelper(root, p, q);
7   }
8
9   TreeNode ancestorHelper(TreeNode root, TreeNode p, TreeNode q) {
10    if (root == null || root == p || root == q) {
11      return root;
12    }
13
14    boolean pIsOnLeft = covers(root.left, p);
15    boolean qIsOnLeft = covers(root.left, q);
16    if (pIsOnLeft != qIsOnLeft) { // Nodes are on different side
17      return root;
18    }
19    TreeNode childSide = pIsOnLeft ? root.left : root.right;
20    return ancestorHelper(childSide, p, q);
21  }
22
23  boolean covers(TreeNode root, TreeNode p) {
24    if (root == null) return false;
25    if (root == p) return true;
26    return covers(root.left, p) || covers(root.right, p);
27  }
```

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 #4: Optimized

Although Solution #3 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:

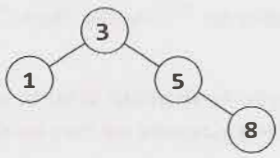
- Returns p, if `root`'s subtree includes p (and not q).
- Returns q, if `root`'s subtree includes q (and not p).
- Returns `null`, if neither p nor q are in `root`'s subtree.
- Else, returns the common ancestor of p and q.

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 commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
3    if (root == null) return null;
4    if (root == p && root == q) return root;
5
6    TreeNode x = commonAncestor(root.left, p, q);
7    if (x != null && x != p && x != q) { // Already found ancestor
8      return x;
9    }
10
11   TreeNode y = commonAncestor(root.right, p, q);
12   if (y != null && y != p && y != q) { // Already found ancestor
13     return y;
14   }
15
16   if (x != null && y != null) { // p and q found in diff. subtrees
17     return root; // This is the common ancestor
18   } else if (root == p || root == q) {
19     return root;
20   } else {
21     return x == null ? y : x; /* return the non-null value */
22   }
23 }
```

The problem with this code occurs in the case where a node is not contained in the tree. For example, look at the following tree:



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  commonAnc(node 3, node 5, node 7)              // --> 5
2    calls commonAnc(node 1, node 5, node 7)      // --> null
```

```
3      calls commonAnc(node 5, node 5, node 7)      // --> 5
4        calls commonAnc(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   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  TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
11    Result r = commonAncestorHelper(root, p, q);
12    if (r.isAncestor) {
13      return r.node;
14    }
15    return null;
16  }
17
18  Result commonAncHelper(TreeNode root, TreeNode p, TreeNode q) {
19    if (root == null) return new Result(null, false);
20
21    if (root == p && root == q) {
22      return new Result(root, true);
23    }
24
25    Result rx = commonAncHelper(root.left, p, q);
26    if (rx.isAncestor) { // Found common ancestor
27      return rx;
28    }
29
30    Result ry = commonAncHelper(root.right, p, q);
31    if (ry.isAncestor) { // Found common ancestor
32      return ry;
33    }
34
35    if (rx.node != null && ry.node != null) {
36      return new Result(root, true); // This is the common ancestor
37    } else if (root == p || root == q) {
38      /* If we're currently at p or q, and we also found one of those nodes in a
39       * subtree, then this is truly an ancestor and the flag should be true. */
40      boolean isAncestor = rx.node != null || ry.node != null;
```

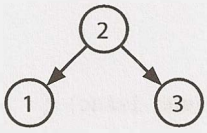
```
41        return new Result(root, isAncestor);
42    } else {
43        return new Result(rx.node!=null ? rx.node : ry.node, false);
44    }
45 }
```

Of course, as this issue only comes up when 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.

4.9 **BST Sequences:** A binary search tree was created by traversing through an array from left to right and inserting each element. Given a binary search tree with distinct elements, print all possible arrays that could have led to this tree.

EXAMPLE

Input:

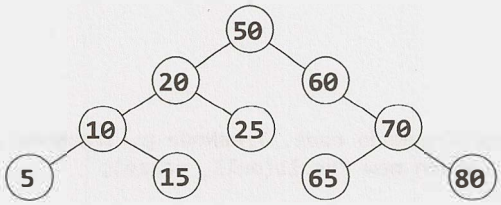


Output: {2, 1, 3},{2, 3, 1}

pg 110

SOLUTION

It's useful to kick off this question with a good example.



We should also think about the ordering of items in a binary search tree. Given a node, all nodes on its left must be less than all nodes on its right. Once we reach a place without a node, we insert the new value there.

What this means is that the very first element in our array must have been a 50 in order to create the above tree. If it were anything else, then that value would have been the root instead.

What else can we say? Some people jump to the conclusion that everything on the left must have been inserted before elements on the right, but that's not actually true. In fact, the reverse is true: the order of the left or right items doesn't matter.

Once the 50 is inserted, all items less than 50 will be routed to the left and all items greater than 50 will be routed to the right. The 60 or the 20 could be inserted first, and it wouldn't matter.

Let's think about this problem recursively. If we had all arrays that could have created the subtree rooted at 20 (call this `arraySet20`), and all arrays that could have created the subtree rooted at 60 (call this `arraySet60`), how would that give us the full answer? We could just "weave" each array from `arraySet20` with each array from `arraySet60`—and then prepend each array with a 50.

Here's what we mean by weaving. We are merging two arrays in all possible ways, while keeping the elements within each array in the same relative order.

```
array1: {1, 2}
array2: {3, 4}
weaved: {1, 2, 3, 4}, {1, 3, 2, 4}, {1, 3, 4, 2},
        {3, 1, 2, 4}, {3, 1, 4, 2}, {3, 4, 1, 2}
```

Note that, as long as there aren't any duplicates in the original array sets, we won't have to worry that weaving will create duplicates.

The last piece to talk about here is how the weaving works. Let's think recursively about how to weave `{1, 2, 3}` and `{4, 5, 6}`. What are the subproblems?

- Prepend a 1 to all weaves of `{2, 3}` and `{4, 5, 6}`.
- Prepend a 4 to all weaves of `{1, 2, 3}` and `{5, 6}`.

To implement this, we'll store each as linked lists. This will make it easy to add and remove elements. When we recurse, we'll push the prefixed elements down the recursion. When `first` or `second` are empty, we add the remainder to `prefix` and store the result.

It works something like this:

```
weave(first, second, prefix):
    weave({1, 2}, {3, 4}, {})
        weave({2}, {3, 4}, {1})
            weave({}, {3, 4}, {1, 2})
                {1, 2, 3, 4}
            weave({2}, {4}, {1, 3})
                weave({}, {4}, {1, 3, 2})
                    {1, 3, 2, 4}
                weave({2}, {}, {1, 3, 4})
                    {1, 3, 4, 2}
        weave({1, 2}, {4}, {3})
            weave({2}, {4}, {3, 1})
                weave({}, {4}, {3, 1, 2})
                    {3, 1, 2, 4}
                weave({2}, {}, {3, 1, 4})
                    {3, 1, 4, 2}
            weave({1, 2}, {}, {3, 4})
                {3, 4, 1, 2}
```

Now, let's think through the implementation of removing, say, 1 from `{1, 2}` and recursing. We need to be careful about modifying this list, since a later recursive call (e.g., `weave({1, 2}, {4}, {3})`) might need the 1 still in `{1, 2}`.

We could clone the list when we recurse, so that we only modify the recursive calls. Or, we could modify the list, but then "revert" the changes after we're done with recursing.

We've chosen to implement it the latter way. Since we're keeping the same reference to `first`, `second`, and `prefix` the entire way down the recursive call stack, then we'll need to clone `prefix` just before we store the complete result.

```
1  ArrayList<LinkedList<Integer>> allSequences(TreeNode node) {
2    ArrayList<LinkedList<Integer>> result = new ArrayList<LinkedList<Integer>>();
3
4    if (node == null) {
5      result.add(new LinkedList<Integer>());
6      return result;
```

```
    }

    LinkedList<Integer> prefix = new LinkedList<Integer>();
    prefix.add(node.data);

    /* Recurse on left and right subtrees. */
    ArrayList<LinkedList<Integer>> leftSeq = allSequences(node.left);
    ArrayList<LinkedList<Integer>> rightSeq = allSequences(node.right);

    /* Weave together each list from the left and right sides. */
    for (LinkedList<Integer> left : leftSeq) {
      for (LinkedList<Integer> right : rightSeq) {
        ArrayList<LinkedList<Integer>> weaved =
          new ArrayList<LinkedList<Integer>>();
        weaveLists(left, right, weaved, prefix);
        result.addAll(weaved);
      }
    }
    return result;
  }

  /* Weave lists together in all possible ways. This algorithm works by removing the
   * head from one list, recursing, and then doing the same thing with the other
   * list. */
  void weaveLists(LinkedList<Integer> first, LinkedList<Integer> second,
        ArrayList<LinkedList<Integer>> results, LinkedList<Integer> prefix) {
    /* One list is empty. Add remainder to [a cloned] prefix and store result. */
    if (first.size() == 0 || second.size() == 0) {
      LinkedList<Integer> result = (LinkedList<Integer>) prefix.clone();
      result.addAll(first);
      result.addAll(second);
      results.add(result);
      return;
    }

    /* Recurse with head of first added to the prefix. Removing the head will damage
     * first, so we'll need to put it back where we found it afterwards. */
    int headFirst = first.removeFirst();
    prefix.addLast(headFirst);
    weaveLists(first, second, results, prefix);
    prefix.removeLast();
    first.addFirst(headFirst);

    /* Do the same thing with second, damaging and then restoring the list.*/
    int headSecond = second.removeFirst();
    prefix.addLast(headSecond);
    weaveLists(first, second, results, prefix);
    prefix.removeLast();
    second.addFirst(headSecond);
  }
```

Some people struggle with this problem because there are two different recursive algorithms that must be designed and implemented. They get confused with how the algorithms should interact with each other and they try to juggle both in their heads.

If this sounds like you, try this: *trust and focus*. *Trust* that one method does the right thing when implementing an independent method, and *focus* on the one thing that this independent method needs to do.

Look at `weaveLists`. It has a specific job: to weave two lists together and return a list of all possible weaves. The existence of `allSequences` is irrelevant. Focus on the task that `weaveLists` has to do and design this algorithm.

As you're implementing `allSequences` (whether you do this before or after `weaveLists`), trust that `weaveLists` will do the right thing. Don't concern yourself with the particulars of how `weaveLists` operates while implementing something that is essentially independent. Focus on what you're doing while you're doing it.

In fact, this is good advice in general when you're confused during whiteboard coding. Have a good understanding of what a particular function should do ("okay, this function is going to return a list of ___"). You should verify that it's really doing what you think. But when you're not dealing with that function, focus on the one you are dealing with and trust that the others do the right thing. It's often too much to keep the implementations of multiple algorithms straight in your head.

4.10 Check Subtree: T1 and T2 are two very large binary trees, with T1 much bigger than T2. Create an algorithm to determine 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 111

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.

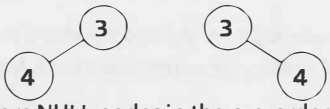
The Simple Approach

In this smaller, simpler problem, we could consider comparing string representations of traversals of each tree. If T2 is a subtree of T1, then T2's traversal should be a substring of T1. Is the reverse true? If so, should we use an in-order traversal or a pre-order traversal?

An in-order traversal will definitely not work. After all, consider a scenario in which we were using binary search trees. A binary search tree's in-order traversal always prints out the values in sorted order. Therefore, two binary search trees with the same values will always have the same in-order traversals, even if their structure is different.

What about a pre-order traversal? This is a bit more promising. At least in this case we know certain things, like the first element in the pre-order traversal is the root node. The left and right elements will follow.

Unfortunately, trees with different structures could still have the same pre-order traversal.

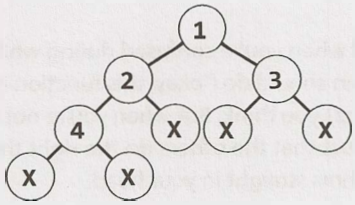


There's a simple fix though. We can store NULL nodes in the pre-order traversal string as a special character, like an 'X'. (We'll assume that the binary trees contain only integers.) The left tree would have the traversal `{3, 4, X}` and the right tree will have the traversal `{3, X, 4}`.

Observe that, as long as we represent the NULL nodes, the pre-order traversal of a tree is unique. That is, if two trees have the same pre-order traversal, then we know they are identical trees in values and structure.

To see this, consider reconstructing a tree from its pre-order traversal (with NULL nodes indicated). For example: 1, 2, 4, X, X, X, 3, X, X.

The root is 1, and its left node, 2, follows it. 2.left must be 4. 4 must have two NULL nodes (since it is followed by two Xs). 4 is complete, so we move back up to its parent, 2. 2.right is another X (NULL). 1's left subtree is now complete, so we move to 1's right child. We place a 3 with two NULL children there. The tree is now complete.



This whole process was deterministic, as it will be on any other tree. A pre-order traversal always starts at the root and, from there, the path we take is entirely defined by the traversal. Therefore, two trees are identical if they have the same pre-order traversal.

Now consider the subtree problem. If T2's pre-order traversal is a substring of T1's pre-order traversal, then T2's root element must be found in T1. If we do a pre-order traversal from this element in T1, we will follow an identical path to T2's traversal. Therefore, T2 is a subtree of T1.

Implementing this is quite straightforward. We just need to construct and compare the pre-order traversals.

```
1  boolean containsTree(TreeNode t1, TreeNode t2) {
2     StringBuilder string1 = new StringBuilder();
3     StringBuilder string2 = new StringBuilder();
4
5     getOrderString(t1, string1);
6     getOrderString(t2, string2);
7
8     return string1.indexOf(string2.toString()) != -1;
9  }
10
11 void getOrderString(TreeNode node, StringBuilder sb) {
12    if (node == null) {
13       sb.append("X");              // Add null indicator
14       return;
15    }
16    sb.append(node.data + " ");     // Add root
17    getOrderString(node.left, sb);  // Add left
18    getOrderString(node.right, sb); // Add right
19 }
```

This approach takes `O(n + m)` time and `O(n + m)` space, where `n` and `m` are the number of nodes in T1 and T2, respectively. Given millions of nodes, we might want to reduce the space complexity.

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 `matchTree`. The `matchTree` 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 `matchTree` on every node in T1. 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 `matchTree` when we find a difference between T1 and T2. We therefore probably do not actually look at m nodes on each call of `matchTree`.

The code below implements this algorithm.

```
1  boolean containsTree(TreeNode t1, TreeNode t2) {
2    if (t2 == null) return true; // The empty tree is always a subtree
3    return subTree(t1, t2);
4  }
5
6  boolean subTree(TreeNode r1, TreeNode r2) {
7    if (r1 == null) {
8      return false; // big tree empty & subtree still not found.
9    } else if (r1.data == r2.data && matchTree(r1, r2)) {
10     return true;
11   }
12   return subTree(r1.left, r2) || subTree(r1.right, r2);
13 }
14
15 boolean matchTree(TreeNode r1, TreeNode r2) {
16   if (r1 == null && r2 == null) {
17     return true; // nothing left in the subtree
18   } else if (r1 == null || r2 == null) {
19     return false; // exactly tree is empty, therefore trees don't match
20   } else if (r1.data != r2.data) {
21     return false;  // data doesn't match
22   } else {
23     return matchTree(r1.left, r2.left) && matchTree(r1.right, r2.right);
24   }
25 }
```

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:

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 ($1100000 = 1000000 + \frac{100*1000000}{1000}$).
4. More complex mathematics and assumptions could get us an even tighter bound. We assumed in #3 above that if we call `matchTree`, we would 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

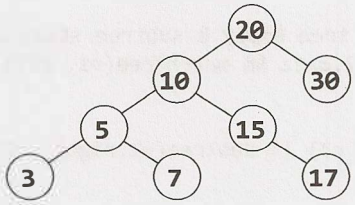
the average case runtime at the expense of the worst case runtime. This is an excellent point to make to your interviewer.

4.11 Random Node: You are implementing a binary search tree class from scratch, which, in addition to `insert`, `find`, and `delete`, has a method `getRandomNode()` which returns a random node from the tree. All nodes should be equally likely to be chosen. Design and implement an algorithm for `getRandomNode`, and explain how you would implement the rest of the methods.

pg 111

SOLUTION

Let's draw an example.



We're going to explore many solutions until we get to an optimal one that works.

One thing we should realize here is that the question was phrased in a very interesting way. The interviewer did not simply say, "Design an algorithm to return a random node from a binary tree." We were told that this is a class that we're building from scratch. There is a reason the question was phrased that way. We probably need access to some part of the internals of the data structure.

Option #1 [Slow & Working]

One solution is to copy all the nodes to an array and return a random element in the array. This solution will take `O(N)` time and `O(N)` space, where `N` is the number of nodes in the tree.

We can guess our interviewer is probably looking for something more optimal, since this is a little too straightforward (and should make us wonder why the interviewer gave us a binary tree, since we don't need that information).

We should keep in mind as we develop this solution that we probably need to know something about the internals of the tree. Otherwise, the question probably wouldn't specify that we're developing the tree class from scratch.

Option #2 [Slow & Working]

Returning to our original solution of copying the nodes to an array, we can explore a solution where we maintain an array at all times that lists all the nodes in the tree. The problem is that we'll need to remove nodes from this array as we delete them from the tree, and that will take `O(N)` time.

Option #3 [Slow & Working]

We could label all the nodes with an index from 1 to `N` and label them in binary search tree order (that is, according to its inorder traversal). Then, when we call `getRandomNode`, we generate a random index between 1 and `N`. If we apply the label correctly, we can use a binary search tree search to find this index.

However, this leads to a similar issue as earlier solutions. When we insert a node or a delete a node, all of the indices might need to be updated. This can take O(N) time.

Option #4 [Fast & Not Working]

What if we knew the depth of the tree? (Since we're building our own class, we can ensure that we know this. It's an easy enough piece of data to track.)

We could pick a random depth, and then traverse left/right randomly until we go to that depth. This wouldn't actually ensure that all nodes are equally likely to be chosen though.

First, the tree doesn't necessarily have an equal number of nodes at each level. This means that nodes on levels with fewer nodes might be more likely to be chosen than nodes on a level with more nodes.

Second, the random path we take might end up terminating before we get to the desired level. Then what? We could just return the last node we find, but that would mean unequal probabilities at each node.

Option #5 [Fast & Not Working]

We could try just a simple approach: traverse randomly down the tree. At each node:

- With $1/3$ odds, we return the current node.
- With $1/3$ odds, we traverse left.
- With $1/3$ odds, we traverse right.

This solution, like some of the others, does not distribute the probabilities evenly across the nodes. The root has a $1/3$ probability of being selected—the same as all the nodes in the left put together.

Option #6 [Fast & Working]

Rather than just continuing to brainstorm new solutions, let's see if we can fix some of the issues in the previous solutions. To do so, we must diagnose—deeply—the root problem in a solution.

Let's look at Option #5. It fails because the probabilities aren't evenly distributed across the options. Can we fix that while keeping the basic algorithm the same?

We can start with the root. With what probability should we return the root? Since we have N nodes, we must return the root node with $1/N$ probability. (In fact, we must return each node with $1/N$ probability. After all, we have N nodes and each must have equal probability. The total must be 1 (100%), therefore each must have $1/N$ probability.)

We've resolved the issue with the root. Now what about the rest of the problem? With what probability should we traverse left versus right? It's not 50/50. Even in a balanced tree, the number of nodes on each side might not be equal. If we have more nodes on the left than the right, then we need to go left more often.

One way to think about it is that the odds of picking something—anything—from the left must be the sum of each individual probability. Since each node must have probability $1/N$, the odds of picking something from the left must have probability `LEFT_SIZE` * $1/N$. This should therefore be the odds of going left.

Likewise, the odds of going right should be `RIGHT_SIZE` * $1/N$.

This means that each node must know the size of the nodes on the left and the size of the nodes on the right. Fortunately, our interviewer has told us that we're building this tree class from scratch. It's easy to keep track of this size information on inserts and deletes. We can just store a `size` variable in each node. Increment `size` on inserts and decrement it on deletes.

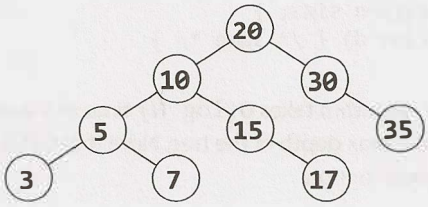
```
1  class TreeNode {
2    private int data;
3    public TreeNode left;
4    public TreeNode right;
5    private int size = 0;
6
7    public TreeNode(int d) {
8      data = d;
9      size = 1;
10   }
11
12   public TreeNode getRandomNode() {
13     int leftSize = left == null ? 0 : left.size();
14     Random random = new Random();
15     int index = random.nextInt(size);
16     if (index < leftSize) {
17       return left.getRandomNode();
18     } else if (index == leftSize) {
19       return this;
20     } else {
21       return right.getRandomNode();
22     }
23   }
24
25   public void insertInOrder(int d) {
26     if (d <= data) {
27       if (left == null) {
28         left = new TreeNode(d);
29       } else {
30         left.insertInOrder(d);
31       }
32     } else {
33       if (right == null) {
34         right = new TreeNode(d);
35       } else {
36         right.insertInOrder(d);
37       }
38     }
39     size++;
40   }
41
42   public int size() { return size; }
43   public int data() { return data; }
44
45   public TreeNode find(int d) {
46     if (d == data) {
47       return this;
48     } else if (d <= data) {
49       return left != null ? left.find(d) : null;
50     } else if (d > data) {
51       return right != null ? right.find(d) : null;
52     }
53     return null;
54   }
55 }
```

In a balanced tree, this algorithm will be $O(\log N)$, where N is the number of nodes.

Option #7 [Fast & Working]

Random number calls can be expensive. If we'd like, we can reduce the number of random number calls substantially.

Imagine we called `getRandomNode` *on the tree below, and then traversed left.*



We traversed left because we picked a number between 0 and 5 (inclusive). When we traverse left, we again pick a random number between 0 and 5. Why re-pick? The first number will work just fine.

But what if we went right instead? We have a number between 7 and 8 (inclusive) but we would need a number between 0 and 1 (inclusive). That's easy to fix: just subtract out `LEFT_SIZE + 1`.

Another way to think about what we're doing is that the initial random number call indicates which node (`i`) to return, and then we're locating the `i`th node in an in-order traversal. Subtracting `LEFT_SIZE + 1` from `i` reflects that, when we go right, we skip over `LEFT_SIZE + 1` nodes in the in-order traversal.

```
1  class Tree {
2    TreeNode root = null;
3
4    public int size() { return root == null ? 0 : root.size(); }
5
6    public TreeNode getRandomNode() {
7      if (root == null) return null;
8
9      Random random = new Random();
10     int i = random.nextInt(size());
11     return root.getIthNode(i);
12   }
13
14   public void insertInOrder(int value) {
15     if (root == null) {
16       root = new TreeNode(value);
17     } else {
18       root.insertInOrder(value);
19     }
20   }
21 }
22
23 class TreeNode {
24   /* constructor and variables are the same. */
25
26   public TreeNode getIthNode(int i) {
27     int leftSize = left == null ? 0 : left.size();
28     if (i < leftSize) {
29       return left.getIthNode(i);
30     } else if (i == leftSize) {
31       return this;
32     } else {
```

```
33          /* Skipping over leftSize + 1 nodes, so subtract them. */
34          return right.getIthNode(i - (leftSize + 1));
35        }
36      }
37
38      public void insertInOrder(int d) { /* same */ }
39      public int size() { return size; }
40      public TreeNode find(int d) { /* same */ }
41  }
```

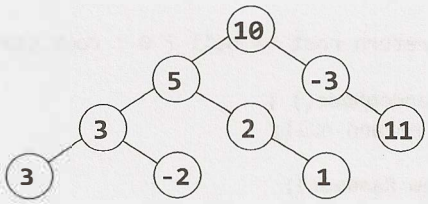
Like the previous algorithm, this algorithm takes O(log N) time in a balanced tree. We can also describe the runtime as O(D), where D is the max depth of the tree. Note that O(D) is an accurate description of the runtime whether the tree is balanced or not.

4.12 Paths with Sum: You are given a binary tree in which each node contains an integer value (which might be positive or negative). Design an algorithm to count the number of paths that sum to a given value. The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes).

pg 111

SOLUTION

Let's pick a potential sum—say, 8—and then draw a binary tree based on this. This tree intentionally has a number of paths with this sum.



One option is the brute force approach.

Solution #1: Brute Force

In the brute force approach, we just look at all possible paths. To do this, we traverse to each node. At each node, we recursively try all paths downwards, tracking the sum as we go. As soon as we hit our target sum, we increment the total.

```
1   int countPathsWithSum(TreeNode root, int targetSum) {
2     if (root == null) return 0;
3
4     /* Count paths with sum starting from the root. */
5     int pathsFromRoot = countPathsWithSumFromNode(root, targetSum, 0);
6
7     /* Try the nodes on the left and right. */
8     int pathsOnLeft = countPathsWithSum(root.left, targetSum);
9     int pathsOnRight = countPathsWithSum(root.right, targetSum);
10
11    return pathsFromRoot + pathsOnLeft + pathsOnRight;
12  }
13
14  /* Returns the number of paths with this sum starting from this node. */
```

```
15 int countPathsWithSumFromNode(TreeNode node, int targetSum, int currentSum) {
16   if (node == null) return 0;
17
18   currentSum += node.data;
19
20   int totalPaths = 0;
21   if (currentSum == targetSum) { // Found a path from the root
22     totalPaths++;
23   }
24
25   totalPaths += countPathsWithSumFromNode(node.left, targetSum, currentSum);
26   totalPaths += countPathsWithSumFromNode(node.right, targetSum, currentSum);
27   return totalPaths;
28 }
```

What is the time complexity of this algorithm?

Consider that node at depth d will be "touched" (via `countPathsWithSumFromNode`) by d nodes above it.

In a balanced binary tree, d will be no more than approximately `log N`. Therefore, we know that with N nodes in the tree, `countPathsWithSumFromNode` will be called `O(N log N)` times. The runtime is O(N log N).

We can also approach this from the other direction. At the root node, we traverse to all `N - 1` nodes beneath it (via `countPathsWithSumFromNode`). At the second level (where there are two nodes), we traverse to `N - 3` nodes. At the third level (where there are four nodes, plus three above those), we traverse to `N - 7` nodes. Following this pattern, the total work is roughly:

```
(N - 1) + (N - 3) + (N - 7) + (N - 15) + (N - 31) + ... + (N - N)
```

To simplify this, notice that the left side of each term is always N and the right side is one less than a power of two. The number of terms is the depth of the tree, which is `O(log N)`. For the right side, we can ignore the fact that it's one less than a power of two. Therefore, we really have this:

```
O(N * [number of terms] - [sum of powers of two from 1 through N])
O(N log N - N)
O(N log N)
```

If the value of the sum of powers of two from 1 through N isn't obvious to you, think about what the powers of two look like in binary:

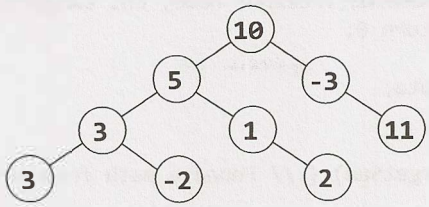
```
  0001
+ 0010
+ 0100
+ 1000
= 1111
```

Therefore, the runtime is `O(N log N)` in a balanced tree.

In an unbalanced tree, the runtime could be much worse. Consider a tree that is just a straight line down. At the root, we traverse to `N - 1` nodes. At the next level (with just a single node), we traverse to `N - 2` nodes. At the third level, we traverse to `N - 3` nodes, and so on. This leads us to the sum of numbers between 1 and N, which is $O(N^2)$.

Solution #2: Optimized

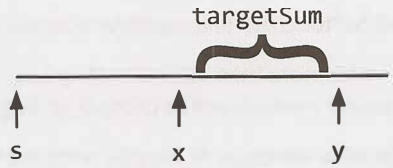
In analyzing the last solution, we may realize that we repeat some work. For a path such as `10 -> 5 -> 3 -> -2`, we traverse this path (or parts of it) repeatedly. We do it when we start with node 10, then when we go to node 5 (looking at 5, then 3, then -2), then when we go to node 3, and then finally when we go to node -2. Ideally, we'd like to reuse this work.



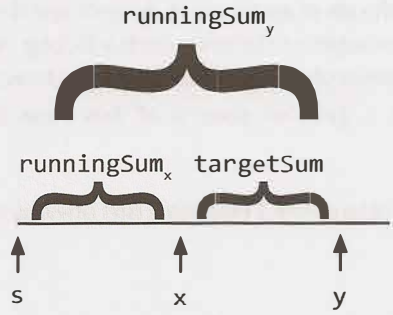
Let's isolate a given path and treat it as just an array. Consider a (hypothetical, extended) path like:

```
10 -> 5 -> 1 -> 2 -> -1 -> -1 -> 7 -> 1 -> 2
```

What we're really saying then is: How many contiguous subsequences in this array sum to a target sum such as 8? In other words, for each `y`, we're trying to find the `x` values below. (Or, more accurately, the number of x values below.)



If each value knows its running sum (the sum of values from `s` through itself), then we can find this pretty easily. We just need to leverage this simple equation: $\texttt{runningSum}_x$ `=` $\texttt{runningSum}_y$ `- targetSum`. We then look for the values of x where this is true.



Since we're just looking for the number of paths, we can use a hash table. As we iterate through the array, build a hash table that maps from a `runningSum` to the number of times we've seen that sum. Then, for each `y`, look up $\texttt{runningSum}_y$ `- targetSum` in the hash table. The value in the hash table will tell you the number of paths with sum `targetSum` that end at `y`.

For example:

```
index:   0    1    2    3    4    5    6    7    8
value:  10 -> 5 -> 1 -> 2 -> -1 -> -1 -> 7 -> 1 -> 2
sum:    10   15   16   18   17   16   23   24   26
```

The value of $\texttt{runningSum}_y$ is 24. If `targetSum` is 8, then we'd look up 16 in the hash table. This would have a value of 2 (originating from index 2 and index 5). As we can see above, indexes 3 through 7 and indexes 6 through 7 have sums of 8.

Now that we've settled the algorithm for an array, let's review this on a tree. We take a similar approach.

We traverse through the tree using depth-first search. As we visit each node:

1. Track its `runningSum`. We'll take this in as a parameter and immediately increment it by `node.value`.
2. Look up `runningSum - targetSum` in the hash table. The value there indicates the total number. Set `totalPaths` to this value.
3. If `runningSum == targetSum`, then there's one additional path that starts at the root. Increment `totalPaths`.
4. Add `runningSum` to the hash table (incrementing the value if it's already there).
5. Recurse left and right, counting the number of paths with sum `targetSum`.
6. After we're done recursing left and right, decrement the value of `runningSum` in the hash table. This is essentially backing out of our work; it reverses the changes to the hash table so that other nodes don't use it (since we're now done with node).

Despite the complexity of deriving this algorithm, the code to implement this is relatively simple.

```
1  int countPathsWithSum(TreeNode root, int targetSum) {
2    return countPathsWithSum(root, targetSum, 0, new HashMap<Integer, Integer>());
3  }
4
5  int countPathsWithSum(TreeNode node, int targetSum, int runningSum,
6                        HashMap<Integer, Integer> pathCount) {
7    if (node == null) return 0; // Base case
8
9    /* Count paths with sum ending at the current node. */
10   runningSum += node.data;
11   int sum = runningSum - targetSum;
12   int totalPaths = pathCount.getOrDefault(sum, 0);
13
14   /* If runningSum equals targetSum, then one additional path starts at root.
15    * Add in this path.*/
16   if (runningSum == targetSum) {
17     totalPaths++;
18   }
19
20   /* Increment pathCount, recurse, then decrement pathCount. */
21   incrementHashTable(pathCount, runningSum, 1); // Increment pathCount
22   totalPaths += countPathsWithSum(node.left, targetSum, runningSum, pathCount);
23   totalPaths += countPathsWithSum(node.right, targetSum, runningSum, pathCount);
24   incrementHashTable(pathCount, runningSum, -1); // Decrement pathCount
25
26   return totalPaths;
27 }
28
29 void incrementHashTable(HashMap<Integer, Integer> hashTable, int key, int delta) {
30   int newCount = hashTable.getOrDefault(key, 0) + delta;
31   if (newCount == 0) { // Remove when zero to reduce space usage
32     hashTable.remove(key);
33   } else {
34     hashTable.put(key, newCount);
35   }
36 }
```

The runtime for this algorithm is $O(N)$, where N is the number of nodes in the tree. We know it is $O(N)$ because we travel to each node just once, doing $O(1)$ work each time. In a balanced tree, the space complexity is $O(\log N)$ due to the hash table. The space complexity can grow to $O(n)$ in an unbalanced tree.

5

Solutions to Bit Manipulation

5.1 **Insertion:** 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 115

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. EXAMPLE: i = 2, j = 4. Result
3     * should be 11100011. For simplicity, we'll use just 8 bits for the example. */
4    int allOnes = ~0; // will equal sequence of all 1s
5
6    // 1s before position j, then 0s. left = 11100000
7    int left = allOnes << (j + 1);
8
9    // 1's after position i. right = 00000011
10   int right = ((1 << i) - 1);
11
12   // All 1s, except for 0s between i and j. mask = 11100011
13   int mask = left | right;
14
15   /* Clear bits j through i then put m in there */
16   int n_cleared = n & mask; // Clear bits j through i.
17   int m_shifted = m << i; // Move m into correct position.
```

```
18
19    return n_cleared | m_shifted; // OR them, and we're done!
20 }
```

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 **Binary to String:** 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 116

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  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.

```
String printBinary2(double num) {
   if (num >= 1 || num <= 0) {
      return "ERROR";
   }

   StringBuilder binary = new StringBuilder();
   double frac = 0.5;
   binary.append(".");
   while (num > 0) {
      /* Setting a limit on length: 32 characters */
      if (binary.length() > 32) {
         return "ERROR";
      }
      if (num >= frac) {
         binary.append(1);
         num -= frac;
      } else {
         binary.append(0);
      }
      frac /= 2;
   }
   return binary.toString();
}
```

Both approaches are equally good; choose the one 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 **Flip Bit to Win:** You have an integer and you can flip exactly one bit from a 0 to a 1. Write code to find the length of the longest sequence of 1s you could create.

EXAMPLE

Input: 1775 (or: 11011101111)

Output: 8

pg 116

SOLUTION

We can think about each integer as being an alternating sequence of 0s and 1s. Whenever a 0s sequence has length one, we can potentially merge the adjacent 1s sequences.

Brute Force

One approach is to convert an integer into an array that reflects the lengths of the 0s and 1s sequences. For example, 11011101111 would be (reading from right to left) $[0_0, 4_1, 1_0, 3_1, 1_0, 2_1, 21_0]$. The subscript reflects whether the integer corresponds to a 0s sequence or a 1s sequence, but the actual solution doesn't need this. It's a strictly alternating sequence, always starting with the 0s sequence.

Once we have this, we just walk through the array. At each 0s sequence, then we consider merging the adjacent 1s sequences if the 0s sequence has length 1.

```
int longestSequence(int n) {
```

```
    if (n == -1) return Integer.BYTES * 8;
    ArrayList<Integer> sequences = getAlternatingSequences(n);
    return findLongestSequence(sequences);
  }

  /* Return a list of the sizes of the sequences. The sequence starts off with the
     number of 0s (which might be 0) and then alternates with the counts of each
     value.*/
  ArrayList<Integer> getAlternatingSequences(int n) {
    ArrayList<Integer> sequences = new ArrayList<Integer>();

    int searchingFor = 0;
    int counter = 0;

    for (int i = 0; i < Integer.BYTES * 8; i++) {
      if ((n & 1) != searchingFor) {
        sequences.add(counter);
        searchingFor = n & 1; // Flip 1 to 0 or 0 to 1
        counter = 0;
      }
      counter++;
      n >>>= 1;
    }
    sequences.add(counter);

    return sequences;
  }

  /* Given the lengths of alternating sequences of 0s and 1s, find the longest one
   * we can build. */
  int findLongestSequence(ArrayList<Integer> seq) {
    int maxSeq = 1;

    for (int i = 0; i < seq.size(); i += 2) {
      int zerosSeq = seq.get(i);
      int onesSeqRight = i - 1 >= 0 ? seq.get(i - 1) : 0;
      int onesSeqLeft = i + 1 < seq.size() ? seq.get(i + 1) : 0;

      int thisSeq = 0;
      if (zerosSeq == 1) { // Can merge
        thisSeq = onesSeqLeft + 1 + onesSeqRight;
      } if (zerosSeq > 1) { // Just add a zero to either side
        thisSeq = 1 + Math.max(onesSeqRight, onesSeqLeft);
      } else if (zerosSeq == 0) { // No zero, but take either side
        thisSeq = Math.max(onesSeqRight, onesSeqLeft);
      }
      maxSeq = Math.max(thisSeq, maxSeq);
    }

    return maxSeq;
  }
```

This is pretty good. It's $O(b)$ time and $O(b)$ memory, where b is the length of the sequence.

> Be careful with how you express the runtime. For example, if you say the runtime is O(n), what is n? It is not correct to say that this algorithm is O(value of the integer). This algorithm is O(number of bits). For this reason, when you have potential ambiguity in what n might mean, it's best just to not use n. Then neither you nor your interviewer will be confused. Pick a different variable name. We used "b", for the number of bits. Something logical works well.

Can we do better? Recall the concept of Best Conceivable Runtime. The B.C.R. for this algorithm is O(b) (since we'll always have to read through the sequence), so we know we can't optimize the time. We can, however, reduce the memory usage.

Optimal Algorithm

To reduce the space usage, note that we don't need to hang on to the length of each sequence the entire time. We only need it long enough to compare each 1s sequence to the immediately preceding 1s sequence.

Therefore, we can just walk through the integer doing this, tracking the current 1s sequence length and the previous 1s sequence length. When we see a zero, update `previousLength`:

- If the next bit is a 1, `previousLength` should be set to `currentLength`.
- If the next bit is a 0, then we can't merge these sequences together. So, set `previousLength` to 0.

Update `maxLength` as we go.

```
1  int flipBit(int a) {
2    /* If all 1s, this is already the longest sequence. */
3    if (~a == 0) return Integer.BYTES * 8;
4
5    int currentLength = 0;
6    int previousLength = 0;
7    int maxLength = 1; // We can always have a sequence of at least one 1
8    while (a != 0) {
9      if ((a & 1) == 1) { // Current bit is a 1
10       currentLength++;
11     } else if ((a & 1) == 0) { // Current bit is a 0
12       /* Update to 0 (if next bit is 0) or currentLength (if next bit is 1). */
13       previousLength = (a & 2) == 0 ? 0 : currentLength;
14       currentLength = 0;
15     }
16     maxLength = Math.max(previousLength + currentLength + 1, maxLength);
17     a >>>= 1;
18   }
19   return maxLength;
20 }
```

The runtime of this algorithm is still O(b), but we use only O(1) additional memory.

5.4 **Next Number:** 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 116

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 terminology can be confusing for this problem. We'll call `getNext` the bigger number and `getPrev` the smaller number.

The Brute Force Approach

An easy approach is simply brute force: count the number of `1`s in `n`, and then increment (or decrement) until you find a number with the same number of `1`s. 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 `0`th and `1`st 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 `0`s are on the left and the `1`s 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   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 with the same
17     * 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 - 2` to 1. This will be `c1 - 1` total bits.

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 the interviewer.

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

pg 116

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).

5.6 **Conversion:** Write a function to determine the number of bits you would need to flip to convert integer A to integer B.

EXAMPLE

Input: `29 (or: 11101), 15 (or: 01111)`

Output: `2`

pg 116

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  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.

5.7 **Pairwise Swap:** 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 116

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  int swapOddEvenBits(int x) {
2    return ( ((x & 0xaaaaaaaa) >>> 1) | ((x & 0x55555555) << 1) );
3  }
```

Note that we use the logical right shift, instead of the arithmetic right shift. This is because we want the sign bit to be filled with a zero.

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.

5.8 **Draw Line:** 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 that draws a horizontal line from (x1, y) to (x2, y).

The method signature should look something like:

```
drawLine(byte[] screen, int width, int x1, int x2, int y)
```

pg 116

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.

6

Solutions to Math and Logic Puzzles

6.1 The Heavy Pill: 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 122

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:

$$\frac{\texttt{weight-210grams}}{\texttt{0.1grams}}.$$

So, if the set of pills weighed 211.3 grams, then Bottle #13 would have the heavy pills.

6.2 **Basketball:** 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 123

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 Game 1 if `0 < p < .5` and Game 2 if `.5 < p < 1`.

If `p = 0, 0.5,` or `1`, then `P(Game 1) = P(Game 2)`, so it doesn't matter which game we play.

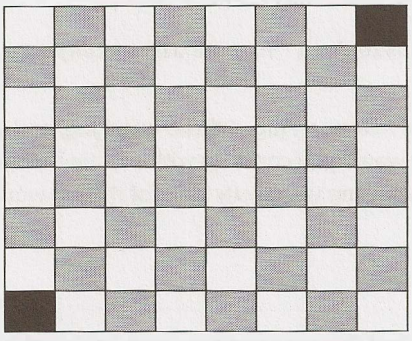
6.3 **Dominos:** There is an 8x8 chessboard in which two diagonally opposite corners have been cut off. You are given 31 dominos, and a single domino can cover exactly two squares. Can you use the 31 dominos to cover the entire board? Prove your answer (by providing an example or showing why it's impossible).

pg 123

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 place, 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 chessboard 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.

6.4 **Ants on a Triangle:** 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 123

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 counterclockwise). 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 collision 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:

$$P(\text{clockwise}) = (1/2)^n$$
$$P(\text{counter}) = (1/2)^n$$
$$P(\text{same direction}) = 2(1/2)^n = (1/2)^{n-1}$$
$$P(\text{collision}) = 1 - P(\text{same direction}) = 1 - (1/2)^{n-1}$$

6.5 **Jugs of Water:** 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 123

SOLUTION

If we just play with the jugs, we'll find that we can pour water back and forth between them as follows:

5 Quart	3 Quart	Action
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 4 quarts.

This question, like many puzzle questions, has a math/computer science root. If the two jug sizes are relatively prime, you can measure any value between one and the sum of the jug sizes.

6.6 **Blue-Eyed Island:** 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 123

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 he who has blue eyes. 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 are 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.

6.7 **The Apocalypse**: In the new post-apocalyptic world, the world queen is desperately concerned about the birth rate. Therefore, she decrees that all families should ensure that they have one girl or else they face massive fines. If all families abide by this policy—that is, they have continue to have children until they have one girl, at which point they immediately stop—what will the gender ratio of the new generation be? (Assume that the odds of someone having a boy or a girl on any given pregnancy is equal.) Solve this out logically and then write a computer simulation of it.

pg 123

SOLUTION

If each family abides by this policy, then each family will have a sequence of zero or more boys followed by a single girl. That is, if "G" indicates a girl and "B" indicates a boy, the sequence of children will look like one of: G; BG; BBG; BBBG; BBBBG; and so on.

We can solve this problem multiple ways.

Mathematically

We can work out the probability for each gender sequence.

- P(G) = $1/2$. That is, 50% of families will have a girl first. The others will go on to have more children.
- P(BG) = $1/4$. Of those who have a second child (which is 50%), 50% of them will have a girl the next time.
- P(BBG) = $1/8$. Of those who have a third child (which is 25%), 50% of them will have a girl the next time.

And so on.

We know that every family has exactly one girl. How many boys does each family have, on average? To compute this, we can look at the expected value of the number of boys. The expected value of the number of boys is the probability of each sequence multiplied by the number of boys in that sequence.

Sequence	Number of Boys	Probability	Number of Boys * Probability
G	0	$1/2$	0
BG	1	$1/4$	$1/4$
BBG	2	$1/8$	$2/8$
BBBG	3	$1/16$	$3/16$
BBBBG	4	$1/32$	$4/32$
BBBBBG	5	$1/64$	$5/64$
BBBBBBG	6	$1/128$	$6/128$

Or in other words, this is the sum of `i` to infinity of i divided by 2^i.

$$\sum_{i=0}^{\infty} \frac{i}{2^i}$$

You probably won't know this off the top of your head, but we can try to estimate it. Let's try converting the above values to a common denominator of 128 (2^6).

$$1/4 = 32/128 \qquad 4/32 = 16/128$$

$$2/8 = 32/128 \qquad 5/64 = 10/128$$

$$3/16 = 24/128 \qquad 6/128 = 6/128$$

$$\frac{32 + 32 + 24 + 16 + 10 + 6}{128} = \frac{120}{128}$$

This looks like it's going to inch closer to $128/128$ (which is of course 1). This "looks like" intuition is valuable, but it's not exactly a mathematical concept. It's a clue though and we can turn to logic here. Should it be 1?

Logically

If the earlier sum is 1, this would mean that the gender ratio is even. Families contribute exactly one girl and on average one boy. The birth policy is therefore ineffective. Does this make sense?

At first glance, this seems wrong. The policy is designed to favor girls as it ensures that all families have a girl.

On the other hand, the families that keep having children contribute (potentially) multiple boys to the population. This could offset the impact of the "one girl" policy.

One way to think about this is to imagine that we put all the gender sequence of each family into one giant string. So if family 1 has BG, family 2 has BBG, and family 3 has G, we would write BGBBGG.

In fact, we don't really care about the groupings of families because we're concerned about the population as a whole. As soon as a child is born, we can just append its gender (B or G) to the string.

What are the odds of the next character being a G? Well, if the odds of having a boy and girl is the same, then the odds of the next character being a G is 50%. Therefore, roughly half of the string should be Gs and half should be Bs, giving an even gender ratio.

This actually makes a lot of sense. Biology hasn't been changed. Half of newborn babies are girls and half are boys. Abiding by some rule about when to stop having children doesn't change this fact.

Therefore, the gender ratio is 50% girls and 50% boys.

Simulation

We'll write this in a simple way that directly corresponds to the problem.

```
1  double runNFamilies(int n) {
2    int boys = 0;
3    int girls = 0;
4    for (int i = 0; i < n; i++) {
5      int[] genders = runOneFamily();
6      girls += genders[0];
7      boys += genders[1];
8    }
9    return girls / (double) (boys + girls);
10 }
11
12 int[] runOneFamily() {
13   Random random = new Random();
14   int boys = 0;
15   int girls = 0;
16   while (girls == 0) { // until we have a girl
17     if (random.nextBoolean()) { // girl
18       girls += 1;
19     } else { // boy
20       boys += 1;
21     }
22   }
23   int[] genders = {girls, boys};
24   return genders;
25 }
```

Sure enough, if you run this on large values of n, you should get something very close to 0.5.

6.8 **The Egg Drop Problem:** 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 124

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. 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.

$$X+(X-1)+(X-2)+...+1=100$$
$$X(X+1)/2=100$$
$$X\approx 13.65$$

X clearly needs to be an integer. Should we round X up or down?

- If we round X up to 14, then we would go up by 14, then 13, then 12, and so on. The last increment would be 4, and it would happen on floor 99. If Egg 1 broke on any of the prior floors, we know we've balanced the eggs such that the number of drops of Egg 1 and Egg 2 always sum to the same thing: 14. If Egg 1 hasn't broken by floor 99, then we just need one more drop to determine if it will break at floor 100. Either way, the number of drops is no more than 14.
- If we round X down to 13, then we would go up by 13, then 12, then 11, and so on. The last increment will be 1 and it will happen at floor 91. This is after 13 drops. Floors 92 through 100 have not been covered yet. We can't cover those floors in just one drop (which would be necessary to merely tie the

"round up" case).

Therefore, we should round X up to 14. That is, 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."

The following code simulates this approach.

```
int breakingPoint = ...;
int countDrops = 0;

boolean drop(int floor) {
  countDrops++;
  return floor >= breakingPoint;
}

int findBreakingPoint(int floors) {
  int interval = 14;
  int previousFloor = 0;
  int egg1 = interval;

  /* Drop egg1 at decreasing intervals. */
  while (!drop(egg1) && egg1 <= floors) {
    interval -= 1;
    previousFloor = egg1;
    egg1 += interval;
  }

  /* Drop egg2 at 1 unit increments. */
  int egg2 = previousFloor + 1;
  while (egg2 < egg1 && egg2 <= floors && !drop(egg2)) {
    egg2 += 1;
  }

  /* If it didn't break, return -1. */
  return egg2 > floors ? -1 : egg2;
}
```

If we want to generalize this code for more building sizes, then we can solve for x in:

$$x(x+1)/2 = \text{number of floors}$$

This will involve the quadratic formula.

6.9 100 Lockers: 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 124

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.

6.10 **Poison**: You have 1000 bottles of soda, and exactly one is poisoned. You have 10 test strips which can be used to detect poison. A single drop of poison will turn the test strip positive permanently. You can put any number of drops on a test strip at once and you can reuse a test strip as many times as you'd like (as long as the results are negative). However, you can only run tests once per day and it takes seven days to return a result. How would you figure out the poisoned bottle in as few days as possible?

Follow up: Write code to simulate your approach.

pg 124

SOLUTION

Observe the wording of the problem. Why seven days? Why not have the results just return immediately?

The fact that there's such a lag between starting a test and reading the results likely means that we'll be doing something else in the meantime (running additional tests). Let's hold on to that thought, but start off with a simple approach just to wrap our heads around the problem.

Naive Approach (28 days)

A simple approach is to divide the bottles across the 10 test strips, first in groups of 100. Then, we wait seven days. When the results come back, we look for a positive result across the test strips. We select the bottles associated with the positive test strip, "toss" (i.e., ignore) all the other bottles, and repeat the process. We perform this operation until there is only one bottle left in the test set.

1. Divide bottles across available test strips, one drop per test strip.
2. After seven days, check the test strips for results.
3. On the positive test strip: select the bottles associated with it into a new set of bottles. If this set size is 1,

we have located the poisoned bottle. If it's greater than one, go to step 1.

To simulate this, we'll build classes for `Bottle` and `TestStrip` that mirror the problem's functionality.

```
class Bottle {
  private boolean poisoned = false;
  private int id;

  public Bottle(int id) { this.id = id; }
  public int getId() { return id; }
  public void setAsPoisoned() { poisoned = true; }
  public boolean isPoisoned() { return poisoned; }
}

class TestStrip {
  public static int DAYS_FOR_RESULT = 7;
  private ArrayList<ArrayList<Bottle>> dropsByDay =
    new ArrayList<ArrayList<Bottle>>();
  private int id;

  public TestStrip(int id) { this.id = id; }
  public int getId() { return id; }

  /* Resize list of days/drops to be large enough. */
  private void sizeDropsForDay(int day) {
    while (dropsByDay.size() <= day) {
      dropsByDay.add(new ArrayList<Bottle>());
    }
  }

  /* Add drop from bottle on specific day. */
  public void addDropOnDay(int day, Bottle bottle) {
    sizeDropsForDay(day);
    ArrayList<Bottle> drops = dropsByDay.get(day);
    drops.add(bottle);
  }

  /* Checks if any of the bottles in the set are poisoned. */
  private boolean hasPoison(ArrayList<Bottle> bottles) {
    for (Bottle b : bottles) {
      if (b.isPoisoned()) {
        return true;
      }
    }
    return false;
  }

  /* Gets bottles used in the test DAYS_FOR_RESULT days ago. */
  public ArrayList<Bottle> getLastWeeksBottles(int day) {
    if (day < DAYS_FOR_RESULT) {
      return null;
    }
    return dropsByDay.get(day - DAYS_FOR_RESULT);
  }

  /* Checks for poisoned bottles since before DAYS_FOR_RESULT */
  public boolean isPositiveOnDay(int day) {
```

```
54        int testDay = day - DAYS_FOR_RESULT;
55        if (testDay < 0 || testDay >= dropsByDay.size()) {
56           return false;
57        }
58        for (int d = 0; d <= testDay; d++) {
59           ArrayList<Bottle> bottles = dropsByDay.get(d);
60           if (hasPoison(bottles)) {
61              return true;
62           }
63        }
64        return false;
65     }
66  }
```

This is just one way of simulating the behavior of the bottles and test strips, and each has its pros and cons.

With this infrastructure built, we can now implement code to test our approach.

```
1   int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
2      int today = 0;
3
4      while (bottles.size() > 1 && strips.size() > 0) {
5         /* Run tests. */
6         runTestSet(bottles, strips, today);
7
8         /* Wait for results. */
9         today += TestStrip.DAYS_FOR_RESULT;
10
11        /* Check results. */
12        for (TestStrip strip : strips) {
13           if (strip.isPositiveOnDay(today)) {
14              bottles = strip.getLastWeeksBottles(today);
15              strips.remove(strip);
16              break;
17           }
18        }
19     }
20
21     if (bottles.size() == 1) {
22        return bottles.get(0).getId();
23     }
24     return -1;
25  }
26
27  /* Distribute bottles across test strips evenly. */
28  void runTestSet(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips, int day) {
29     int index = 0;
30     for (Bottle bottle : bottles) {
31        TestStrip strip = strips.get(index);
32        strip.addDropOnDay(day, bottle);
33        index = (index + 1) % strips.size();
34     }
35  }
36
37  /* The complete code can be found in the downloadable code attachment. */
```

Note that this approach makes the assumption that there will always be multiple test strips at each round. This assumption is valid for 1000 bottles and 10 test strips.

If we can't assume this, we can implement a fail-safe. If we have just one test strip remaining, we start doing one bottle at a time: test a bottle, wait a week, test another bottle. This approach will take at most 28 days.

Optimized Approach (10 days)

As noted in the beginning of the solution, it might be more optimal to run multiple tests at once.

If we divide the bottles up into 10 groups (with bottles 0 - 99 going to strip 0, bottles 100 - 199 going to strip 1, bottles 200 - 299 going to strip 2, and so on), then day 7 will reveal the first digit of the bottle number. A positive result on strip `i` at day 7 shows that the first digit (100's digit) of the bottle number is `i`.

Dividing the bottles in a different way can reveal the second or third digit. We just need to run these tests on different days so that we don't confuse the results.

	Day 0 -> 7	Day 1 -> 8	Day 2 -> 9
Strip 0	0xx	x0x	xx0
Strip 1	1xx	x1x	xx1
Strip 2	2xx	x2x	xx2
Strip 3	3xx	x3x	xx3
Strip 4	4xx	x4x	xx4
Strip 5	5xx	x5x	xx5
Strip 6	6xx	x6x	xx6
Strip 7	7xx	x7x	xx7
Strip 8	8xx	x8x	xx8
Strip 9	9xx	x9x	xx9

For example, if day 7 showed a positive result on strip 4, day 8 showed a positive result on strip 3, and day 9 showed a positive result on strip 8, then this would map to bottle #438.

This mostly works, except for one edge case: what happens if the poisoned bottle has a duplicate digit? For example, bottle #882 or bottle #383.

In fact, these cases are quite different. If day 8 doesn't have any "new" positive results, then we can conclude that digit 2 equals digit 1.

The bigger issue is what happens if day 9 doesn't have any new positive results. In this case, all we know is that digit 3 equals either digit 1 or digit 2. We could not distinguish between bottle #383 and bottle #388. They will both have the same pattern of test results.

We will need to run one additional test. We could run this at the end to clear up ambiguity, but we can also run it at day 3, just in case there's any ambiguity. All we need to do is shift the final digit so that it winds up in a different place than day 2's results.

	Day 0 -> 7	Day 1 -> 8	Day 2 -> 9	Day 3 -> 10
Strip 0	0xx	x0x	xx0	xx9
Strip 1	1xx	x1x	xx1	xx0
Strip 2	2xx	x2x	xx2	xx1
Strip 3	3xx	x3x	xx3	xx2
Strip 4	4xx	x4x	xx4	xx3
Strip 5	5xx	x5x	xx5	xx4

	Day 0 -> 7	Day 1 -> 8	Day 2 -> 9	Day 3 -> 10
Strip 6	6xx	x6x	xx6	xx5
Strip 7	7xx	x7x	xx7	xx6
Strip 8	8xx	x8x	xx8	xx7
Strip 9	9xx	x9x	xx9	xx8

Now, bottle #383 will see (Day 7 = #3, Day 8 -> #8, Day 9 -> [NONE], Day 10 -> #4), while bottle #388 will see (Day 7 = #3, Day 8 -> #8, Day 9 -> [NONE], Day 10 -> #9). We can distinguish between these by "reversing" the shifting on day 10's results.

What happens, though, if day 10 still doesn't see any new results? Could this happen?

Actually, yes. Bottle #898 would see (Day 7 = #8, Day 8 -> #9, Day 9 -> [NONE], Day 10 -> [NONE]). That's okay, though. We just need to distinguish bottle #898 from #899. Bottle #899 will see (Day 7 = #8, Day 8 -> #9, Day 9 -> [NONE], Day 10 -> #0).

The "ambiguous" bottles from day 9 will always map to different values on day 10. The logic is:

- If Day 3->10's test reveals a new test result, "unshift" this value to derive the third digit.
- Otherwise, we know that the third digit equals either the first digit or the second digit *and* that the third digit, when shifted, still equals either the first digit or the second digit. Therefore, we just need to figure out whether the first digit "shifts" into the second digit or the other way around. In the former case, the third digit equals the first digit. In the latter case, the third digit equals the second digit.

Implementing this requires some careful work to prevent bugs.

```
1  int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
2     if (bottles.size() > 1000 || strips.size() < 10) return -1;
3
4     int tests = 4; // three digits, plus one extra
5     int nTestStrips = strips.size();
6
7     /* Run tests. */
8     for (int day = 0; day < tests; day++) {
9        runTestSet(bottles, strips, day);
10    }
11
12    /* Get results. */
13    HashSet<Integer> previousResults = new HashSet<Integer>();
14    int[] digits = new int[tests];
15    for (int day = 0; day < tests; day++) {
16       int resultDay = day + TestStrip.DAYS_FOR_RESULT;
17       digits[day] = getPositiveOnDay(strips, resultDay, previousResults);
18       previousResults.add(digits[day]);
19    }
20
21    /* If day 1's results matched day 0's, update the digit. */
22    if (digits[1] == -1) {
23       digits[1] = digits[0];
24    }
25
26    /* If day 2 matched day 0 or day 1, check day 3. Day 3 is the same as day 2, but
27     * incremented by 1. */
28    if (digits[2] == -1) {
```

```
    if (digits[3] == -1) { /* Day 3 didn't give new result */
      /* Digit 2 equals digit 0 or digit 1. But, digit 2, when incremented also
       * matches digit 0 or digit 1. This means that digit 0 incremented matches
       * digit 1, or the other way around. */
      digits[2] = ((digits[0] + 1) % nTestStrips) == digits[1] ?
                  digits[0] : digits[1];
    } else {
        digits[2] = (digits[3] - 1 + nTestStrips) % nTestStrips;
    }
  }

  return digits[0] * 100 + digits[1] * 10 + digits[2];
}

/* Run set of tests for this day. */
void runTestSet(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips, int day) {
  if (day > 3) return; // only works for 3 days (digits) + one extra

  for (Bottle bottle : bottles) {
    int index = getTestStripIndexForDay(bottle, day, strips.size());
    TestStrip testStrip = strips.get(index);
    testStrip.addDropOnDay(day, bottle);
  }
}

/* Get strip that should be used on this bottle on this day. */
int getTestStripIndexForDay(Bottle bottle, int day, int nTestStrips) {
  int id = bottle.getId();
  switch (day) {
    case 0: return id /100;
    case 1: return (id % 100) / 10;
    case 2: return id % 10;
    case 3: return (id % 10 + 1) % nTestStrips;
    default: return -1;
  }
}

/* Get results that are positive for a particular day, excluding prior results. */
int getPositiveOnDay(ArrayList<TestStrip> testStrips, int day,
                     HashSet<Integer> previousResults) {
  for (TestStrip testStrip : testStrips) {
    int id = testStrip.getId();
    if (testStrip.isPositiveOnDay(day) && !previousResults.contains(id)) {
      return testStrip.getId();
    }
  }
  return -1;
}
```

It will take 10 days in the worst case to get a result with this approach.

Optimal Approach (7 days)

We can actually optimize this slightly more, to return a result in just seven days. This is of course the minimum number of days possible.

Notice what each test strip really means. It's a binary indicator for poisoned or unpoisoned. Is it possible to map 1000 keys to 10 binary values such that each key is mapped to a unique configuration of values? Yes, of course. This is what a binary number is.

We can take each bottle number and look at its binary representation. If there's a 1 in the `i`th digit, then we will add a drop of this bottle's contents to test strip `i`. Observe that 2^{10} is 1024, so 10 test strips will be enough to handle up to 1024 bottles.

We wait seven days, and then read the results. If test strip `i` is positive, then set bit `i` of the result value. Reading all the test strips will give us the ID of the poisoned bottle.

```
1  int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
2    runTests(bottles, strips);
3    ArrayList<Integer> positive = getPositiveOnDay(strips, 7);
4    return setBits(positive);
5  }
6
7  /* Add bottle contents to test strips */
8  void runTests(ArrayList<Bottle> bottles, ArrayList<TestStrip> testStrips) {
9    for (Bottle bottle : bottles) {
10     int id = bottle.getId();
11     int bitIndex = 0;
12     while (id > 0) {
13       if ((id & 1) == 1) {
14         testStrips.get(bitIndex).addDropOnDay(0, bottle);
15       }
16       bitIndex++;
17       id >>= 1;
18     }
19   }
20 }
21
22 /* Get test strips that are positive on a particular day. */
23 ArrayList<Integer> getPositiveOnDay(ArrayList<TestStrip> testStrips, int day) {
24   ArrayList<Integer> positive = new ArrayList<Integer>();
25   for (TestStrip testStrip : testStrips) {
26     int id = testStrip.getId();
27     if (testStrip.isPositiveOnDay(day)) {
28       positive.add(id);
29     }
30   }
31   return positive;
32 }
33
34 /* Create number by setting bits with indices specified in positive. */
35 int setBits(ArrayList<Integer> positive) {
36   int id = 0;
37   for (Integer bitIndex : positive) {
38     id |= 1 << bitIndex;
39   }
40   return id;
41 }
```

This approach will work as long as $2^T >= B$, where T is the number of test strips and B is the number of bottles.

7

Solutions to Object-Oriented Design

7.1 Deck of Cards: Design the data structures for a generic deck of cards. Explain how you would subclass the data structures to implement blackjack.

pg 127

SOLUTION

First, we need to recognize that a "generic" deck of cards can mean many things. Generic could mean a standard deck of cards that can play a poker-like game, or it could even stretch to Uno or Baseball cards. It is important to ask your interviewer what she means by generic.

Let's assume that your interviewer clarifies that the deck is a standard 52-card set, like you might see used in a blackjack or poker game. If so, the design might look like this:

```
1  public enum Suit {
2    Club (0), Diamond (1), Heart (2), Spade (3);
3    private int value;
4    private Suit(int v) { value = v; }
5    public int getValue() { return value; }
6    public static Suit getSuitFromValue(int value) { ... }
7  }
8
9  public class Deck <T extends Card> {
10   private ArrayList<T> cards; // all cards, dealt or not
11   private int dealtIndex = 0; // marks first undealt card
12
13   public void setDeckOfCards(ArrayList<T> deckOfCards) { ... }
14
15   public void shuffle() { ... }
16   public int remainingCards() {
17      return cards.size() - dealtIndex;
18   }
19   public T[] dealHand(int number) { ... }
20   public T dealCard() { ... }
21 }
22
23 public abstract class Card {
24   private boolean available = true;
25
26   /* number or face that's on card - a number 2 through 10, or 11 for Jack, 12 for
27    * Queen, 13 for King, or 1 for Ace */
28   protected int faceValue;
29   protected Suit suit;
```

```
30
31    public Card(int c, Suit s) {
32       faceValue = c;
33       suit = s;
34    }
35
36    public abstract int value();
37    public Suit suit() { return suit; }
38
39    /* Checks if the card is available to be given out to someone */
40    public boolean isAvailable() { return available; }
41    public void markUnavailable() { available = false; }
42    public void markAvailable() { available = true; }
43 }
44
45 public class Hand <T extends Card> {
46    protected ArrayList<T> cards = new ArrayList<T>();
47
48    public int score() {
49       int score = 0;
50       for (T card : cards) {
51          score += card.value();
52       }
53       return score;
54    }
55
56    public void addCard(T card) {
57       cards.add(card);
58    }
59 }
```

In the above code, we have implemented `Deck` with generics but restricted the type of `T` to `Card`. We have also implemented `Card` as an abstract class, since methods like `value()` don't make much sense without a specific game attached to them. (You could make a compelling argument that they should be implemented anyway, by defaulting to standard poker rules.)

Now, let's say we're building a blackjack game, so we need to know the value of the cards. Face cards are 10 and an ace is 11 (most of the time, but that's the job of the Hand class, not the following class).

```
1  public class BlackJackHand extends Hand<BlackJackCard> {
2     /* There are multiple possible scores for a blackjack hand, since aces have
3      * multiple values. Return the highest possible score that's under 21, or the
4      * lowest score that's over. */
5     public int score() {
6        ArrayList<Integer> scores = possibleScores();
7        int maxUnder = Integer.MIN_VALUE;
8        int minOver = Integer.MAX_VALUE;
9        for (int score : scores) {
10          if (score > 21 && score < minOver) {
11             minOver = score;
12          } else if (score <= 21 && score > maxUnder) {
13             maxUnder = score;
14          }
15       }
16       return maxUnder == Integer.MIN_VALUE ? minOver : maxUnder;
17    }
18
```

```
19    /* return a list of all possible scores this hand could have (evaluating each
20     * ace as both 1 and 11 */
21    private ArrayList<Integer> possibleScores() { ... }
22
23    public boolean busted() { return score() > 21; }
24    public boolean is21() { return score() == 21; }
25    public boolean isBlackJack() { ... }
26 }
27
28 public class BlackJackCard extends Card {
29    public BlackJackCard(int c, Suit s) { super(c, s); }
30    public int value() {
31       if (isAce()) return 1;
32       else if (faceValue >= 11 && faceValue <= 13) return 10;
33       else return faceValue;
34    }
35
36    public int minValue() {
37       if (isAce()) return 1;
38       else return value();
39    }
40
41    public int maxValue() {
42       if (isAce()) return 11;
43       else return value();
44    }
45
46    public boolean isAce() {
47       return faceValue == 1;
48    }
49
50    public boolean isFaceCard() {
51       return faceValue >= 11 && faceValue <= 13;
52    }
53 }
```

This is just one way of handling aces. We could, alternatively, create a class of type `Ace` that extends `BlackJackCard`.

An executable, fully automated version of blackjack is provided in the downloadable code attachment.

7.2 **Call Center:** 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 127

SOLUTION

All three ranks of employees have different work to be done, so those specific functions are profile specific. We should keep these things within their respective class.

There are a few things which are common to them, like address, name, job title, and age. These things can be kept in one class and can be extended or inherited by others.

Finally, there should be one `CallHandler` class which would route the calls to the correct person.

Note that on any object-oriented design question, there are many ways to design the objects. Discuss the trade-offs of different solutions with your interviewer. You should usually design for long-term code flexibility and maintenance.

We'll go through each of the classes below in detail.

`CallHandler` represents the body of the program, and all calls are funneled first through it.

```
1  public class CallHandler {
2    /* 3 levels of employees: respondents, managers, directors. */
3    private final int LEVELS = 3;
4
5    /* Initialize 10 respondents, 4 managers, and 2 directors. */
6    private final int NUM_RESPONDENTS = 10;
7    private final int NUM_MANAGERS = 4;
8    private final int NUM_DIRECTORS = 2;
9
10   /* List of employees, by level.
11    * employeeLevels[0] = respondents
12    * employeeLevels[1] = managers
13    * employeeLevels[2] = directors
14    */
15   List<List<Employee>> employeeLevels;
16
17   /* queues for each call's rank */
18   List<List<Call>> callQueues;
19
20   public CallHandler() { ... }
21
22   /* Gets the first available employee who can handle this call.*/
23   public Employee getHandlerForCall(Call call) { ... }
24
25   /* Routes the call to an available employee, or saves in a queue if no employee
26    * is available. */
27   public void dispatchCall(Caller caller) {
28     Call call = new Call(caller);
29     dispatchCall(call);
30   }
31
32   /* Routes the call to an available employee, or saves in a queue if no employee
33    * is available. */
34   public void dispatchCall(Call call) {
35     /* Try to route the call to an employee with minimal rank. */
36     Employee emp = getHandlerForCall(call);
37     if (emp != null) {
38       emp.receiveCall(call);
39       call.setHandler(emp);
40     } else {
41       /* Place the call into corresponding call queue according to its rank. */
42       call.reply("Please wait for free employee to reply");
43       callQueues.get(call.getRank().getValue()).add(call);
44     }
45   }
```

```
46
47    / *An employee got free. Look for a waiting call that employee can serve. Return
48     * true if we assigned a call, false otherwise. */
49    public boolean assignCall(Employee emp) { ... }
50 }
```

`Call` represents a call from a user. A call has a minimum rank and is assigned to the first employee who can handle it.

```
1  public class Call {
2     / *Minimal rank of employee who can handle this call. */
3     private Rank rank;
4
5     / *Person who is calling. */
6     private Caller caller;
7
8     / *Employee who is handling call. */
9     private Employee handler;
10
11    public Call(Caller c) {
12       rank = Rank.Responder;
13       caller = c;
14    }
15
16    / *Set employee who is handling call. */
17    public void setHandler(Employee e) { handler = e; }
18
19    public void reply(String message) { ... }
20    public Rank getRank() { return rank; }
21    public void setRank(Rank r) { rank = r; }
22    public Rank incrementRank() { ... }
23    public void disconnect() { ... }
24 }
```

`Employee` is a super class for the `Director`, `Manager`, and `Respondent` classes. It is implemented as an abstract class since there should be no reason to instantiate an `Employee` type directly.

```
1  abstract class Employee {
2     private Call currentCall = null;
3     protected Rank rank;
4
5     public Employee(CallHandler handler) { ... }
6
7     / *Start the conversation */
8     public void receiveCall(Call call) { ... }
9
10    / *the issue is resolved, finish the call */
11    public void callCompleted() { ... }
12
13    / *The issue has not been resolved. Escalate the call, and assign a new call to
14     * the employee. */
15    public void escalateAndReassign() { ... }
16
17    / *Assign a new call to an employee, if the employee is free. */
18    public boolean assignNewCall() { ... }
19
20    / *Returns whether or not the employee is free. */
21    public boolean isFree() { return currentCall == null; }
22
```

```
23    public Rank getRank() { return rank; }
24  }
25
```

The `Respondent`, `Director`, and `Manager` classes are now just simple extensions of the `Employee` class.

```
1   class Director extends Employee {
2     public Director() {
3       rank = Rank.Director;
4     }
5   }
6
7   class Manager extends Employee {
8     public Manager() {
9       rank = Rank.Manager;
10    }
11  }
12
13  class Respondent extends Employee {
14    public Respondent() {
15      rank = Rank.Responder;
16    }
17  }
```

This is just one way of designing this problem. Note that there are many other ways that are equally good.

This may seem like an awful lot of code to write in an interview, and it is. We've been much more thorough here than you would need. In a real interview, you would likely be much lighter on some of the details until you have time to fill them in.

7.3 Jukebox: Design a musical jukebox using object-oriented principles.

pg 127

SOLUTION

In any object-oriented design question, you first want to start off with asking your interviewer some questions to clarify design constraints. Is this jukebox playing CDs? Records? MP3s? Is it a simulation on a computer, or is it supposed to represent a physical jukebox? Does it take money, or is it free? And if it takes money, which currency? And does it deliver change?

Unfortunately, we don't have an interviewer here that we can have this dialogue with. Instead, we'll make some assumptions. We'll assume that the jukebox is a computer simulation that closely mirrors physical jukeboxes, and we'll assume that it's free.

Now that we have that out of the way, we'll outline the basic system components:

- Jukebox
- CD
- Song
- Artist
- Playlist
- Display (displays details on the screen)

Now, let's break this down further and think about the possible actions.

- Playlist creation (includes add, delete, and shuffle)
- CD selector
- Song selector
- Queuing up a song
- Get next song from playlist

A user also can be introduced:

- Adding
- Deleting
- Credit information

Each of the main system components translates roughly to an object, and each action translates to a method. Let's walk through one potential design.

The `Jukebox` class represents the body of the problem. Many of the interactions between the components of the system, or between the system and the user, are channeled through here.

```
1  public class Jukebox {
2    private CDPlayer cdPlayer;
3    private User user;
4    private Set<CD> cdCollection;
5    private SongSelector ts;
6
7    public Jukebox(CDPlayer cdPlayer, User user, Set<CD> cdCollection,
8                   SongSelector ts) { ... }
9
10   public Song getCurrentSong() { return ts.getCurrentSong(); }
11   public void setUser(User u) { this.user = u; }
12 }
```

Like a real CD player, the `CDPlayer` class supports storing just one CD at a time. The CDs that are not in play are stored in the jukebox.

```
1  public class CDPlayer {
2    private Playlist p;
3    private CD c;
4
5    / * Constructors. */
6    public CDPlayer(CD c, Playlist p) { ... }
7    public CDPlayer(Playlist p) { this.p = p; }
8    public CDPlayer(CD c) { this.c = c; }
9
10   / *Play song */
11   public void playSong(Song s) { ... }
12
13   / *Getters and setters */
14   public Playlist getPlaylist() { return p; }
15   public void setPlaylist(Playlist p) { this.p = p; }
16
17   public CD getCD() { return c; }
18   public void setCD(CD c) { this.c = c; }
19 }
```

The `Playlist` manages the current and next songs to play. It is essentially a wrapper class for a queue and offers some additional methods for convenience.

```
public class Playlist {
   private Song song;
   private Queue<Song> queue;
   public Playlist(Song song, Queue<Song> queue) {
      ...
   }
   public Song getNextSToPlay() {
      return queue.peek();
   }
   public void queueUpSong(Song s) {
      queue.add(s);
   }
}
```

The classes for CD, Song, and User are all fairly straightforward. They consist mainly of member variables and getters and setters.

```
public class CD { /* data for id, artist, songs, etc */ }

public class Song { /* data for id, CD (could be null), title, length, etc */ }

public class User {
   private String name;
   public String getName() { return name; }
   public void setName(String name) {   this.name = name; }
   public long getID() { return ID; }
   public void setID(long iD) { ID = iD; }
   private long ID;
   public User(String name, long iD) { ... }
   public User getUser() { return this; }
   public static User addUser(String name, long iD) { ... }
}
```

This is by no means the only "correct" implementation. The interviewer's responses to initial questions, as well as other constraints, will shape the design of the jukebox classes.

7.4 Parking Lot: Design a parking lot using object-oriented principles.

pg 127

SOLUTION

The wording of this question is vague, just as it would be in an actual interview. This requires you to have a conversation with your interviewer about what types of vehicles it can support, whether the parking lot has multiple levels, and so on.

For our purposes right now, we'll make the following assumptions. We made these specific assumptions to add a bit of complexity to the problem without adding too much. If you made different assumptions, that's totally fine.

- The parking lot has multiple levels. Each level has multiple rows of spots.
- The parking lot can park motorcycles, cars, and buses.
- The parking lot has motorcycle spots, compact spots, and large spots.
- A motorcycle can park in any spot.
- A car can park in either a single compact spot or a single large spot.

- A bus can park in five large spots that are consecutive and within the same row. It cannot park in small spots.

In the below implementation, we have created an abstract class `Vehicle`, from which `Car`, `Bus`, and `Motorcycle` inherit. To handle the different parking spot sizes, we have just one class `ParkingSpot` which has a member variable indicating the size.

```
1  public enum VehicleSize { Motorcycle, Compact,   Large }
2
3  public abstract class Vehicle {
4    protected ArrayList<ParkingSpot> parkingSpots = new ArrayList<ParkingSpot>();
5    protected String licensePlate;
6    protected int spotsNeeded;
7    protected VehicleSize size;
8
9    public int getSpotsNeeded() { return spotsNeeded; }
10   public VehicleSize getSize() { return size; }
11
12   /* Park vehicle in this spot (among others, potentially) */
13   public void parkInSpot(ParkingSpot s) { parkingSpots.add(s); }
14
15   /* Remove car from spot, and notify spot that it's gone */
16   public void clearSpots() { ... }
17
18   /* Checks if the spot is big enough for the vehicle (and is available). This
19    * compares the SIZE only. It does not check if it has enough spots. */
20   public abstract boolean canFitInSpot(ParkingSpot spot);
21 }
22
23 public class Bus extends Vehicle {
24   public Bus() {
25     spotsNeeded = 5;
26     size = VehicleSize.Large;
27   }
28
29   /* Checks if the spot is a Large. Doesn't check num of spots */
30   public boolean canFitInSpot(ParkingSpot spot) { ... }
31 }
32
33 public class Car extends Vehicle {
34   public Car() {
35     spotsNeeded = 1;
36     size = VehicleSize.Compact;
37   }
38
39   /* Checks if the spot is a Compact or a Large. */
40   public boolean canFitInSpot(ParkingSpot spot) { ... }
41 }
42
43 public class Motorcycle extends Vehicle {
44   public Motorcycle() {
45     spotsNeeded = 1;
46     size = VehicleSize.Motorcycle;
47   }
48
49   public boolean canFitInSpot(ParkingSpot spot) { ... }
50 }
```

The `ParkingLot` class is essentially a wrapper class for an array of `Levels`. By implementing it this way, we are able to separate out logic that deals with actually finding free spots and parking cars out from the broader actions of the `ParkingLot`. If we didn't do it this way, we would need to hold parking spots in some sort of double array (or hash table which maps from a level number to the list of spots). It's cleaner to just separate `ParkingLot` from `Level`.

```
1  public class ParkingLot {
2    private Level[] levels;
3    private final int NUM_LEVELS = 5;
4
5    public ParkingLot() { ... }
6
7    /* Park the vehicle in a spot (or multiple spots). Return false if failed. */
8    public boolean parkVehicle(Vehicle vehicle) { ... }
9  }
10
11 /* Represents a level in a parking garage */
12 public class Level {
13   private int floor;
14   private ParkingSpot[] spots;
15   private int availableSpots = 0; // number of free spots
16   private static final int SPOTS_PER_ROW = 10;
17
18   public Level(int flr, int numberSpots) { ... }
19
20   public int availableSpots() { return availableSpots; }
21
22   /* Find a place to park this vehicle. Return false if failed. */
23   public boolean parkVehicle(Vehicle vehicle) { ... }
24
25   /* Park a vehicle starting at the spot spotNumber, and continuing until
26    * vehicle.spotsNeeded. */
27   private boolean parkStartingAtSpot(int num, Vehicle v) { ... }
28
29   /* Find a spot to park this vehicle. Return index of spot, or -1 on failure. */
30   private int findAvailableSpots(Vehicle vehicle) { ... }
31
32   /* When a car was removed from the spot, increment availableSpots */
33   public void spotFreed() { availableSpots++; }
34 }
```

The `ParkingSpot` is implemented by having just a variable which represents the size of the spot. We could have implemented this by having classes for `LargeSpot`, `CompactSpot`, and `MotorcycleSpot` which inherit from `ParkingSpot`, but this is probably overkill. The spots probably do not have different behaviors, other than their sizes.

```
1  public class ParkingSpot {
2    private Vehicle vehicle;
3    private VehicleSize spotSize;
4    private int row;
5    private int spotNumber;
6    private Level level;
7
8    public ParkingSpot(Level lvl, int r, int n, VehicleSize s) {...}
9
10   public boolean isAvailable() { return vehicle == null; }
11
```

```
12    /* Check if the spot is big enough and is available */
13    public boolean canFitVehicle(Vehicle vehicle) { ... }
14
15    /* Park vehicle in this spot. */
16    public boolean park(Vehicle v) { ... }
17
18    public int getRow() { return row; }
19    public int getSpotNumber() { return spotNumber; }
20
21    /* Remove vehicle from spot, and notify level that a new spot is available */
22    public void removeVehicle() { ... }
23 }
```

A full implementation of this code, including executable test code, is provided in the downloadable code attachment.

7.5 Online Book Reader: Design the data structures for an online book reader system.

pg 127

SOLUTION

Since the problem doesn't describe much about the functionality, let's assume we want to design a basic online reading system which provides the following functionality:

- User membership creation and extension.
- Searching the database of books.
- Reading a book.
- Only one active user at a time
- Only one active book by this user.

To implement these operations we may require many other functions, like `get`, `set`, `update`, and so on. The objects required would likely include `User`, Book, and `Library`.

The class `OnlineReaderSystem` represents the body of our program. We could implement the class such that it stores information about all the books, deals with user management, and refreshes the display, but that would make this class rather hefty. Instead, we've chosen to tear off these components into `Library`, `UserManager`, and `Display` classes.

```
1  public class OnlineReaderSystem {
2     private Library library;
3     private UserManager userManager;
4     private Display display;
5
6     private Book activeBook;
7     private User activeUser;
8
9     public OnlineReaderSystem() {
10       userManager = new UserManager();
11       library = new Library();
12       display = new Display();
13    }
14
15    public Library getLibrary() { return library;  }
16    public UserManager getUserManager() { return userManager; }
```

```
17    public Display getDisplay() { return display; }
18
19    public Book getActiveBook() { return activeBook; }
20    public void setActiveBook(Book book) {
21       activeBook = book;
22       display.displayBook(book);
23    }
24
25    public User getActiveUser() { return activeUser; }
26    public void setActiveUser(User user) {
27       activeUser = user;
28       display.displayUser(user);
29    }
30 }
```

We then implement separate classes to handle the user manager, the library, and the display components.

```
1  public class Library {
2     private HashMap<Integer, Book> books;
3
4     public Book addBook(int id, String details) {
5        if (books.containsKey(id)) {
6           return null;
7        }
8        Book book = new Book(id, details);
9        books.put(id, book);
10       return book;
11    }
12
13    public boolean remove(Book b) { return remove(b.getID()); }
14    public boolean remove(int id) {
15       if (!books.containsKey(id)) {
16          return false;
17       }
18       books.remove(id);
19       return true;
20    }
21
22    public Book find(int id) {
23       return books.get(id);
24    }
25 }
26
27 public class UserManager {
28    private HashMap<Integer, User> users;
29
30    public User addUser(int id, String details, int accountType) {
31       if (users.containsKey(id)) {
32          return null;
33       }
34       User user = new User(id, details, accountType);
35       users.put(id, user);
36       return user;
37    }
38
39    public User find(int id) { return users.get(id); }
40    public boolean remove(User u) { return remove(u.getID()); }
41    public boolean remove(int id) {
```

```
42        if (!users.containsKey(id)) {
43           return false;
44        }
45        users.remove(id);
46        return true;
47     }
48  }
49
50  public class Display {
51     private Book activeBook;
52     private User activeUser;
53     private int pageNumber = 0;
54
55     public void displayUser(User user) {
56        activeUser = user;
57        refreshUsername();
58     }
59
60     public void displayBook(Book book) {
61        pageNumber = 0;
62        activeBook = book;
63
64        refreshTitle();
65        refreshDetails();
66        refreshPage();
67     }
68
69     public void turnPageForward() {
70        pageNumber++;
71        refreshPage();
72     }
73
74     public void turnPageBackward() {
75        pageNumber--;
76        refreshPage();
77     }
78
79     public void refreshUsername() { /* updates username display */ }
80     public void refreshTitle() { /* updates title display */ }
81     public void refreshDetails() { /* updates details display */   }
82     public void refreshPage() { /* updated page display */ }
83  }
```

The classes for `User` and `Book` simply hold data and provide little true functionality.

```
1   public class Book {
2      private int bookId;
3      private String details;
4
5      public Book(int id, String det) {
6         bookId = id;
7         details = det;
8      }
9
10     public int getID() { return bookId;  }
11     public void setID(int id) { bookId = id; }
12     public String getDetails() { return details; }
13     public void setDetails(String d) { details = d; }
```

```
}

public class User {
   private int userId;
   private String details;
   private int accountType;

   public void renewMembership() {  }

   public User(int id, String details, int accountType) {
      userId = id;
      this.details = details;
      this.accountType = accountType;
   }

   /* Getters and setters */
   public int getID() { return userId; }
   public void setID(int id) { userId = id; }
   public String getDetails() {
      return details;
   }

   public void setDetails(String details) {
      this.details = details;
   }
   public int getAccountType() { return accountType; }
   public void setAccountType(int t) { accountType = t; }
}
```

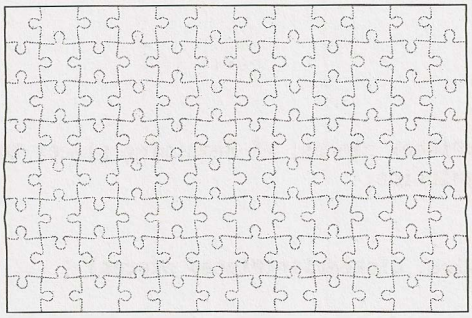
The decision to tear off user management, library, and display into their own classes, when this functionality could have been in the general `OnlineReaderSystem` class, is an interesting one. On a very small system, making this decision could make the system overly complex. However, as the system grows, and more and more functionality gets added to `OnlineReaderSystem`, breaking off such components prevents this main class from getting overwhelmingly lengthy.

7.6 **Jigsaw:** Implement an NxN 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 edges, returns true if the two edges belong together.

pg 128

SOLUTION

We have a traditional jigsaw puzzle. The puzzle is grid-like, with rows and columns. Each piece is located in a single row and column and has four edges. Each edge comes in one of three types: inner, outer, and flat. A corner piece, for example, will have two flat edges and two other edges, which could be inner or outer.



As we solve the jigsaw puzzle (manually or algorithmically), we'll need to store the position of each piece. We could think about the position as absolute or relative:

- *Absolute Position:* "This piece is located at position (12, 23)."
- *Relative Position:* "I don't know where this piece is actually located, but I know it is next to this other piece."

For our solution, we will use the absolute position.

We'll need classes to represent `Puzzle`, `Piece`, and `Edge`. Additionally, we'll want enums for the different shapes (`inner`, `outer`, `flat`) and the orientations of the edges (`left`, `top`, `right`, `bottom`).

`Puzzle` will start off with a list of the pieces. When we solve the puzzle, we'll fill in an NxN `solution` matrix of pieces.

`Piece` will have a hash table that maps from an orientation to the appropriate edge. Note that we might rotate the piece at some point, so the hash table could change. The orientation of the edges will be arbitrarily assigned at first.

`Edge` will have just its shape and a pointer back to its parent piece. It will not keep its orientation.

A potential object-oriented design looks like the following:

```
1   public enum Orientation {
2     LEFT, TOP, RIGHT, BOTTOM; // Should stay in this order
3
4     public Orientation getOpposite() {
5       switch (this) {
6         case LEFT: return RIGHT;
7         case RIGHT: return LEFT;
8         case TOP: return BOTTOM;
9         case BOTTOM: return TOP;
10        default: return null;
11      }
12    }
13  }
14
15  public enum Shape {
16    INNER, OUTER, FLAT;
17
18    public Shape getOpposite() {
19      switch (this) {
20        case INNER: return OUTER;
21        case OUTER: return INNER;
22        default: return null;
```

```
    }
  }
}

public class Puzzle {
  private LinkedList<Piece> pieces; /* Remaining pieces to put away. */
  private Piece[][] solution;
  private int size;

  public Puzzle(int size, LinkedList<Piece> pieces) { ... }


  /* Put piece into the solution, turn it appropriately, and remove from list. */
  private void setEdgeInSolution(LinkedList<Piece> pieces, Edge edge, int row,
                                 int column, Orientation orientation) {
    Piece piece = edge.getParentPiece();
    piece.setEdgeAsOrientation(edge, orientation);
    pieces.remove(piece);
    solution[row][column] = piece;
  }

  /* Find the matching piece in piecesToSearch and insert it at row, column. */
  private boolean fitNextEdge(LinkedList<Piece> piecesToSearch, int row, int col);

  /* Solve puzzle. */
  public boolean solve() { ... }
}

public class Piece {
  private HashMap<Orientation, Edge> edges = new HashMap<Orientation, Edge>();

  public Piece(Edge[] edgeList) { ... }

  /* Rotate edges by "numberRotations". */
  public void rotateEdgesBy(int numberRotations) { ... }

  public boolean isCorner() { ... }
  public boolean isBorder() { ... }
}

public class Edge {
  private Shape shape;
  private Piece parentPiece;
  public Edge(Shape shape) { ... }
  public boolean fitsWith(Edge edge) { ... }
}
```

Algorithm to Solve the Puzzle

Just as a kid might in solving a puzzle, we'll start with grouping the pieces into corner pieces, border pieces, and inside pieces.

Once we've done that, we'll pick an arbitrary corner piece and put it in the top left corner. We will then walk through the puzzle in order, filling in piece by piece. At each location, we search through the correct group of pieces to find the matching piece. When we insert the piece into the puzzle, we need to rotate the piece to fit correctly.

The code below outlines this algorithm.

```
1  /* Find the matching piece within piecesToSearch and insert it at row, column. */
2  boolean fitNextEdge(LinkedList<Piece> piecesToSearch, int row, int column) {
3     if (row == 0 && column == 0) { // On top left corner, just put in a piece
4        Piece p = piecesToSearch.remove();
5        orientTopLeftCorner(p);
6        solution[0][0] = p;
7     } else {
8        /* Get the right edge and list to match. */
9        Piece pieceToMatch = column == 0 ? solution[row - 1][0] :
10                                          solution[row][column - 1];
11       Orientation orientationToMatch = column == 0 ? Orientation.BOTTOM :
12                                                     Orientation.RIGHT;
13       Edge edgeToMatch = pieceToMatch.getEdgeWithOrientation(orientationToMatch);
14
15       /* Get matching edge. */
16       Edge edge = getMatchingEdge(edgeToMatch, piecesToSearch);
17       if (edge == null) return false; // Can't solve
18
19       /* Insert piece and edge. */
20       Orientation orientation = orientationToMatch.getOpposite();
21       setEdgeInSolution(piecesToSearch, edge, row, column, orientation);
22    }
23    return true;
24 }
25
26 boolean solve() {
27    /* Group pieces. */
28    LinkedList<Piece> cornerPieces = new LinkedList<Piece>();
29    LinkedList<Piece> borderPieces = new LinkedList<Piece>();
30    LinkedList<Piece> insidePieces = new LinkedList<Piece>();
31    groupPieces(cornerPieces, borderPieces, insidePieces);
32
33    /* Walk through puzzle, finding the piece that joins the previous one. */
34    solution = new Piece[size][size];
35    for (int row = 0; row < size; row++) {
36       for (int column = 0; column < size; column++) {
37          LinkedList<Piece> piecesToSearch = getPieceListToSearch(cornerPieces,
38             borderPieces, insidePieces, row, column);
39          if (!fitNextEdge(piecesToSearch, row, column)) {
40             return false;
41          }
42       }
43    }
44
45    return true;
46 }
```

The full code for this solution can be found in the downloadable code attachment.

7.7 **Chat Server:** 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 128

SOLUTION

Designing a chat server is a huge project, and it is certainly far beyond the scope of what could be completed in an interview. After all, teams of many people spend months or years creating a chat server. Part of your job, as a candidate, is to focus on an aspect of the problem that is reasonably broad, but focused enough that you could accomplish it during an interview. It need not match real life exactly, but it should be a fair representation of an actual implementation.

For our purposes, we'll focus on the core user management and conversation aspects: adding a user, creating a conversation, updating one's status, and so on. In the interest of time and space, we will not go into the networking aspects of the problem, or how the data actually gets pushed out to the clients.

We will assume that "friending" is mutual; I am only your contact if you are mine. Our chat system will support both group chat and one-on-one (private) chats. We will not worry about voice chat, video chat, or file transfer.

What specific actions does it need to support?

This is also something to discuss with your interviewer, but here are some ideas:

- Signing online and offline.
- Add requests (sending, accepting, and rejecting).
- Updating a status message.
- Creating private and group chats.
- Adding new messages to private and group chats.

This is just a partial list. If you have more time, you can add more actions.

What can we learn about these requirements?

We must have a concept of users, add request status, online status, and messages.

What are the core components of the system?

The system would likely consist of a database, a set of clients, and a set of servers. We won't include these parts in our object-oriented design, but we can discuss the overall view of the system.

The database will be used for more permanent storage, such as the user list or chat archives. A SQL database is a good bet, or, if we need more scalability, we could potentially use BigTable or a similar system.

For communication between the client and servers, using XML will work well. Although it's not the most compressed format (and you should point this out to your interviewer), it's nice because it's easy for both computers and humans to read. Using XML will make your debugging efforts easier—and that matters a lot.

The server will consist of a set of machines. Data will be split across machines, requiring us to potentially hop from machine to machine. When possible, we will try to replicate some data across machines to minimize the lookups. One major design constraint here is to prevent having a single point of failure. For instance,

if one machine controlled all the user sign-ins, then we'd cut off millions of users potentially if a single machine lost network connectivity.

What are the key objects and methods?

The key objects of the system will be a concept of users, conversations, and status messages. We've implemented a `UserManager` class. If we were looking more at the networking aspects of the problem, or a different component, we might have instead dived into those objects.

```
1   /* UserManager serves as a central place for core user actions. */
1   public class UserManager {
2     private static UserManager instance;
3     /* maps from a user id to a user */
4     private HashMap<Integer, User> usersById;
5
6     /* maps from an account name to a user */
7     private HashMap<String, User> usersByAccountName;
8
9     /* maps from the user id to an online user */
10    private HashMap<Integer, User> onlineUsers;
11
12    public static UserManager getInstance() {
13      if (instance == null) instance = new UserManager();
14      return instance;
15    }
16
17    public void addUser(User fromUser, String toAccountName) { ... }
18    public void approveAddRequest(AddRequest req) { ... }
19    public void rejectAddRequest(AddRequest req) { ... }
20    public void userSignedOn(String accountName) { ... }
21    public void userSignedOff(String accountName) { ... }
22  }
```

The method `receivedAddRequest`, in the `User` class, notifies User B that User A has requested to add him. User B approves or rejects the request (via `UserManager.approveAddRequest` or `rejectAddRequest`), and the `UserManager` takes care of adding the users to each other's contact lists.

The method `sentAddRequest` in the `User` class is called by `UserManager` to add an `AddRequest` to User A's list of requests. So the flow is:

1. User A clicks "add user" on the client, and it gets sent to the server.
2. User A calls `requestAddUser(User B)`.
3. This method calls `UserManager.addUser`.
4. `UserManager` calls both `User A.sentAddRequest` and `User B.receivedAddRequest`.

Again, this is just *one* way of designing these interactions. It is not the only way, or even the only "good" way.

```
1   public class User {
2     private int id;
3     private UserStatus status = null;
4
5     /* maps from the other participant's user id to the chat */
6     private HashMap<Integer, PrivateChat> privateChats;
7
8     /* list of group chats */
```

```
    private ArrayList<GroupChat> groupChats;

    /* maps from the other person's user id to the add request */
    private HashMap<Integer, AddRequest> receivedAddRequests;

    /* maps from the other person's user id to the add request */
    private HashMap<Integer, AddRequest> sentAddRequests;

    /* maps from the user id to user object */
    private HashMap<Integer, User> contacts;

    private String accountName;
    private String fullName;

    public User(int id, String accountName, String fullName) { ... }
    public boolean sendMessageToUser(User to, String content){ ... }
    public boolean sendMessageToGroupChat(int id, String cnt){...}
    public void setStatus(UserStatus status) { ... }
    public UserStatus getStatus() { ... }
    public boolean addContact(User user) { ... }
    public void receivedAddRequest(AddRequest req) { ...}
    public void sentAddRequest(AddRequest req) { ... }
    public void removeAddRequest(AddRequest req) { ... }
    public void requestAddUser(String accountName) { ... }
    public void addConversation(PrivateChat conversation) { ... }
    public void addConversation(GroupChat conversation) { ... }
    public int getId() { ... }
    public String getAccountName() { ... }
    public String getFullName() { ... }
}
```

The `Conversation` class is implemented as an abstract class, since all `Conversations` must be either a `GroupChat` or a `PrivateChat`, and since these two classes each have their own functionality.

```
public abstract class Conversation {
    protected ArrayList<User> participants;
    protected int id;
    protected ArrayList<Message> messages;

    public ArrayList<Message> getMessages() { ... }
    public boolean addMessage(Message m) { ... }
    public int getId() { ... }
}

public class GroupChat extends Conversation {
    public void removeParticipant(User user) { ... }
    public void addParticipant(User user) { ... }
}

public class PrivateChat extends Conversation {
    public PrivateChat(User user1, User user2) { ...
    public User getOtherParticipant(User primary) { ... }
}

public class Message {
    private String content;
    private Date date;
    public Message(String content, Date date) { ... }
```

```
25    public String getContent() { ... }
26    public Date getDate() { ... }
27  }
```

`AddRequest` and `UserStatus` are simple classes with little functionality. Their main purpose is to group data that other classes will act upon.

```
1   public class AddRequest {
2     private User fromUser;
3     private User toUser;
4     private Date date;
5     RequestStatus status;
6
7     public AddRequest(User from, User to, Date date) { ... }
8     public RequestStatus getStatus() { ... }
9     public User getFromUser() { ... }
10    public User getToUser() { ... }
11    public Date getDate() { ... }
12  }
13
14  public class UserStatus {
15      private String message;
16      private UserStatusType type;
17      public UserStatus(UserStatusType type, String message) { ... }
18      public UserStatusType getStatusType() { ... }
19      public String getMessage() { ... }
20  }
21
22  public enum UserStatusType {
23    Offline, Away, Idle, Available, Busy
24  }
25
26  public enum RequestStatus {
27    Unread, Read, Accepted, Rejected
28  }
```

The downloadable code attachment provides a more detailed look at these methods, including implementations for the methods shown above.

What problems would be the hardest to solve (or the most interesting)?

The following questions may be interesting to discuss with your interviewer further.

Q1: How do we know if someone is online—I mean, really, really know?

While we would like users to tell us when they sign off, we can't know for sure. A user's connection might have died, for example. To make sure that we know when a user has signed off, we might try regularly pinging the client to make sure it's still there.

Q2: How do we deal with conflicting information?

We have some information stored in the computer's memory and some in the database. What happens if they get out of sync? Which one is "right"?

Q3: How do we make our server scale?

While we designed out chat server without worrying—too much– about scalability, in real life this would be a concern. We'd need to split our data across many servers, which would increase our concern about out-of-sync data.

Q4: How we do prevent denial of service attacks?

Clients can push data to us—what if they try to DOS (denial of service) us? How do we prevent that?

7.8 **Othello:** 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.

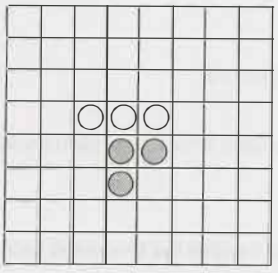
pg 128

SOLUTION

Let's start with an example. Suppose we have the following moves in an Othello game:

1. Initialize the board with two black and two white pieces in the center. The black pieces are placed at the upper left hand and lower right hand corners.
2. Play a black piece at (row 6, column 4). This flips the piece at (row 5, column 4) from white to black.
3. Play a white piece at (row 4, column 3). This flips the piece at (row 4, column 4) from black to white.

This sequence of moves leads to the board below.



The core objects in Othello are probably the game, the board, the pieces (black or white), and the players. How do we represent these with elegant object-oriented design?

Should BlackPiece and WhitePiece be classes?

At first, we might think we want to have a `BlackPiece` class and a `WhitePiece` class, which inherit from an abstract `Piece`. However, this is probably not a great idea. Each piece may flip back and forth between colors frequently, so continuously destroying and creating what is really the same object is probably not wise. It may be better to just have a `Piece` class, with a flag in it representing the current color.

Do we need separate Board and Game classes?

Strictly speaking, it may not be necessary to have both a `Game` object and a `Board` object. Keeping the objects separate allows us to have a logical separation between the board (which contains just logic

involving placing pieces) and the game (which involves times, game flow, etc.). However, the drawback is that we are adding extra layers to our program. A function may call out to a method in Game, only to have it immediately call Board. We have made the choice below to keep Game and Board separate, but you should discuss this with your interviewer.

Who keeps score?

We know we should probably have some sort of score keeping for the number of black and white pieces. But who should maintain this information? One could make a strong argument for either Game or Board maintaining this information, and possibly even for Piece (in static methods). We have implemented this with Board holding this information, since it can be logically grouped with the board. It is updated by Piece or Board calling the colorChanged and colorAdded methods within Board.

Should Game be a Singleton class?

Implementing Game as a singleton class has the advantage of making it easy for anyone to call a method within Game, without having to pass around references to the Game object.

However, making Game a singleton means it can only be instantiated once. Can we make this assumption? You should discuss this with your interviewer.

One possible design for Othello is below.

```
1  public enum Direction {
2     left, right, up, down
3  }
4
5  public enum Color {
6     White, Black
7  }
8
9  public class Game {
10    private Player[] players;
11    private static Game instance;
12    private Board board;
13    private final int ROWS = 10;
14    private final int COLUMNS = 10;
15
16    private Game() {
17       board = new Board(ROWS, COLUMNS);
18       players = new Player[2];
19       players[0] = new Player(Color.Black);
20       players[1] = new Player(Color.White);
21    }
22
23    public static Game getInstance() {
24       if (instance == null) instance = new Game();
25       return instance;
26    }
27
28    public Board getBoard() {
29       return board;
30    }
31 }
```

The Board class manages the actual pieces themselves. It does not handle much of the game play, leaving that up to the Game class.

```
1  public class Board {
2    private int blackCount = 0;
3    private int whiteCount = 0;
4    private Piece[][] board;
5
6    public Board(int rows, int columns) {
7      board = new Piece[rows][columns];
8    }
9
10   public void initialize() {
11     /* initialize center black and white pieces */
12   }
13
14   /* Attempt to place a piece of color color at (row, column). Return true if we
15    * were successful. */
16   public boolean placeColor(int row, int column, Color color) {
17     ...
18   }
19
20   /* Flips pieces starting at (row, column) and proceeding in direction d. */
21   private int flipSection(int row, int column, Color color, Direction d) { ... }
22
23   public int getScoreForColor(Color c) {
24     if (c == Color.Black) return blackCount;
25     else return whiteCount;
26   }
27
28   /* Update board with additional newPieces pieces of color newColor. Decrease
29    * score of opposite color. */
30   public void updateScore(Color newColor, int newPieces) { ... }
31 }
```

As described earlier, we implement the black and white pieces with the Piece class, which has a simple Color variable representing whether it is a black or white piece.

```
1  public class Piece {
2    private Color color;
3    public Piece(Color c) { color = c; }
4
5    public void flip() {
6      if (color == Color.Black) color = Color.White;
7      else color = Color.Black;
8    }
9
10   public Color getColor() { return color; }
11 }
```

The Player holds only a very limited amount of information. It does not even hold its own score, but it does have a method one can call to get the score. Player.getScore() will call out to the Game object to retrieve this value.

```
1  public class Player {
2    private Color color;
3    public Player(Color c) { color = c; }
4
5    public int getScore() { ... }
```

```
6
7     public boolean playPiece(int r, int c) {
8        return Game.getInstance().getBoard().placeColor(r, c, color);
9     }
10
11    public Color getColor() { return color; }
12 }
```

A fully functioning (automated) version of this code can be found in the downloadable code attachment.

Remember that in many problems, what you did is less important than *why* you did it. Your interviewer probably doesn't care much whether you chose to implement Game as a singleton or not, but she probably does care that you took the time to think about it and discuss the trade-offs.

7.9 Circular Array: Implement a `CircularArray` class that supports an array-like data structure which can be efficiently rotated. If possible, the class should use a generic type (also called a template), and should support iteration via the standard `for (Obj o : circularArray)` notation.

pg 128

SOLUTION

This problem really has two parts to it. First, we need to implement the `CircularArray` class. Second, we need to support iteration. We will address these parts separately.

Implementing the CircularArray class

One way to implement the `CircularArray` class is to actually shift the elements each time we call `rotate(int shiftRight)`. Doing this is, of course, not very efficient.

Instead, we can just create a member variable `head` which points to what should be *conceptually* viewed as the start of the circular array. Rather than shifting around the elements in the array, we just increment `head` by `shiftRight`.

The code below implements this approach.

```
1  public class CircularArray<T> {
2     private T[] items;
3     private int head = 0;
4
5     public CircularArray(int size) {
6        items = (T[]) new Object[size];
7     }
8
9     private int convert(int index) {
10       if (index < 0) {
11          index += items.length;
12       }
13       return (head + index) % items.length;
14    }
15
16    public void rotate(int shiftRight) {
17       head = convert(shiftRight);
18    }
19
20    public T get(int i) {
21       if (i < 0 || i >= items.length) {
```

```
22          throw new java.lang.IndexOutOfBoundsException("...");
23       }
24       return items[convert(i)];
25    }
26
27    public void set(int i, T item) {
28       items[convert(i)] = item;
29    }
30 }
```

There are a number of things here which are easy to make mistakes on, such as:

- In Java, we cannot create an array of the generic type. Instead, we must either cast the array or define `items` to be of type `List<T>`. For simplicity, we have done the former.
- The % operator will return a negative value when we do `negValue % posVal`. For example, `-8 % 3` is `-2`. This is different from how mathematicians would define the modulus function. We must add `items.length` to a negative index to get the correct positive result.
- We need to be sure to consistently convert the raw index to the rotated index. For this reason, we have implemented a `convert` function that is used by other methods. Even the `rotate` function uses `convert`. This is a good example of code reuse.

Now that we have the basic code for `CircularArray` out of the way, we can focus on implementing an iterator.

Implementing the Iterator Interface

The second part of this question asks us to implement the `CircularArray` class such that we can do the following:

```
1  CircularArray<String> array = ...
2  for (String s : array) { ... }
```

Implementing this requires implementing the `Iterator` interface. The details of this implementation apply to Java, but similar things can be implemented in other languages.

To implement the `Iterator` interface, we need to do the following:

- Modify the `CircularArray<T>` definition to add `implements Iterable<T>`. This will also require us to add an `iterator()` method to `CircularArray<T>`.
- Create a `CircularArrayIterator<T>` which implements `Iterator<T>`. This will also require us to implement, in the `CircularArrayIterator`, the methods `hasNext()`, `next()`, and `remove()`.

Once we've done the above items, the for loop will "magically" work.

In the code below, we have removed the aspects of `CircularArray` which were identical to the earlier implementation.

```
1  public class CircularArray<T> implements Iterable<T> {
2     ...
3     public Iterator<T> iterator() {
4        return new CircularArrayIterator<T>(this);
5     }
6
7     private class CircularArrayIterator<TI> implements Iterator<TI>{
8        /* current reflects the offset from the rotated head, not from the actual
9         * start of the raw array. */
10       private int _current = -1;
```

```
11      private TI[] _items;
12
13      public CircularArrayIterator(CircularArray<TI> array){
14         _items = array.items;
15      }
16
17      @Override
18      public boolean hasNext() {
19         return _current < items.length - 1;
20      }
21
22      @Override
23      public TI next() {
24         _current++;
25         TI item = (TI) _items[convert(_current)];
26         return item;
27      }
28
29      @Override
30      public void remove() {
31         throw new UnsupportedOperationException("...");
32      }
33   }
34 }
```

In the above code, note that the first iteration of the for loop will call `hasNext()` and then `next()`. Be very sure that your implementation will return the correct values here.

When you get a problem like this one in an interview, there's a good chance you don't remember exactly what the various methods and interfaces are called. In this case, work through the problem as well as you can. If you can reason out what sorts of methods one might need, that alone will show a good degree of competency.

7.10 **Minesweeper:** Design and implement a text-based Minesweeper game. Minesweeper is the classic single-player computer game where an NxN grid has B mines (or bombs) hidden across the grid. The remaining cells are either blank or have a number behind them. The numbers reflect the number of bombs in the surrounding eight cells. The user then uncovers a cell. If it is a bomb, the player loses. If it is a number, the number is exposed. If it is a blank cell, this cell and all adjacent blank cells (up to and including the surrounding numeric cells) are exposed. The player wins when all non-bomb cells are exposed. The player can also flag certain places as potential bombs. This doesn't affect game play, other than to block the user from accidentally clicking a cell that is thought to have a bomb. (Tip for the reader: if you're not familiar with this game, please play a few rounds online first.)

This is a fully exposed board with 3 bombs. This is not shown to the user.

	1	1	1			
	1	*	1			
	2	2	2			
	1	*	1			
	1	1	1			
			1	1	1	
			1	*	1	

The player initially sees a board with nothing exposed.

?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?

Clicking on cell (row = 1, col = 0) would expose this:

	1	?	?	?	?	?
	1	?	?	?	?	?
	2	?	?	?	?	?
	1	?	?	?	?	?
	1	1	1	?	?	?
			1	?	?	?
			1	?	?	?

The user wins when everything other than bombs has been exposed.

	1	1	1			
	1	?	1			
	2	2	2			
	1	?	1			
	1	1	1			
			1	1	1	
			1	?	1	

pg 129

SOLUTION

Writing an entire game—even a text-based one—would take far longer than the allotted time you have in an interview. This doesn't mean that it's not fair game as a question. It just means that your interviewer's expectation will not be that you actually write all of this in an interview. It also means that you need to focus on getting the key ideas—or structure—out.

Let's start with what the classes are. We certainly want a `Cell` class as well as a `Board` class. We also probably want to have a `Game` class.

We could potentially merge `Board` and `Game` together, but it's probably best to keep them separate. Err towards more organization, not less. `Board` can hold the list of `Cell` objects and do some basic moves with flipping over cells. `Game` will hold the game state and handle user input.

Design: Cell

`Cell` will need to have knowledge of whether it's a bomb, a number, or a blank. We could potentially subclass `Cell` to hold this data, but I'm not sure that offers us much benefit.

We could also have an enum `TYPE {BOMB, NUMBER, BLANK}` to describe the type of cell. We've chosen not to do this because `BLANK` is really a type of `NUMBER` cell, where the number is 0. It's sufficient to just have an `isBomb` flag.

It's okay to have made different choices here. These aren't the only good choices. Explain the choices you make and their tradeoffs with your interviewer.

We also need to store state for whether the cell is exposed or not. We probably do not want to subclass `Cell` for `ExposedCell` and `UnexposedCell`. This is a bad idea because `Board` holds a reference to the cells, and we'd have to change the reference when we flip a cell. And then what if other objects reference the instance of `Cell`?

It's better to just have a boolean flag for `isExposed`. We'll do a similar thing for `isGuess`.

```
1  public class Cell {
2    private int row;
3    private int column;
4    private boolean isBomb;
5    private int number;
6    private boolean isExposed = false;
7    private boolean isGuess = false;
8
9    public Cell(int r, int c) { ... }
10
11   /* Getters and setters for above variables. */
12   ...
13
14   public boolean flip() {
15     isExposed = true;
16     return !isBomb;
17   }
18
19   public boolean toggleGuess() {
20     if (!isExposed) {
21       isGuess = !isGuess;
22     }
23     return isGuess;
24   }
25
26   /* Full code can be found in downloadable code solutions. */
27 }
```

Design: Board

`Board` will need to have an array of all the `Cell` objects. A two-dimension array will work just fine.

We'll probably want `Board` to keep state of how many unexposed cells there are. We'll track this as we go, so we don't have to continuously count it.

`Board` will also handle some of the basic algorithms:

- Initializing the board and laying out the bombs.
- Flipping a cell.

- Expanding blank areas.

It will receive the game plays from the Game object and carry them out. It will then need to return the result of the play, which could be any of {clicked a bomb and lost, clicked out of bounds, clicked an already exposed area, clicked a blank area and still playing, clicked a blank area and won, clicked a number and won}. This is really two different items that need to be returned: successful (whether or not the play was successfully made) and a game state (won, lost, playing). We'll use an additional `GamePlayResult` to return this data.

We'll also use a `GamePlay` class to hold the move that the player plays. We need to use a row, column, and then a flag to indicate whether this was an actual flip or the user was just marking this as a "guess" at a possible bomb.

The basic skeleton of this class might look something like this:

```
1  public class Board {
2     private int nRows;
3     private int nColumns;
4     private int nBombs = 0;
5     private Cell[][] cells;
6     private Cell[] bombs;
7     private int numUnexposedRemaining;
8
9     public Board(int r, int c, int b) { ... }
10
11    private void initializeBoard() { ... }
12    private boolean flipCell(Cell cell) { ... }
13    public void expandBlank(Cell cell) { ... }
14    public UserPlayResult playFlip(UserPlay play) { ... }
15    public int getNumRemaining() { return numUnexposedRemaining; }
16 }
17
18 public class UserPlay {
19    private int row;
20    private int column;
21    private boolean isGuess;
22    /* constructor, getters, setters. */
23 }
24
25 public class UserPlayResult {
26    private boolean successful;
27    private Game.GameState resultingState;
28    /* constructor, getters, setters. */
29 }
```

Design: Game

The Game class will store references to the board and hold the game state. It also takes the user input and sends it off to Board.

```
1  public class Game {
2     public enum GameState { WON, LOST, RUNNING }
3
4     private Board board;
5     private int rows;
6     private int columns;
7     private int bombs;
8     private GameState state;
```

```
9
10    public Game(int r, int c, int b) { ... }
11
12    public boolean initialize() { ... }
13    public boolean start() { ... }
14    private boolean playGame() { ... } // Loops until game is over.
15 }
```

Algorithms

This is the basic object-oriented design in our code. Our interviewer might ask us now to implement a few of the most interesting algorithms.

In this case, the three interesting algorithms is the initialization (placing the bombs randomly), setting the values of the numbered cells, and expanding the blank region.

Placing the Bombs

To place the bombs, we could randomly pick a cell and then place a bomb if it's still available, and otherwise pick a different location for it. The problem with this is that if there are a lot of bombs, it could get very slow. We could end up in a situation where we repeatedly pick cells with bombs.

To get around this, we could take an approach similar to the card deck shuffling problem (pg 531). We could place the K bombs in the first K cells and then shuffle all the cells around.

Shuffling an array operates by iterating through the array from `i = 0` through `N-1`. For each `i`, we pick a random index between `i` and `N-1` and swap it with that index.

To shuffle a grid, we do a very similar thing, just converting the index into a row and column location.

```
1  void shuffleBoard() {
2    int nCells = nRows * nColumns;
3    Random random = new Random();
4    for (int index1 = 0; index1 < nCells; index1++) {
5      int index2 = index1 + random.nextInt(nCells - index1);
6      if (index1 != index2) {
7        /* Get cell at index1. */
8        int row1 = index1 / nColumns;
9        int column1 = (index1 - row1 * nColumns) % nColumns;
10       Cell cell1 = cells[row1][column1];
11
12       /* Get cell at index2. */
13       int row2 = index2 / nColumns;
14       int column2 = (index2 - row2 * nColumns) % nColumns;
15       Cell cell2 = cells[row2][column2];
16
17       /* Swap. */
18       cells[row1][column1] = cell2;
19       cell2.setRowAndColumn(row1, column1);
20       cells[row2][column2] = cell1;
21       cell1.setRowAndColumn(row2, column2);
22     }
23   }
24 }
```

Setting the Numbered Cells

Once the bombs have been placed, we need to set the values of the numbered cells. We could go through each cell and check how many bombs are around it. This would work, but it's actually a bit slower than is necessary.

Instead, we can go to each bomb and increment each cell around it. For example, cells with 3 bombs will get `incrementNumber` called three times on them and will wind up with a number of 3.

```
1  /* Set the cells around the bombs to the right number. Although the bombs have
2   * been shuffled, the reference in the bombs array is still to same object. */
3  void setNumberedCells() {
4    int[][] deltas = { // Offsets of 8 surrounding cells
5         {-1, -1}, {-1, 0}, {-1, 1},
6         { 0, -1},          { 0, 1},
7         { 1, -1}, { 1, 0}, { 1, 1}
8    };
9    for (Cell bomb : bombs) {
10     int row = bomb.getRow();
11     int col = bomb.getColumn();
12     for (int[] delta : deltas) {
13       int r = row + delta[0];
14       int c = col + delta[1];
15       if (inBounds(r, c)) {
16         cells[r][c].incrementNumber();
17       }
18     }
19   }
20 }
```

Expanding a Blank Region

Expanding the blank region could be done either iteratively or recursively. We implemented it iteratively.

You can think about this algorithm like this: each blank cell is surrounded by either blank cells or numbered cells (never a bomb). All need to be flipped. But, if you're flipping a blank cell, you also need to add the blank cells to a queue, to flip their neighboring cells.

```
1  void expandBlank(Cell cell) {
2    int[][] deltas = {
3         {-1, -1}, {-1, 0}, {-1, 1},
4         { 0, -1},          { 0, 1},
5         { 1, -1}, { 1, 0}, { 1, 1}
6    };
7
8    Queue<Cell> toExplore = new LinkedList<Cell>();
9    toExplore.add(cell);
10
11   while (!toExplore.isEmpty()) {
12     Cell current = toExplore.remove();
13
14     for (int[] delta : deltas) {
15       int r = current.getRow() + delta[0];
16       int c = current.getColumn() + delta[1];
17
18       if (inBounds(r, c)) {
19         Cell neighbor = cells[r][c];
20         if (flipCell(neighbor) && neighbor.isBlank()) {
21           toExplore.add(neighbor);
```

```
            }
        }
    }
}
```

You could instead implement this algorithm recursively. In this algorithm, rather than adding the cell to a queue, you would make a recursive call.

Your implementation of these algorithms could vary substantially depending on your class design.

7.11 File System: 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 129

SOLUTION

Many candidates may see this problem and instantly panic. A file system seems so low level!

However, there's no need to panic. If we think through the components of a file system, we can tackle this problem just like any other object-oriented design question.

A file system, in its most simplistic version, consists of `Files` and `Directories`. Each `Directory` contains a set of `Files` and `Directories`. Since `Files` and `Directories` share so many characteristics, we've implemented them such that they inherit from the same class, `Entry`.

```
1  public abstract class Entry {
2    protected Directory parent;
3    protected long created;
4    protected long lastUpdated;
5    protected long lastAccessed;
6    protected String name;
7
8    public Entry(String n, Directory p) {
9      name = n;
10     parent = p;
11     created = System.currentTimeMillis();
12     lastUpdated = System.currentTimeMillis();
13     lastAccessed = System.currentTimeMillis();
14   }
15
16   public boolean delete() {
17     if (parent == null) return false;
18     return parent.deleteEntry(this);
19   }
20
21   public abstract int size();
22
23   public String getFullPath() {
24     if (parent == null) return name;
25     else return parent.getFullPath() + "/" + name;
26   }
27
28   /* Getters and setters. */
29   public long getCreationTime() { return created; }
30   public long getLastUpdatedTime() { return lastUpdated; }
31   public long getLastAccessedTime() { return lastAccessed; }
```

```
32    public void changeName(String n) { name = n; }
33    public String getName() { return name; }
34  }
35
36  public class File extends Entry {
37    private String content;
38    private int size;
39
40    public File(String n, Directory p, int sz) {
41      super(n, p);
42      size = sz;
43    }
44
45    public int size() { return size; }
46    public String getContents() { return content; }
47    public void setContents(String c) { content = c; }
48  }
49
50  public class Directory extends Entry {
51    protected ArrayList<Entry> contents;
52
53    public Directory(String n, Directory p) {
54      super(n, p);
55      contents = new ArrayList<Entry>();
56    }
57
58    public int size() {
59      int size = 0;
60      for (Entry e : contents) {
61        size += e.size();
62      }
63      return size;
64    }
65
66    public int numberOfFiles() {
67      int count = 0;
68      for (Entry e : contents) {
69        if (e instanceof Directory) {
70          count++; // Directory counts as a file
71          Directory d = (Directory) e;
72          count += d.numberOfFiles();
73        } else if (e instanceof File) {
74          count++;
75        }
76      }
77      return count;
78    }
79
80    public boolean deleteEntry(Entry entry) {
81      return contents.remove(entry);
82    }
83
84    public void addEntry(Entry entry) {
85      contents.add(entry);
86    }
87
```

```
88    protected ArrayList<Entry> getContents() { return contents; }
89  }
```

Alternatively, we could have implemented `Directory` such that it contains separate lists for files and subdirectories. This makes the `numberOfFiles()` method a bit cleaner, since it doesn't need to use the `instanceof` operator, but it does prohibit us from cleanly sorting files and directories by dates or names.

7.12 Hash Table: Design and implement a hash table which uses chaining (linked lists) to handle collisions.

pg 129

SOLUTION

Suppose we are implementing a hash table that looks like `Hash<K, V>`. That is, the hash table maps from objects of type K to objects of type V.

At first, we might think our data structure would look something like this:

```
1  class Hash<K, V> {
2    LinkedList<V>[] items;
3    public void put(K key, V value) { ... }
4    public V get(K key) { ... }
5  }
```

Note that `items` is an array of linked lists, where `items[i]` is a linked list of all objects with keys that map to index `i` (that is, all the objects that collided at `i`).

This would seem to work until we think more deeply about collisions.

Suppose we have a very simple hash function that uses the string length.

```
1  int hashCodeOfKey(K key) {
2    return key.toString().length() % items.length;
3  }
```

The keys `jim` and `bob` will map to the same index in the array, even though they are different keys. We need to search through the linked list to find the actual object that corresponds to these keys. But how would we do that? All we've stored in the linked list is the value, not the original key.

This is why we need to store both the value and the original key.

One way to do that is to create another object called `Cell` which pairs keys and values. With this implementation, our linked list is of type `Cell`.

The code below uses this implementation.

```
1  public class Hasher<K, V> {
2    /* Linked list node class. Used only within hash table. No one else should get
3     * access to this. Implemented as doubly linked list. */
4    private static class LinkedListNode<K, V> {
5      public LinkedListNode<K, V> next;
6      public LinkedListNode<K, V> prev;
7      public K key;
8      public V value;
9      public LinkedListNode(K k, V v) {
10       key = k;
11       value = v;
12     }
13   }
14
```

```
15    private ArrayList<LinkedListNode<K, V>> arr;
16    public Hasher(int capacity) {
17       /* Create list of linked lists at a particular size. Fill list with null
18        * values, as it's the only way to make the array the desired size. */
19       arr = new ArrayList<LinkedListNode<K, V>>();
20       arr.ensureCapacity(capacity); // Optional optimization
21       for (int i = 0; i < capacity; i++) {
22          arr.add(null);
23       }
24    }
25
26    /* Insert key and value into hash table. */
27    public void put(K key, V value) {
28       LinkedListNode<K, V> node = getNodeForKey(key);
29       if (node != null) { // Already there
30          node.value = value; // just update the value.
31          return;
32       }
33
34       node = new LinkedListNode<K, V>(key, value);
35       int index = getIndexForKey(key);
36       if (arr.get(index) != null) {
37          node.next = arr.get(index);
38          node.next.prev = node;
39       }
40       arr.set(index, node);
41    }
42
43    /* Remove node for key. */
44    public void remove(K key) {
45       LinkedListNode<K, V> node = getNodeForKey(key);
46       if (node.prev != null) {
47          node.prev.next = node.next;
48       } else {
49          /* Removing head - update. */
50          int hashKey = getIndexForKey(key);
51          arr.set(hashKey, node.next);
52       }
53
54       if (node.next != null) {
55          node.next.prev = node.prev;
56       }
57    }
58
59    /* Get value for key. */
60    public V get(K key) {
61       LinkedListNode<K, V> node = getNodeForKey(key);
62       return node == null ? null : node.value;
63    }
64
65    /* Get linked list node associated with a given key. */
66    private LinkedListNode<K, V> getNodeForKey(K key) {
67       int index = getIndexForKey(key);
68       LinkedListNode<K, V> current = arr.get(index);
69       while (current != null) {
70          if (current.key == key) {
```

```
71              return current;
72            }
73            current = current.next;
74          }
75          return null;
76        }
77
78        /* Really naive function to map a key to an index. */
79        public int getIndexForKey(K key) {
80          return Math.abs(key.hashCode() % arr.size());
81        }
82    }
83
```

Alternatively, we could implement a similar data structure (a key->value lookup) with a binary search tree as the underlying data structure. Retrieving an element will no longer be O(1) (although, technically, this implementation is not O(1) if there are many collisions), but it prevents us from creating an unnecessarily large array to hold items.

8

Solutions to Recursion and Dynamic Programming

8.1 Triple Step: 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 134

SOLUTION

Let's think about this with the following question: What is the very last step that is done?

The very last hop the child makes—the one that lands her on the nth step—was either a 3-step hop, a 2-step hop, or a 1-step hop.

How many ways then are there to get up to the nth step? We don't know yet, but we can relate it to some subproblems.

If we thought about all of the paths to the nth step, we could just build them off the paths to the three previous steps. We can get up to the nth step by any of the following:

- Going to the (n-1)st step and hopping 1 step.
- Going to the (n-2)nd step and hopping 2 steps.
- Going to the (n-3)rd step and hopping 3 steps.

Therefore, we just need to add the number of these paths together.

Be very careful here. A lot of people want to multiply them. Multiplying one path with another would signify taking one path and then taking the other. That's not what's happening here.

Brute Force Solution

This is a fairly straightforward algorithm to implement recursively. We just need to follow logic like this:

```
countWays(n-1) + countWays(n-2) + countWays(n-3)
```

The one tricky bit is defining the base case. If we have 0 steps to go (we're currently standing on the step), are there zero paths to that step or one path?

That is, what is `countWays(0)`? Is it 1 or 0?

You could define it either way. There is no "right" answer here.

However, it's a lot easier to define it as 1. If you defined it as 0, then you would need some additional base cases (or else you'd just wind up with a series of 0s getting added).

A simple implementation of this code is below.

```
1  int countWays(int n) {
2    if (n < 0) {
3      return 0;
4    } else if (n == 0) {
5      return 1;
6    } else {
7      return countWays(n-1) + countWays(n-2) + countWays(n-3);
8    }
9  }
```

Like the Fibonacci problem, the runtime of this algorithm is exponential (roughly $O(3^n)$), since each call branches out to three more calls.

Memoization Solution

The previous solution for `countWays` is called many times for the same values, which is unnecessary. We can fix this through memoization.

Essentially, if we've seen this value of n before, return the cached value. Each time we compute a fresh value, add it to the cache.

Typically we use a `HashMap<Integer, Integer>` for a cache. In this case, the keys will be exactly 1 through n. It's more compact to use an integer array.

```
1  int countWays(int n) {
2    int[] memo = new int[n + 1];
3    Arrays.fill(memo, -1);
4    return countWays(n, memo);
5  }
6
7  int countWays(int n, int[] memo) {
8    if (n < 0) {
9      return 0;
10   } else if (n == 0) {
11     return 1;
12   } else if (memo[n] > -1) {
13     return memo[n];
14   } else {
15     memo[n] = countWays(n - 1, memo) + countWays(n - 2, memo) +
16               countWays(n - 3, memo);
17     return memo[n];
18   }
19 }
```

Regardless of whether or not you use memoization, note that the number of ways will quickly overflow the bounds of an integer. By the time you get to just `n = 37`, the result has already overflowed. Using a `long` will delay, but not completely solve, this issue.

It is great to communicate this issue to your interviewer. He probably won't ask you to work around it (although you could, with a `BigInteger` class), but it's nice to demonstrate that you think about these issues.

8.2 **Robot in a Grid:** Imagine a robot sitting on the upper left corner of grid with r rows and c columns. The robot can only move in two directions, right and down, but certain cells 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 135

SOLUTION

If we picture this grid, the only way to move to spot `(r,c)` is by moving to one of the adjacent spots: `(r-1,c)` or `(r,c-1)`. So, we need to find a path to either `(r-1,c)` or `(r,c-1)`.

How do we find a path to those spots? To find a path to `(r-1,c)` or `(r,c-1)`, we need to move to one of its adjacent cells. So, we need to find a path to a spot adjacent to `(r-1,c)`, which are coordinates `(r-2,c)` and `(r-1,c-1)`, or a spot adjacent to `(r,c-1)`, which are spots `(r-1,c-1)` and `(r,c-2)`. Observe that we list the point `(r-1,c-1)` twice; we'll discuss that issue later.

> Tip: A lot of people use the variable names x and y when dealing with two-dimensional arrays. This can actually cause some bugs. People tend to think about x as the first coordinate in the matrix and y as the second coordinate (e.g., `matrix[x][y]`). But, this isn't really correct. The first coordinate is usually thought of as the row number, which is in fact the y value (it goes vertically!). You should write `matrix[y][x]`. Or, just make your life easier by using r (row) and c (column) instead.

So then, to find a path from the origin, we just work backwards like this. Starting from the last cell, we try to find a path to each of its adjacent cells. The recursive code below implements this algorithm.

```
1  ArrayList<Point> getPath(boolean[][] maze) {
2    if (maze == null || maze.length == 0) return null;
3    ArrayList<Point> path = new ArrayList<Point>();
4    if (getPath(maze, maze.length - 1, maze[0].length - 1, path)) {
5      return path;
6    }
7    return null;
8  }
9
10 boolean getPath(boolean[][] maze, int row, int col, ArrayList<Point> path) {
11   /* If out of bounds or not available, return.*/
12   if (col < 0 || row < 0 || !maze[row][col]) {
13     return false;
14   }
15
16   boolean isAtOrigin = (row == 0) && (col == 0);
17
18   /* If there's a path from the start to here, add my location. */
19   if (isAtOrigin || getPath(maze, row, col - 1, path) ||
20       getPath(maze, row - 1, col, path)) {
21     Point p = new Point(row, col);
22     path.add(p);
23     return true;
24   }
25
26   return false;
27 }
```

This solution is $O(2^{r+c})$, since each path has r+c steps and there are two choices we can make at each step.

We should look for a faster way.

Often, we can optimize exponential algorithms by finding duplicate work. What work are we repeating?

If we walk through the algorithm, we'll see that we are visiting squares multiple times. In fact, we visit each square many, many times. After all, we have `rc` squares but we're doing $O(2^{r+c})$ work. If we were only visiting each square once, we would probably have an algorithm that was $O(rc)$ (unless we were somehow doing a lot of work during each visit).

How does our current algorithm work? To find a path to `(r,c)`, we look for a path to an adjacent coordinate: `(r-1,c)` or `(r,c-1)`. Of course, if one of those squares is off limits, we ignore it. Then, we look at their adjacent coordinates: `(r-2,c)`, `(r-1,c-1)`, `(r-1,c-1)`, and `(r,c-2)`. The spot `(r-1,c-1)` appears twice, which means that we're duplicating effort. Ideally, we should remember that we already visited `(r-1,c-1)` so that we don't waste our time.

This is what the dynamic programming algorithm below does.

```
1  ArrayList<Point> getPath(boolean[][] maze) {
2    if (maze = =null || maze.length == 0) return null;
3    ArrayList<Point> path = new ArrayList<Point>();
4    HashSet<Point> failedPoints = new HashSet<Point>();
5    if (getPath(maze, maze.length - 1, maze[0].length - 1, path, failedPoints)) {
6      return path;
7    }
8    return null;
9  }
10
11 boolean getPath(boolean[][] maze, int row, int col, ArrayList<Point> path,
12                 HashSet<Point> failedPoints) {
13   /* If out of bounds or not available, return.*/
14   if (col < 0 || row < 0 || !maze[row][col]) {
15     return false;
16   }
17
18   Point p = new Point(row, col);
19
20   /* If we've already visited this cell, return. */
21   if (failedPoints.contains(p)) {
22     return false;
23   }
24
25   boolean isAtOrigin = (row == 0) && (col == 0);
26
27   /* If there's a path from start to my current location, add my location.*/
28   if (isAtOrigin || getPath(maze, row, col - 1, path, failedPoints) ||
29       getPath(maze, row - 1, col, path, failedPoints)) {
30     path.add(p);
31     return true;
32   }
33
34   failedPoints.add(p); // Cache result
35   return false;
36 }
```

This simple change will make our code run substantially faster. The algorithm will now take $O(XY)$ time because we hit each cell just once.

8.3 **Magic Index:** A magic index in an array A[1...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 135

SOLUTION

Immediately, the brute force solution should jump to mind—and there's no shame in mentioning it. We simply iterate through the array, looking for an element which matches this condition.

```
1  int magicSlow(int[] array) {
2    for (int i = 0; i < array.length; i++) {
3      if (array[i] == i) {
4        return i;
5      }
6    }
7    return -1;
8  }
```

Given that the array is sorted, though, it's very likely that we're supposed to use this condition.

We may recognize that this problem sounds a lot like the classic binary search problem. Leveraging the Pattern Matching approach for generating algorithms, how might we apply binary search here?

In binary search, we find an element k by comparing it to the middle element, x, and determining if k would land on the left or the right side of x.

Building off this approach, is there a way that we can look at the middle element to determine where a magic index might be? Let's look at a sample array:

-40	-20	-1	1	2	3	5	7	9	12	13
0	1	2	3	4	5	6	7	8	9	10

When we look at the middle element A[5] = 3, we know that the magic index must be on the right side, since A[mid] < mid.

Why couldn't the magic index be on the left side? Observe that when we move from i to i-1, the value at this index must decrease by at least 1, if not more (since the array is sorted and all the elements are distinct). So, if the middle element is already too small to be a magic index, then when we move to the left, subtracting k indexes and (at least) k values, all subsequent elements will also be too small.

We continue to apply this recursive algorithm, developing code that looks very much like binary search.

```
1  int magicFast(int[] array) {
2    return magicFast(array, 0, array.length - 1);
3  }
4
5  int magicFast(int[] array, int start, int end) {
6    if (end < start) {
7      return -1;
8    }
9    int mid = (start + end) / 2;
10   if (array[mid] == mid) {
11     return mid;
12   } else if (array[mid] > mid){
```

```
13        return magicFast(array, start, mid - 1);
14      } else {
15        return magicFast(array, mid + 1, end);
16      }
17    }
```

Follow Up: What if the elements are not distinct?

If the elements are not distinct, then this algorithm fails. Consider the following array:

-10	-5	2	2	2	3	4	7	9	12	13
0	1	2	3	4	5	6	7	8	9	10

When we see that `A[mid] < mid`, we cannot conclude which side the magic index is on. It could be on the right side, as before. Or, it could be on the left side (as it, in fact, is).

Could it be *anywhere* on the left side? Not exactly. Since `A[5] = 3`, we know that `A[4]` couldn't be a magic index. `A[4]` would need to be 4 to be the magic index, but `A[4]` must be less than or equal to `A[5]`.

In fact, when we see that `A[5] = 3`, we'll need to recursively search the right side as before. But, to search the left side, we can skip a bunch of elements and only recursively search elements `A[0]` through `A[3]`. `A[3]` is the first element that could be a magic index.

The general pattern is that we compare `midIndex` and `midValue` for equality first. Then, if they are not equal, we recursively search the left and right sides as follows:

- Left side: search indices `start` through `Math.min(midIndex - 1, midValue)`.
- Right side: search indices `Math.max(midIndex + 1, midValue)` through `end`.

The code below implements this algorithm.

```
1   int magicFast(int[] array) {
2     return magicFast(array, 0, array.length - 1);
3   }
4
5   int magicFast(int[] array, int start, int end) {
6     if (end < start) return -1;
7
8     int midIndex = (start + end) / 2;
9     int midValue = array[midIndex];
10    if (midValue == midIndex) {
11      return midIndex;
12    }
13
14    /* Search left * /
15    int leftIndex = Math.min(midIndex - 1, midValue);
16    int left = magicFast(array, start, leftIndex);
17    if (left >= 0) {
18      return left;
19    }
20
21    /* Search right * /
22    int rightIndex = Math.max(midIndex + 1, midValue);
23    int right = magicFast(array, rightIndex, end);
24
25    return right;
26  }
```

Note that in the above code, if the elements are all distinct, the method operates almost identically to the first solution.

8.4 Power Set: Write a method to return all subsets of a set.

pg 135

SOLUTION

We should first have some reasonable expectations of our time and space complexity.

How many subsets of a set are there? When we generate a subset, each element has the "choice" of either being in there or not. That is, for the first element, there are two choices: it is either in the set, or it is not. For the second, there are two, etc. So, doing $\{2 * 2 * \ldots\}$ n times gives us 2^n subsets.

Assuming that we're going to be returning a list of subsets, then our best case time is actually the total number of elements across all of those subsets. There are 2^n subsets and each of the n elements will be contained in half of the subsets (which 2^{n-1} subsets). Therefore, the total number of elements across all of those subsets is $n * 2^{n-1}$.

We will not be able to beat $O(n2^n)$ in space or time complexity.

The subsets of $\{a_1, a_2, \ldots, a_n\}$ are also called the powerset, $P(\{a_1, a_2, \ldots, a_n\})$, or just $P(n)$.

Solution #1: Recursion

This problem is a good candidate for the Base Case and Build approach. Imagine that we are trying to find all subsets of a set like $S = \{a_1, a_2, \ldots, a_n\}$. We can start with the Base Case.

Base Case: $n = 0$.

There is just one subset of the empty set: $\{\}$.

Case: $n = 1$.

There are two subsets of the set $\{a_1\}$: $\{\}, \{a_1\}$.

Case: $n = 2$.

There are four subsets of the set $\{a_1, a_2\}$: $\{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$.

Case: $n = 3$.

Now here's where things get interesting. We want to find a way of generating the solution for $n = 3$ based on the prior solutions.

What is the difference between the solution for $n = 3$ and the solution for $n = 2$? Let's look at this more deeply:

$P(2) = \{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$
$P(3) = \{\}, \{a_1\}, \{a_2\}, \{a_3\}, \{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$

The difference between these solutions is that $P(2)$ is missing all the subsets containing a_3.

$P(3) - P(2) = \{a_3\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$

How can we use $P(2)$ to create $P(3)$? We can simply clone the subsets in $P(2)$ and add a_3 to them:

$P(2) = \{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$
$P(2) + a_3 = \{a_3\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$

When merged together, the lines above make P(3).

Case: n > 0

Generating P(n) for the general case is just a simple generalization of the above steps. We compute P(n-1), clone the results, and then add a_n to each of these cloned sets.

The following code implements this algorithm:

```
1  ArrayList<ArrayList<Integer>> getSubsets(ArrayList<Integer> set, int index) {
2    ArrayList<ArrayList<Integer>> allsubsets;
3    if (set.size() == index) { // Base case - add empty set
4      allsubsets = new ArrayList<ArrayList<Integer>>();
5      allsubsets.add(new ArrayList<Integer>()); // Empty set
6    } else {
7      allsubsets = getSubsets(set, index + 1);
8      int item = set.get(index);
9      ArrayList<ArrayList<Integer>> moresubsets =
10       new ArrayList<ArrayList<Integer>>();
11     for (ArrayList<Integer> subset : allsubsets) {
12       ArrayList<Integer> newsubset = new ArrayList<Integer>();
13       newsubset.addAll(subset); //
14       newsubset.add(item);
15       moresubsets.add(newsubset);
16     }
17     allsubsets.addAll(moresubsets);
18   }
19   return allsubsets;
20 }
```

This solution will be $O(n2^n)$ in time and space, which is the best we can do. For a slight optimization, we could also implement this algorithm iteratively.

Solution #2: Combinatorics

While there's nothing wrong with the above solution, there's another way to approach it.

Recall that when we're generating a set, we have two choices for each element: (1) the element is in the set (the "yes" state) or (2) the element is not in the set (the "no" state). This means that each subset is a sequence of yeses / nos—e.g., "yes, yes, no, no, yes, no"

This gives us 2^n possible subsets. How can we iterate through all possible sequences of "yes" / "no" states for all elements? If each "yes" can be treated as a 1 and each "no" can be treated as a 0, then each subset can be represented as a binary string.

Generating all subsets, then, really just comes down to generating all binary numbers (that is, all integers). We iterate through all numbers from 0 to 2^n (exclusive) and translate the binary representation of the numbers into a set. Easy!

```
1  ArrayList<ArrayList<Integer>> getSubsets2(ArrayList<Integer> set) {
2    ArrayList<ArrayList<Integer>> allsubsets = new ArrayList<ArrayList<Integer>>();
3    int max = 1 << set.size(); /* Compute 2^n */
4    for (int k = 0; k < max; k++) {
5      ArrayList<Integer> subset = convertIntToSet(k, set);
6      allsubsets.add(subset);
7    }
8    return allsubsets;
9  }
```

```
10
11 ArrayList<Integer> convertIntToSet(int x, ArrayList<Integer> set) {
12    ArrayList<Integer> subset = new ArrayList<Integer>();
13    int index = 0;
14    for (int k = x; k > 0; k >>= 1) {
15       if ((k & 1) == 1) {
16          subset.add(set.get(index));
17       }
18       index++;
19    }
20    return subset;
21 }
```

There's nothing substantially better or worse about this solution compared to the first one.

8.5 **Recursive Multiply:** Write a recursive function to multiply two positive integers without using the * operator (or / operator). You can use addition, subtraction, and bit shifting, but you should minimize the number of those operations.

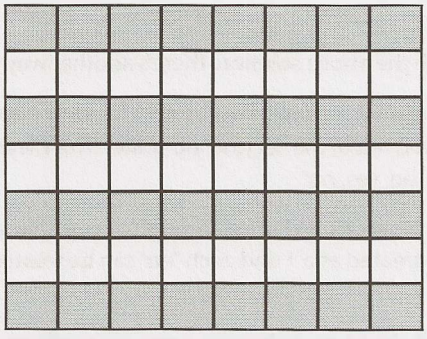
pg 135

SOLUTION

Let's pause for a moment and think about what it means to do multiplication.

> This is a good approach for a lot of interview questions. It's often useful to think about what it really means to do something, even when it's pretty obvious.

We can think about multiplying 8x7 as doing 8+8+8+8+8+8+8 (or adding 7 eight times). We can also think about it as the number of squares in an 8x7 grid.



Solution #1

How would we count the number of squares in this grid? We could just count each cell. That's pretty slow, though.

Alternatively, we could count half the squares and then double it (by adding this count to itself). To count half the squares, we repeat the same process.

Of course, this "doubling" only works if the number is in fact even. When it's not even, we need to do the counting/summing from scratch.

```
1  int minProduct(int a, int b) {
2     int bigger = a < b ? b : a;
```

```
  int smaller = a < b ? a : b;
  return minProductHelper(smaller, bigger);
}

int minProductHelper(int smaller, int bigger) {
  if (smaller == 0) { // 0 x bigger = 0
    return 0;
  } else if (smaller == 1) { // 1 x bigger = bigger
    return bigger;
  }

  /* Compute half. If uneven, compute other half. If even, double it. */
  int s = smaller >> 1; // Divide by 2
  int side1 = minProduct(s, bigger);
  int side2 = side1;
  if (smaller % 2 == 1) {
    side2 = minProductHelper(smaller - s, bigger);
  }

  return side1 + side2;
}
```

Can we do better? Yes.

Solution #2

If we observe how the recursion operates, we'll notice that we have duplicated work. Consider this example:

```
        minProduct(17, 23)
             minProduct(8, 23)
                 minProduct(4, 23) * 2
                     ...
           + minProduct(9, 23)
                 minProduct(4, 23)
                     ...
             + minProduct(5, 23)
                     ...
```

The second call to `minProduct(4, 23)` is unaware of the prior call, and so it repeats the same work. We should cache these results.

```
int minProduct(int a, int b) {
  int bigger = a < b ? b : a;
  int smaller = a < b ? a : b;

  int memo[] = new int[smaller + 1];
  return minProduct(smaller, bigger, memo);
}

int minProduct(int smaller, int bigger, int[] memo) {
  if (smaller == 0) {
    return 0;
  } else if (smaller == 1) {
    return bigger;
  } else if (memo[smaller] > 0) {
    return memo[smaller];
  }

  /* Compute half. If uneven, compute other half. If even, double it. */
```

```
    int s = smaller >> 1; // Divide by 2
    int side1 = minProduct(s, bigger, memo); // Compute half
    int side2 = side1;
    if (smaller % 2 == 1) {
       side2 = minProduct(smaller - s, bigger, memo);
    }

    /* Sum and cache.*/
    memo[smaller] = side1 + side2;
    return memo[smaller];
}
```

We can still make this a bit faster.

Solution #3

One thing we might notice when we look at this code is that a call to `minProduct` on an even number is much faster than one on an odd number. For example, if we call `minProduct(30, 35)`, then we'll just do `minProduct(15, 35)` and double the result. However, if we do `minProduct(31, 35)`, then we'll need to call `minProduct(15, 35)` and `minProduct(16, 35)`.

This is unnecessary. Instead, we can do:

```
minProduct(31, 35) = 2 * minProduct(15, 35) + 35
```

After all, since `31 = 2*15+1`, then `31x35 = 2*15*35+35`.

The logic in this final solution is that, on even numbers, we just divide `smaller` by 2 and double the result of the recursive call. On odd numbers, we do the same, but then we also add `bigger` to this result.

In doing so, we have an unexpected "win." Our `minProduct` function just recurses straight downwards, with increasingly small numbers each time. It will never repeat the same call, so there's no need to cache any information.

```
int minProduct(int a, int b) {
   int bigger = a < b ? b : a;
   int smaller = a < b ? a : b;
   return minProductHelper(smaller, bigger);
}

int minProductHelper(int smaller, int bigger) {
   if (smaller == 0) return 0;
   else if (smaller == 1) return bigger;

   int s = smaller >> 1; // Divide by 2
   int halfProd = minProductHelper(s, bigger);

   if (smaller % 2 == 0) {
      return halfProd + halfProd;
   } else {
      return halfProd + halfProd + bigger;
   }
}
```

This algorithm will run in `O(log s)` time, where `s` is the smaller of the two numbers.

8.6 **Towers of Hanoi:** 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 another tower.

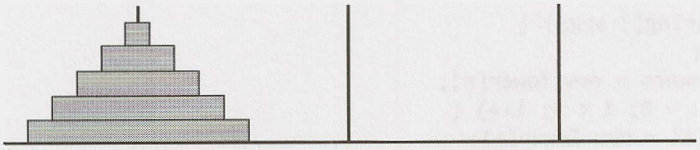
(3) A disk cannot be placed on top of a smaller disk.

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

pg 135

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—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:

```
moveDisks(int n, Tower origin, Tower destination, Tower buffer) {
  /* Base case */
  if (n <= 0) return;

  /* move top n - 1 disks from origin to buffer, using destination as a buffer. */
  moveDisks(n - 1, origin, buffer, destination);

  /* move top from origin to destination
  moveTop(origin, destination);

  /* move top n - 1 disks from buffer to destination, using origin as a buffer. */
  moveDisks(n - 1, buffer, destination, origin);
}
```

The following code provides a more detailed implementation of this algorithm, using concepts of object-oriented design.

```
void main(String[] args) {
  int n = 3;
  Tower[] towers = new Tower[n];
  for (int i = 0; i < 3; i++) {
    towers[i] = new Tower(i);
  }

  for (int i = n - 1; i >= 0; i--) {
    towers[0].add(i);
  }
  towers[0].moveDisks(n, towers[2], towers[1]);
}

class Tower {
  private Stack<Integer> disks;
  private int index;
  public Tower(int i) {
    disks = new Stack<Integer>();
    index = i;
  }

  public int index() {
    return index;
  }

  public void add(int d) {
    if (!disks.isEmpty() && disks.peek() <= d) {
      System.out.println("Error placing disk " + d);
    } else {
      disks.push(d);
    }
  }

  public void moveTopTo(Tower t) {
    int top = disks.pop();
    t.add(top);
  }

```

```
39     public void moveDisks(int n, Tower destination, Tower buffer) {
40       if (n > 0) {
41         moveDisks(n - 1, buffer, destination);
42         moveTopTo(destination);
43         buffer.moveDisks(n - 1, destination, this);
44       }
45     }
46   }
```

Implementing the towers as their own objects is not strictly necessary, but it does help to make the code cleaner in some respects.

8.7 Permutations without Dups: Write a method to compute all permutations of a string of unique characters.

pg 135

SOLUTION

Like in many recursive problems, the Base Case and Build approach will be useful. Assume we have a string S represented by the characters $a_1a_2...a_n$.

Approach 1: Building from permutations of first n-1 characters.

Base Case: permutations of first character substring

The only permutation of a_1 is the string a_1. So:

$P(a_1) = a_1$

Case: permutations of a_1a_2

$P(a_1a_2) = a_1a_2$ and a_2a_1

Case: permutations of $a_1a_2a_3$

$P(a_1a_2a_3) = a_1a_2a_3,\ a_1a_3a_2,\ a_2a_1a_3,\ a_2a_3a_1,\ a_3a_1a_2,\ a_3a_2a_1,$

Case: permutations of $a_1a_2a_3a_4$

This is the first interesting case. How can we generate permutations of $a_1a_2a_3a_4$ from $a_1a_2a_3$?

Each permutation of $a_1a_2a_3a_4$ represents an ordering of $a_1a_2a_3$. For example, $a_2a_4a_1a_3$ represents the order $a_2a_1a_3$.

Therefore, if we took all the permutations of $a_1a_2a_3$ and added a_4 into all possible locations, we would get all permutations of $a_1a_2a_3a_4$.

$a_1a_2a_3$ -> $a_4a_1a_2a_3,\ a_1a_4a_2a_3,\ a_1a_2a_4a_3,\ a_1a_2a_3a_4$
$a_1a_3a_2$ -> $a_4a_1a_3a_2,\ a_1a_4a_3a_2,\ a_1a_3a_4a_2,\ a_1a_3a_2a_4$
$a_3a_1a_2$ -> $a_4a_3a_1a_2,\ a_3a_4a_1a_2,\ a_3a_1a_4a_2,\ a_3a_1a_2a_4$
$a_2a_1a_3$ -> $a_4a_2a_1a_3,\ a_2a_4a_1a_3,\ a_2a_1a_4a_3,\ a_2a_1a_3a_4$
$a_2a_3a_1$ -> $a_4a_2a_3a_1,\ a_2a_4a_3a_1,\ a_2a_3a_4a_1,\ a_2a_3a_1a_4$
$a_3a_2a_1$ -> $a_4a_3a_2a_1,\ a_3a_4a_2a_1,\ a_3a_2a_4a_1,\ a_3a_2a_1a_4$

We can now implement this algorithm recursively.

```
1  ArrayList<String> getPerms(String str) {
2    if (str == null) return null;
3
4    ArrayList<String> permutations = new ArrayList<String>();
5    if (str.length() == 0) { // base case
6      permutations.add("");
```

```
      return permutations;
    }

    char first = str.charAt(0); // get the first char
    String remainder = str.substring(1); // remove the first char
    ArrayList<String> words = getPerms(remainder);
    for (String word : words) {
      for (int j = 0; j <= word.length(); j++) {
        String s = insertCharAt(word, first, j);
        permutations.add(s);
      }
    }
    return permutations;
  }

  /* Insert char c at index i in word. */
  String insertCharAt(String word, char c, int i) {
    String start = word.substring(0, i);
    String end = word.substring(i);
    return start + c + end;
  }
```

Approach 2: Building from permutations of all n-1 character substrings.

Base Case: single-character strings

The only permutation of a_1 is the string a_1. So:

$P(a_1) = a_1$

Case: two-character strings

$P(a_1a_2) = a_1a_2$ and a_2a_1.
$P(a_2a_3) = a_2a_3$ and a_3a_2.
$P(a_1a_3) = a_1a_3$ and a_3a_1.

Case: three-character strings

Here is where the cases get more interesting. How can we generate all permutations of three-character strings, such as $a_1a_2a_3$, given the permutations of two-character strings?

Well, in essence, we just need to "try" each character as the first character and then append the permutations.

$$P(a_1a_2a_3) = \{a_1 + P(a_2a_3)\} + a_2 + P(a_1a_3)\} + \{a_3 + P(a_1a_2)\}$$
$$\{a_1 + P(a_2a_3)\} \rightarrow a_1a_2a_3,\ a_1a_3a_2$$
$$\{a_2 + P(a_1a_3)\} \rightarrow a_2a_1a_3,\ a_2a_3a_1$$
$$\{a_3 + P(a_1a_2)\} \rightarrow a_3a_1a_2,\ a_3a_2a_1$$

Now that we can generate all permutations of three-character strings, we can use this to generate permutations of four-character strings.

$$P(a_1a_2a_3a_4) = \{a_1 + P(a_2a_3a_4)\} + \{a_2 + P(a_1a_3a_4)\} + \{a_3 + P(a_1a_2a_4)\} + \{a_4 + P(a_1a_2a_3)\}$$

This is now a fairly straightforward algorithm to implement.

```
1  ArrayList<String> getPerms(String remainder) {
2    int len = remainder.length();
3    ArrayList<String> result = new ArrayList<String>();
4
5    /* Base case. */
```

```
6     if (len == 0) {
7       result.add(""); // Be sure to return empty string!
8       return result;
9     }
10
11
12    for (int i = 0; i < len; i++) {
13      /* Remove char i and find permutations of remaining chars.*/
14      String before = remainder.substring(0, i);
15      String after = remainder.substring(i + 1, len);
16      ArrayList<String> partials = getPerms(before + after);
17
18      /* Prepend char i to each permutation.*/
19      for (String s : partials) {
20        result.add(remainder.charAt(i) + s);
21      }
22    }
23
24    return result;
25  }
```

Alternatively, instead of passing the permutations back up the stack, we can push the prefix down the stack. When we get to the bottom (base case), `prefix` holds a full permutation.

```
1   ArrayList<String> getPerms(String str) {
2     ArrayList<String> result = new ArrayList<String>();
3     getPerms("", str, result);
4     return result;
5   }
6
7   void getPerms(String prefix, String remainder, ArrayList<String> result) {
8     if (remainder.length() == 0) result.add(prefix);
9
10    int len = remainder.length();
11    for (int i = 0; i < len; i++) {
12      String before = remainder.substring(0, i);
13      String after = remainder.substring(i + 1, len);
14      char c = remainder.charAt(i);
15      getPerms(prefix + c, before + after, result);
16    }
17  }
```

For a discussion of the runtime of this algorithm, see Example 12 on page 51.

8.8 Permutations with Duplicates: Write a method to compute all permutations of a string whose characters are not necessarily unique. The list of permutations should not have duplicates.

pg 135

SOLUTION

This is very similar to the previous problem, except that now we could potentially have duplicate characters in the word.

One simple way of handling this problem is to do the same work to check if a permutation has been created before and then, if not, add it to the list. A simple hash table will do the trick here. This solution will take $O(n!)$ time in the worst case (and, in fact, in all cases).

While it's true that we can't beat this worst case time, we should be able to design an algorithm to beat this in many cases. Consider a string with all duplicate characters, like aaaaaaaaaaaaa. This will take an extremely long time (since there are over 6 billion permutations of a 13-character string), even though there is only one unique permutation.

Ideally, we would like to only create the unique permutations, rather than creating every permutation and then ruling out the duplicates.

We can start with computing the count of each letter (easy enough to get this—just use a hash table). For a string such as aabbbbc, this would be:

```
a->2 | b->4 | c->1
```

Let's imagine generating a permutation of this string (now represented as a hash table). The first choice we make is whether to use an a, b, or c as the first character. After that, we have a subproblem to solve: find all permutations of the remaining characters, and append those to the already picked "prefix."

```
P(a->2 | b->4 | c->1) = {a + P(a->1 | b->4 | c->1)} +
                        {b + P(a->2 | b->3 | c->1)} +
                        {c + P(a->2 | b->4 | c->0)}
    P(a->1 | b->4 | c->1) = {a + P(a->0 | b->4 | c->1)} +
                            {b + P(a->1 | b->3 | c->1)} +
                            {c + P(a->1 | b->4 | c->0)}
    P(a->2 | b->3 | c->1) = {a + P(a->1 | b->3 | c->1)} +
                            {b + P(a->2 | b->2 | c->1)} +
                            {c + P(a->2 | b->3 | c->0)}
    P(a->2 | b->4 | c->0) = {a + P(a->1 | b->4 | c->0)} +
                            {b + P(a->2 | b->3 | c->0)}
```

Eventually, we'll get down to no more characters remaining.

The code below implements this algorithm.

```
1  ArrayList<String> printPerms(String s) {
2    ArrayList<String> result = new ArrayList<String>();
3    HashMap<Character, Integer> map = buildFreqTable(s);
4    printPerms(map, "", s.length(), result);
5    return result;
6  }
7
8  HashMap<Character, Integer> buildFreqTable(String s) {
9    HashMap<Character, Integer> map = new HashMap<Character, Integer>();
10   for (char c : s.toCharArray()) {
11     if (!map.containsKey(c)) {
12       map.put(c, 0);
13     }
14     map.put(c, map.get(c) + 1);
15   }
16   return map;
17 }
18
19 void printPerms(HashMap<Character, Integer> map, String prefix, int remaining,
20                 ArrayList<String> result) {
21   /* Base case. Permutation has been completed. */
22   if (remaining == 0) {
23     result.add(prefix);
24     return;
25   }
26
27   /* Try remaining letters for next char, and generate remaining permutations. */
```

```
28    for (Character c : map.keySet()) {
29      int count = map.get(c);
30      if (count > 0) {
31        map.put(c,  count - 1);
32        printPerms(map, prefix + c, remaining - 1, result);
33        map.put(c,  count);
34      }
35    }
36  }
```

In situations where the string has many duplicates, this algorithm will run a lot faster than the earlier algorithm.

8.9 **Parens:** Implement an algorithm to print all valid (i.e., properly opened and closed) combinations of n pairs of parentheses.

EXAMPLE

Input: 3

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

pg 136

SOLUTION

Our first thought here might be to apply a recursive approach where we build the solution for `f(n)` by adding pairs of parentheses to `f(n-1)`. That's certainly a good instinct.

Let's consider the solution for `n = 3`:

```
(()())    ((()))    ()(())    (())()    ()()()
```

How might we build this from `n = 2`?

```
(())    ()()
```

We can do this by inserting a pair of parentheses inside every existing pair of parentheses, as well as one at the beginning of the string. Any other places that we could insert parentheses, such as at the end of the string, would reduce to the earlier cases.

So, we have the following:

```
(()) -> (()()) /* inserted pair after 1st left paren */
     -> ((())) / *inserted pair after 2nd left paren */
     -> ()(()) / *inserted pair at beginning of string */
()() -> (())() / *inserted pair after 1st left paren */
     -> ()(()) / *inserted pair after 2nd left paren */
     -> ()()() / *inserted pair at beginning of string */
```

But wait—we have some duplicate pairs listed. The string `()(())` is listed twice.

If we're going to apply this approach, we'll need to check for duplicate values before adding a string to our list.

```
1  Set<String> generateParens(int remaining) {
2    Set<String> set = new HashSet<String>();
3    if (remaining == 0) {
4      set.add("");
5    } else {
6      Set<String> prev = generateParens(remaining - 1);
7      for (String str : prev) {
8        for (int i = 0; i < str.length(); i++) {
```

```
            if (str.charAt(i) == '(') {
               String s = insertInside(str, i);
               /* Add s to set if it's not already in there. Note:  HashSet
                * automatically checks for duplicates before adding, so an explicit
                * check is not necessary. */
               set.add(s);
            }
         }
         set.add("()" + str);
      }
   }
   return set;
}

String insertInside(String str, int leftIndex) {
   String left = str.substring(0, leftIndex + 1);
   String right = str.substring(leftIndex + 1, str.length());
   return left + "()" + right;
}
```

This works, but it's not very efficient. We waste a lot of time coming up with the duplicate strings.

We can avoid this duplicate string issue by building the string from scratch. Under this approach, we add left and right parens, as long as our expression stays valid.

On each recursive call, we have the index for a particular character in the string. We need to select either a left or a right paren. When can we use a left paren, and when can we use a right paren?

1. *Left Paren:* As long as we haven't used up all the left parentheses, we can always insert a left paren.
2. *Right Paren:* We can insert a right paren as long as it won't lead to a syntax error. When will we get a syntax error? We will get a syntax error if there are more right parentheses than left.

So, we simply keep track of the number of left and right parentheses allowed. If there are left parens remaining, we'll insert a left paren and recurse. If there are more right parens remaining than left (i.e., if there are more left parens in use than right parens), then we'll insert a right paren and recurse.

```
void addParen(ArrayList<String> list, int leftRem, int rightRem, char[] str,
              int index) {
   if (leftRem < 0 || rightRem < leftRem) return; // invalid state

   if (leftRem == 0 && rightRem == 0) { /* Out of left and right parentheses */
      list.add(String.copyValueOf(str));
   } else {
      str[index] = '('; // Add left and recurse
      addParen(list, leftRem - 1, rightRem, str, index + 1);

      str[index] = ')'; // Add right and recurse
      addParen(list, leftRem, rightRem - 1, str, index + 1);
   }
}

ArrayList<String> generateParens(int count) {
   char[] str = new char[count*2];
   ArrayList<String> list = new ArrayList<String>();
   addParen(list, count, count, str, 0);
   return list;
}
```

Because we insert left and right parentheses at each index in the string, and we never repeat an index, each string is guaranteed to be unique.

8.10 Paint Fill: 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 136

SOLUTION

First, let's visualize how this method works. When we call `paintFill` (i.e., "click" paint fill in the image editing application) on, say, a green pixel, we want to "bleed" outwards. Pixel by pixel, we expand outwards by calling `paintFill` on the surrounding pixel. When we hit a pixel that is not green, we stop.

We can implement this algorithm recursively:

```
1  enum Color { Black, White, Red, Yellow, Green }
2
3  boolean PaintFill(Color[][] screen, int r, int c, Color ncolor) {
4     if (screen[r][c] == ncolor) return false;
5     return PaintFill(screen, r, c, screen[r][c], ncolor);
6  }
7
8  boolean PaintFill(Color[][] screen, int r, int c, Color ocolor, Color ncolor) {
9     if (r < 0 || r >= screen.length || c < 0 || c >= screen[0].length) {
10       return false;
11    }
12
13    if (screen[r][c] == ocolor) {
14       screen[r][c] = ncolor;
15       PaintFill(screen, r - 1, c, ocolor, ncolor); // up
16       PaintFill(screen, r + 1, c, ocolor, ncolor); // down
17       PaintFill(screen, r, c - 1, ocolor, ncolor); // left
18       PaintFill(screen, r, c + 1, ocolor, ncolor); // right
19    }
20    return true;
21 }
```

If you used the variable names `x` and `y` to implement this, be careful about the ordering of the variables in `screen[y][x]`. Because `x` represents the *horizontal* axis (that is, it's left to right), it actually corresponds to the column number, not the row number. The value of `y` equals the number of rows. This is a very easy place to make a mistake in an interview, as well as in your daily coding. It's typically clearer to use `row` and `column` instead, as we've done here.

Does this algorithm seem familiar? It should! This is essentially depth-first search on a graph. At each pixel, we are searching outwards to each surrounding pixel. We stop once we've fully traversed all the surrounding pixels of this color.

We could alternatively implement this using breadth-first search.

8.11 Coins: 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 136

SOLUTION

This is a recursive problem, so let's figure out how to compute `makeChange(n)` using prior solutions (i.e., subproblems).

Let's say `n = 100`. We want to compute the number of ways of making change for 100 cents. What is the relationship between this problem and its subproblems?

We know that making change for 100 cents will involve either 0, 1, 2, 3, or 4 quarters. So:

```
makeChange(100) = makeChange(100 using 0 quarters) +
                  makeChange(100 using 1 quarter)  +
                  makeChange(100 using 2 quarters) +
                  makeChange(100 using 3 quarters) +
                  makeChange(100 using 4 quarters)
```

Inspecting this further, we can see that some of these problems reduce. For example, `makeChange(100 using 1 quarter)` will equal `makeChange(75 using 0 quarters)`. This is because, if we must use exactly one quarter to make change for 100 cents, then our only remaining choices involve making change for the remaining 75 cents.

We can apply the same logic to `makeChange(100 using 2 quarters)`, `makeChange(100 using 3 quarters)` and `makeChange(100 using 4 quarters)`. We have thus reduced the above statement to the following.

```
makeChange(100) = makeChange(100 using 0 quarters) +
                  makeChange(75 using 0 quarters) +
                  makeChange(50 using 0 quarters) +
                  makeChange(25 using 0 quarters) +
                  1
```

Note that the final statement from above, `makeChange(100 using 4 quarters)`, equals 1. We call this "fully reduced."

Now what? We've used up all our quarters, so now we can start applying our next biggest denomination: dimes.

Our approach for quarters applies to dimes as well, but we apply this for *each* of the four of five parts of the above statement. So, for the first part, we get the following statements:

```
makeChange(100 using 0 quarters) = makeChange(100 using 0 quarters, 0 dimes) +
                                   makeChange(100 using 0 quarters, 1 dime)  +
                                   makeChange(100 using 0 quarters, 2 dimes) +
                                   ...
                                   makeChange(100 using 0 quarters, 10 dimes)

makeChange(75 using 0 quarters) = makeChange(75 using 0 quarters, 0 dimes) +
                                  makeChange(75 using 0 quarters, 1 dime)  +
                                  makeChange(75 using 0 quarters, 2 dimes) +
                                  ...
                                  makeChange(75 using 0 quarters, 7 dimes)

makeChange(50 using 0 quarters) = makeChange(50 using 0 quarters, 0 dimes) +
                                  makeChange(50 using 0 quarters, 1 dime)  +
                                  makeChange(50 using 0 quarters, 2 dimes) +
```

```
                                    ...
                                    makeChange(50 using 0 quarters, 5 dimes)

makeChange(25 using 0 quarters) = makeChange(25 using 0 quarters, 0 dimes) +
                                  makeChange(25 using 0 quarters, 1 dime)  +
                                  makeChange(25 using 0 quarters, 2 dimes)
```

Each one of these, in turn, expands out once we start applying nickels. We end up with a tree-like recursive structure where each call expands out to four or more calls.

The base case of our recursion is the fully reduced statement. For example, `makeChange(50 using 0 quarters, 5 dimes)` is fully reduced to 1, since 5 dimes equals 50 cents.

This leads to a recursive algorithm that looks like this:

```
1  int makeChange(int amount, int[] denoms, int index) {
2    if (index >= denoms.length - 1) return 1; // last denom
3    int denomAmount = denoms[index];
4    int ways = 0;
5    for (int i = 0; i * denomAmount <= amount; i++) {
6      int amountRemaining = amount - i * denomAmount;
7      ways += makeChange(amountRemaining, denoms, index + 1);
8    }
9    return ways;
10 }
11
12 int makeChange(int n) {
13   int[] denoms = {25, 10, 5, 1};
14   return makeChange(n, denoms, 0);
15 }
```

This works, but it's not as optimal as it could be. The issue is that we will be recursively calling `makeChange` several times for the same values of `amount` and `index`.

We can resolve this issue by storing the previously computed values. We'll need to store a mapping from each pair `(amount, index)` to the precomputed result.

```
1  int makeChange(int n) {
2    int[] denoms = {2  510, 5, 1};
3    int[][] map = new int[n + 1][denoms.length]; // precomputed vals
4    return makeChange(n, denoms, 0, map);
5  }
6
7  int makeChange(int amount, int[] denoms, int index, int[][] map) {
8    if (map[amount][index] > 0) { // retrieve value
9      return map[amount][index];
10   }
11   if (index >= denoms.length - 1) return 1; // one denom remaining
12   int denomAmount = denoms[index];
13   int ways = 0;
14   for (int i = 0; i * denomAmount <= amount; i++) {
15     // go to next denom, assuming i coins of denomAmount
16     int amountRemaining = amount - i * denomAmount;
17     ways += makeChange(amountRemaining, denoms, index + 1, map);
18   }
19   map[amount][index] = ways;
20   return ways;
21 }
```

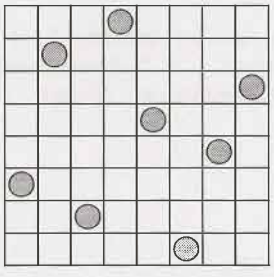
Note that we've used a two-dimensional array of integers to store the previously computed values. This is simpler, but takes up a little extra space. Alternatively, we could use an actual hash table that maps from `amount` *to a new hash table, which then maps from* `denom` *to the precomputed value. There are other alternative data structures as well.*

8.12 **Eight Queens:** 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 136

SOLUTION

We have eight queens which must be lined up on an 8x8 chess board such that none share the same row, column or diagonal. So, we know that each row and column (and diagonal) must be used exactly once.



A "Solved" Board with 8 Queens

Picture the queen that is placed last, which we'll assume is on row 8. (This is an okay assumption to make since the ordering of placing the queens is irrelevant.) On which cell in row 8 is this queen? There are eight possibilities, one for each column.

So if we want to know all the valid ways of arranging 8 queens on an 8x8 chess board, it would be:

```
ways to arrange 8 queens on an 8x8 board =
   ways to arrange 8 queens on an 8x8 board with queen at (7, 0) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 1) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 2) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 3) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 4) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 5) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 6) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 7)
```

We can compute each one of these using a very similar approach:

```
ways to arrange 8 queens on an 8x8 board with queen at (7, 3) =
   ways to ... with queens at (7, 3) and (6, 0) +
   ways to ... with queens at (7, 3) and (6, 1) +
   ways to ... with queens at (7, 3) and (6, 2) +
   ways to ... with queens at (7, 3) and (6, 4) +
   ways to ... with queens at (7, 3) and (6, 5) +
   ways to ... with queens at (7, 3) and (6, 6) +
   ways to ... with queens at (7, 3) and (6, 7)
```

Note that we don't need to consider combinations with queens at `(7, 3)` and `(6, 3)`, since this is a violation of the requirement that every queen is in its own row, column and diagonal.

Implementing this is now reasonably straightforward.

```
int GRID_SIZE = 8;

void placeQueens(int row, Integer[] columns, ArrayList<Integer[]> results) {
   if (row == GRID_SIZE) { // Found valid placement
      results.add(columns.clone());
   } else {
      for (int col = 0; col < GRID_SIZE; col++) {
         if (checkValid(columns, row, col)) {
            columns[row] = col;  // Place queen
            placeQueens(row + 1, columns, results);
         }
      }
   }
}

/* Check if (row1, column1) is a valid spot for a queen by checking if there is a
 * queen in the same column or diagonal. We don't need to check it for queens in
 * the same row because the calling placeQueen only attempts to place one queen at
 * a time. We know this row is empty. */
boolean checkValid(Integer[] columns, int row1, int column1) {
   for (int row2 = 0; row2 < row1; row2++) {
      int column2 = columns[row2];
      /* Check if (row2, column2) invalidates (row1, column1) as a
       * queen spot. */

      /* Check if rows have a queen in the same column */
      if (column1 == column2) {
         return false;
      }

      /* Check diagonals: if the distance between the columns equals the distance
       * between the rows, then they're in the same diagonal. */
      int columnDistance = Math.abs(column2 - column1);

      /* row1 > row2, so no need for abs */
      int rowDistance = row1 - row2;
      if (columnDistance == rowDistance) {
         return false;
      }
   }
   return true;
}
```

Observe that since each row can only have one queen, we don't need to store our board as a full 8x8 matrix. We only need a single array where `column[r] = c` indicates that row `r` has a queen at column `c`.

8.13 Stack of Boxes: 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 compute the height of the tallest possible stack. The height of a stack is the sum of the heights of each box.

pg 136

SOLUTION

To tackle this problem, we need to recognize the relationship between the different subproblems.

Solution #1

Imagine we had the following boxes: $b_1, b_2, \ldots, b_n$. The biggest stack that we can build with all the boxes equals the max of (biggest stack with bottom b_1, biggest stack with bottom b_2, ..., biggest stack with bottom b_n). That is, if we experimented with each box as a bottom and built the biggest stack possible with each, we would find the biggest stack possible.

But, how would we find the biggest stack with a particular bottom? Essentially the same way. We experiment with different boxes for the second level, and so on for each level.

Of course, we only experiment with valid boxes. If b_5 is bigger than b_1, then there's no point in trying to build a stack that looks like $\{b_1, b_5, \ldots\}$. We already know b_1 can't be below b_5.

We can perform a small optimization here. The requirements of this problem stipulate that the lower boxes must be strictly greater than the higher boxes in all dimensions. Therefore, if we sort (descending order) the boxes on a dimension—any dimension—then we know we don't have to look backwards in the list. The box b_1 cannot be on top of box b_5, since its height (or whatever dimension we sorted on) is greater than b_5's height.

The code below implements this algorithm recursively.

```
1  int createStack(ArrayList<Box> boxes) {
2    /* Sort in decending order by height. */
3    Collections.sort(boxes, new BoxComparator());
4    int maxHeight = 0;
5    for (int i = 0; i < boxes.size(); i++) {
6      int height = createStack(boxes, i);
7      maxHeight = Math.max(maxHeight, height);
8    }
9    return maxHeight;
10 }
11
12 int createStack(ArrayList<Box> boxes, int bottomIndex) {
13   Box bottom = boxes.get(bottomIndex);
14   int maxHeight = 0;
15   for (int i = bottomIndex + 1; i < boxes.size(); i++) {
16     if (boxes.get(i).canBeAbove(bottom)) {
17       int height = createStack(boxes, i);
18       maxHeight = Math.max(height, maxHeight);
19     }
20   }
21   maxHeight += bottom.height;
22   return maxHeight;
23 }
24
25 class BoxComparator implements Comparator<Box> {
```

```
26    @Override
27    public int compare(Box x, Box y){
28       return y.height - x.height;
29    }
30 }
```

The problem in this code is that it gets very inefficient. We try to find the best solution that looks like $\{b_3, b_4, \ldots\}$ even though we may have already found the best solution with b_4 at the bottom. Instead of generating these solutions from scratch, we can cache these results using memoization.

```
1  int createStack(ArrayList<Box> boxes) {
2     Collections.sort(boxes, new BoxComparator());
3     int maxHeight = 0;
4     int[] stackMap = new int[boxes.size()];
5     for (int i = 0; i < boxes.size(); i++) {
6        int height = createStack(boxes, i, stackMap);
7        maxHeight = Math.max(maxHeight, height);
8     }
9     return maxHeight;
10 }
11
12 int createStack(ArrayList<Box> boxes, int bottomIndex, int[] stackMap) {
13    if (bottomIndex < boxes.size() && stackMap[bottomIndex] > 0) {
14       return stackMap[bottomIndex];
15    }
16
17    Box bottom = boxes.get(bottomIndex);
18    int maxHeight = 0;
19    for (int i = bottomIndex + 1; i < boxes.size(); i++) {
20       if (boxes.get(i).canBeAbove(bottom)) {
21          int height = createStack(boxes, i, stackMap);
22          maxHeight = Math.max(height, maxHeight);
23       }
24    }
25    maxHeight += bottom.height;
26    stackMap[bottomIndex] = maxHeight;
27    return maxHeight;
28 }
```

Because we're only mapping from an index to a height, we can just use an integer array for our "hash table."

Be very careful here with what each spot in the hash table represents. In this code, `stackMap[i]` represents the tallest stack with box `i` at the bottom. Before pulling the value from the hash table, you have to ensure that box `i` can be placed on top of the current bottom.

It helps to keep the line that recalls from the hash table symmetric with the one that inserts. For example, in this code, we recall from the hash table with `bottomIndex` at the start of the method. We insert into the hash table with `bottomIndex` at the end.

Solution #2

Alternatively, we can think about the recursive algorithm as making a choice, at each step, whether to put a particular box in the stack. (We will again sort our boxes in descending order by a dimension, such as height.)

First, we choose whether or not to put box 0 in the stack. Take one recursive path with box 0 at the bottom and one recursive path without box 0. Return the better of the two options.

Then, we choose whether or not to put box 1 in the stack. Take one recursive path with box 1 at the bottom and one path without box 1. Return the better of the two options.

We will again use memoization to cache the height of the tallest stack with a particular bottom.

```
1  int createStack(ArrayList<Box> boxes) {
2    Collections.sort(boxes, new BoxComparator());
3    int[] stackMap = new int[boxes.size()];
4    return createStack(boxes, null, 0, stackMap);
5  }
6
7  int createStack(ArrayList<Box> boxes, Box bottom, int offset, int[] stackMap) {
8    if (offset >= boxes.size()) return 0; // Base case
9
10   / *height with this bottom */
11   Box newBottom = boxes.get(offset);
12   int heightWithBottom = 0;
13   if (bottom == null || newBottom.canBeAbove(bottom)) {
14     if (stackMap[offset] == 0) {
15       stackMap[offset] = createStack(boxes, newBottom, offset + 1, stackMap);
16       stackMap[offset] += newBottom.height;
17     }
18     heightWithBottom = stackMap[offset];
19   }
20
21   / *without this bottom */
22   int heightWithoutBottom = createStack(boxes, bottom, offset + 1, stackMap);
23
24   /* Return better of two options. */
25   return Math.max(heightWithBottom, heightWithoutBottom);
26 }
```

Again, pay close attention to when you recall and insert values into the hash table. It's typically best if these are symmetric, as they are in lines 15 and 16-18.

8.14 Boolean Evaluation: Given a boolean expression consisting of the symbols `0` (false), `1` (true), `&` (AND), `|` (OR), and `^` (XOR), 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`. The expression should be fully parenthesized (e.g., `(0)^(1)`) but not extraneously (e.g., `(((0))^(1))`).

EXAMPLE

```
countEval("1^0|0|1", false) -> 2
countEval("0&0&0&1^1|0", true) -> 10
```

pg 136

SOLUTION

As in other recursive problems, the key to this problem is to figure out the relationship between a problem and its subproblems.

Brute Force

Consider an expression like `0^0&0^1|1` and the target result `true`. How can we break down `countEval(0^0&0^1|1, true)` into smaller problems?

We could just essentially iterate through each possible place to put a parenthesis.

```
countEval(0^0&0^1|1, true) =
   countEval(0^0&0^1|1 where paren around char 1, true)
 + countEval(0^0&0^1|1 where paren around char 3, true)
 + countEval(0^0&0^1|1 where paren around char 5, true)
 + countEval(0^0&0^1|1 where paren around char 7, true)
```

Now what? Let's look at just one of those expressions—the paren around char 3. This gives us `(0^0)&(0^1)`.

In order to make that expression true, both the left and right sides must be true. So:

```
left = "0^0"
right = "0^1|1"
countEval(left & right, true) = countEval(left, true) * countEval(right, true)
```

The reason we multiply the results of the left and right sides is that each result from the two sides can be paired up with each other to form a unique combination.

Each of those terms can now be decomposed into smaller problems in a similar process.

What happens when we have an "|" (OR)? Or an "^" (XOR)?

If it's an OR, then either the left or the right side must be true—or both.

```
countEval(left | right, true) = countEval(left, true)  * countEval(right, false)
                              + countEval(left, false) * countEval(right, true)
                              + countEval(left, true)  * countEval(right, true)
```

If it's an XOR, then the left or the right side can be true, but not both.

```
countEval(left ^ right, true) = countEval(left, true) * countEval(right, false)
                              + countEval(left, false) * countEval(right, true)
```

What if we were trying to make the result `false` instead? We can switch up the logic from above:

```
countEval(left & right, false) = countEval(left, true)  * countEval(right, false)
                               + countEval(left, false) * countEval(right, true)
                               + countEval(left, false) * countEval(right, false)
countEval(left | right, false) = countEval(left, false) * countEval(right, false)
countEval(left ^ right, false) = countEval(left, false) * countEval(right, false)
                               + countEval(left, true)  * countEval(right, true)
```

Alternatively, we can just use the same logic from above and subtract it out from the total number of ways of evaluating the expression.

```
totalEval(left) = countEval(left, true) + countEval(left, false)
totalEval(right) = countEval(right, true) + countEval(right, false)
totalEval(expression) = totalEval(left) * totalEval(right)
countEval(expression, false) = totalEval(expression) - countEval(expression, true)
```

This makes the code a bit more concise.

```
1  int countEval(String s, boolean result) {
2    if (s.length() == 0) return 0;
3    if (s.length() == 1) return stringToBool(s) == result ? 1 : 0;
4
5    int ways = 0;
6    for (int i = 1; i < s.length(); i += 2) {
7      char c = s.charAt(i);
8      String left = s.substring(0, i);
9      String right = s.substring(i + 1, s.length());
10
11     /* Evaluate each side for each result. */
12     int leftTrue = countEval(left, true);
13     int leftFalse = countEval(left, false);
14     int rightTrue = countEval(right, true);
```

```
15      int rightFalse = countEval(right, false);
16      int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);
17
18      int totalTrue = 0;
19      if (c == '^') { // required: one true and one false
20        totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
21      } else if (c == '&') { // required: both true
22        totalTrue = leftTrue * rightTrue;
23      } else if (c == '|') { // required: anything but both false
24        totalTrue = leftTrue * rightTrue + leftFalse * rightTrue +
25                    leftTrue * rightFalse;
26      }
27
28      int subWays = result ? totalTrue : total - totalTrue;
29      ways += subWays;
30    }
31
32    return ways;
33  }
34
35  boolean stringToBool(String c) {
36    return c.equals("1") ? true : false;
37  }
```

Note that the tradeoff of computing the `false` results from the `true` ones, and of computing the `{leftTrue, rightTrue, leftFalse, and rightFalse}` values upfront, is a small amount of extra work in some cases. For example, if we're looking for the ways that an AND (&) can result in `true`, we never would have needed the `leftFalse` and `rightFalse` results. Likewise, if we're looking for the ways that an OR (|) can result in `false`, we never would have needed the `leftTrue` and `rightTrue` results.

Our current code is blind to what we do and don't actually need to do and instead just computes all of the values. This is probably a reasonable tradeoff to make (especially given the constraints of whiteboard coding) as it makes our code substantially shorter and less tedious to write. Whichever approach you make, you should discuss the tradeoffs with your interviewer.

That said, there are more important optimizations we can make.

Optimized Solutions

If we follow the recursive path, we'll note that we end up doing the same computation repeatedly.

Consider the expression `0^0&0^1|1` and these recursion paths:

- Add parens around char 1. `(0)^(0&0^1|1)`
  - » Add parens around char 3. `(0)^((0)&(0^1|1))`
- Add parens around char 3. `(0^0)&(0^1|1)`
  - » Add parens around char 1. `((0)^(0))&(0^1|1)`

Although these two expressions are different, they have a similar component: `(0^1|1)`. We should reuse our effort on this.

We can do this by using memoization, or a hash table. We just need to store the result of `countEval(expression, result)` for each expression and result. If we see an expression that we've calculated before, we just return it from the cache.

```
1  int countEval(String s, boolean result, HashMap<String, Integer> memo) {
2    if (s.length() == 0) return 0;
3    if (s.length() == 1) return stringToBool(s) == result ? 1 : 0;
```

```
4     if (memo.containsKey(result + s)) return memo.get(result + s);
5
6     int ways = 0;
7
8     for (int i = 1; i < s.length(); i += 2) {
9        char c = s.charAt(i);
10       String left = s.substring(0, i);
11       String right = s.substring(i + 1, s.length());
12       int leftTrue = countEval(left, true, memo);
13       int leftFalse = countEval(left, false, memo);
14       int rightTrue = countEval(right, true, memo);
15       int rightFalse = countEval(right, false, memo);
16       int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);
17
18       int totalTrue = 0;
19       if (c == '^') {
20          totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
21       } else if (c == '&') {
22          totalTrue = leftTrue * rightTrue;
23       } else if (c == '|') {
24          totalTrue = leftTrue * rightTrue + leftFalse * rightTrue +
25                      leftTrue * rightFalse;
26       }
27
28       int subWays = result ? totalTrue : total - totalTrue;
29       ways += subWays;
30    }
31
32    memo.put(result + s, ways);
33    return ways;
34 }
```

The added benefit of this is that we could actually end up with the same substring in multiple parts of the expression. For example, an expression like `0^1^0&0^1^0` has two instances of `0^1^0`. By caching the result of the substring value in a memoization table, we'll get to reuse the result for the right part of the expression after computing it for the left.

There is one further optimization we can make, but it's far beyond the scope of the interview. There *is* a closed form expression for the number of ways of parenthesizing an expression, but you wouldn't be expected to know it. It is given by the Catalan numbers, where n is the number of operators:

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

We could use this to compute the total ways of evaluating the expression. Then, rather than computing `leftTrue` and `leftFalse`, we just compute one of those and calculate the other using the Catalan numbers. We would do the same thing for the right side.

9

Solutions to System Design and Scalability

9.1 **Stock Data:** Imagine you are building some sort of service that will be called by up to 1,000 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 that 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 144

SOLUTION

From the statement of the problem, we want to focus on how we actually distribute the information to clients. We can assume that we have some scripts that magically collect the information.

We want to start off by thinking about what the different aspects we should consider in a given proposal are:

- *Client Ease of Use:* We want the service to be easy for the clients to implement and useful for them.
- *Ease for Ourselves:* This service should be as easy as possible for us to implement, as we shouldn't impose unnecessary work on ourselves. We need to consider in this not only the cost of implementing, but also the cost of maintenance.
- *Flexibility for Future Demands:* This problem is stated in a "what would you do in the real world" way, so we should think like we would in a real-world problem. Ideally, we do not want to overly constrain ourselves in the implementation, such that we can't be flexible if the requirements or demands change.
- *Scalability and Efficiency:* We should be mindful of the efficiency of our solution, so as not to overly burden our service.

With this framework in mind, we can consider various proposals.

Proposal #1

One option is that we could keep the data in simple text files and let clients download the data through some sort of FTP server. This would be easy to maintain in some sense, since files can be easily viewed and backed up, but it would require more complex parsing to do any sort of query. And, if additional data were added to our text file, it might break the clients' parsing mechanism.

Proposal #2

We could use a standard SQL database, and let the clients plug directly into that. This would provide the following benefits:

- Facilitates an easy way for the clients to do query processing over the data, in case there are additional features we need to support. For example, we could easily and efficiently perform a query such as "return all stocks having an open price greater than N and a closing price less than M."
- Rolling back, backing up data, and security could be provided using standard database features. We don't have to "reinvent the wheel," so it's easy for us to implement.
- Reasonably easy for the clients to integrate into existing applications. SQL integration is a standard feature in software development environments.

What are the disadvantages of using a SQL database?

- It's much heavier weight than we really need. We don't necessarily need all the complexity of a SQL backend to support a feed of a few bits of information.
- It's difficult for humans to be able to read it, so we'll likely need to implement an additional layer to view and maintain the data. This increases our implementation costs.
- Security: While a SQL database offers pretty well defined security levels, we would still need to be very careful to not give clients access that they shouldn't have. Additionally, even if clients aren't doing anything "malicious," they might perform expensive and inefficient queries, and our servers would bear the costs of that.

These disadvantages don't mean that we shouldn't provide SQL access. Rather, they mean that we should be aware of the disadvantages.

Proposal #3

XML is another great option for distributing the information. Our data has fixed format and fixed size: `company_name`, `open`, `high`, `low`, `closing price`. The XML could look like this:

```
<root>
  <date value="2008-10-12">
    <company name="foo">
      <open>126.23</open>
      <high>130.27</high>
      <low>122.83</low>
      <closingPrice>127.30</closingPrice>
    </company>
    <company name="bar">
      <open>52.73</open>
      <high>60.27</high>
      <low>50.29</low>
      <closingPrice>54.91</closingPrice>
    </company>
  </date>
  <date value="2008-10-11"> . . . </date>
</root>
```

The advantages of this approach include the following:

- It's very easy to distribute, and it can also be easily read by both machines and humans. This is one reason that XML is a standard data model to share and distribute data.
- Most languages have a library to perform XML parsing, so it's reasonably easy for clients to implement.

- We can add new data to the XML file by adding additional nodes. This would not break the client's parser (provided they have implemented their parser in a reasonable way).
- Since the data is being stored as XML files, we can use existing tools for backing up the data. We don't need to implement our own backup tool.

The disadvantages may include:

- This solution sends the clients all the information, even if they only want part of it. It is inefficient in that way.
- Performing any queries on the data requires parsing the entire file.

Regardless of which solution we use for data storage, we could provide a web service (e.g., SOAP) for client data access. This adds a layer to our work, but it can provide additional security, and it may even make it easier for clients to integrate the system.

However—and this is a pro and a con—clients will be limited to grabbing the data only how we expect or want them to. By contrast, in a pure SQL implementation, clients could query for the highest stock price, even if this wasn't a procedure we "expected" them to need.

So which one of these would we use? There's no clear answer. The pure text file solution is probably a bad choice, but you can make a compelling argument for the SQL or XML solution, with or without a web service.

The goal of a question like this is not to see if you get the "correct" answer (there is no single correct answer). Rather, it's to see how you design a system, and how you evaluate trade-offs.

9.2 **Social Network:** 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 shortest path between two people (e.g., Me -> Bob -> Susan -> Jason -> You).

pg 145

SOLUTION

A good way to approach this problem is to remove some of the constraints and solve it for that situation first.

Step 1: Simplify the Problem—Forget About the Millions of Users

First, let's forget that we're dealing with millions of users. Design this for the simple case.

We can construct a graph by treating each person as a node and letting an edge between two nodes indicate that the two users are friends.

If I wanted to find the path between two people, I could start with one person and do a simple breadth-first search.

Why wouldn't a depth-first search work well? First, depth-first search would just find a path. It wouldn't necessarily find the shortest path. Second, even if we just needed any path, it would be very inefficient. Two users might be only one degree of separation apart, but I could search millions of nodes in their "subtrees" before finding this relatively immediate connection.

Alternatively, I could do what's called a bidirectional breadth-first search. This means doing two breadth-first searches, one from the source and one from the destination. When the searches collide, we know we've found a path.

In the implementation, we'll use two classes to help us. `BFSData` holds the data we need for a breadth-first search, such as the `isVisited` hash table and the `toVisit` queue. `PathNode` will represent the path as we're searching it, storing each `Person` and the `previousNode` we visited in this path.

```
1  LinkedList<Person> findPathBiBFS(HashMap<Integer, Person> people, int source,
2                                   int destination) {
3     BFSData sourceData = new BFSData(people.get(source));
4     BFSData destData = new BFSData(people.get(destination));
5
6     while (!sourceData.isFinished() && !destData.isFinished()) {
7        / *Search out from source. */
8        Person collision = searchLevel(people, sourceData, destData);
9        if (collision != null) {
10          return mergePaths(sourceData, destData, collision.getID());
11       }
12
13       / *Search out from destination. */
14       collision = searchLevel(people, destData, sourceData);
15       if (collision != null) {
16          return mergePaths(sourceData, destData, collision.getID());
17       }
18    }
19    return null;
20 }
21
22 / *Search one level and return collision, if any. */
23 Person searchLevel(HashMap<Integer, Person> people, BFSData primary,
24                    BFSData secondary) {
25    / *We only want to search one level at a time. Count how many nodes are
26     * currently in the primary's level and only do that many nodes. We'll continue
27     * to add nodes to the end. */
28    int count = primary.toVisit.size();
29    for (int i = 0; i < count; i++) {
30       / *Pull out first node. */
31       PathNode pathNode = primary.toVisit.poll();
32       int personId = pathNode.getPerson().getID();
33
34       / *Check if it's already been visited. */
35       if (secondary.visited.containsKey(personId)) {
36          return pathNode.getPerson();
37       }
38
39       / *Add friends to queue. */
40       Person person = pathNode.getPerson();
41       ArrayList<Integer> friends = person.getFriends();
42       for (int friendId : friends) {
43          if (!primary.visited.containsKey(friendId)) {
44             Person friend = people.get(friendId);
45             PathNode next = new PathNode(friend, pathNode);
46             primary.visited.put(friendId, next);
47             primary.toVisit.add(next);
48          }
49       }
50    }
51    return null;
52 }
53
```

```
54 /* Merge paths where searches met at connection. */
55 LinkedList<Person> mergePaths(BFSData bfs1, BFSData bfs2, int connection) {
56    PathNode end1 = bfs1.visited.get(connection); // end1 -> source
57    PathNode end2 = bfs2.visited.get(connection); // end2 -> dest
58    LinkedList<Person> pathOne = end1.collapse(false);
59    LinkedList<Person> pathTwo = end2.collapse(true); // reverse
60    pathTwo.removeFirst(); // remove connection
61    pathOne.addAll(pathTwo); // add second path
62    return pathOne;
63 }
64
65 class PathNode {
66    private Person person = null;
67    private PathNode previousNode = null;
68    public PathNode(Person p, PathNode previous) {
69       person = p;
70       previousNode = previous;
71    }
72
73    public Person getPerson() { return person; }
74
75    public LinkedList<Person> collapse(boolean startsWithRoot) {
76       LinkedList<Person> path = new LinkedList<Person>();
77       PathNode node = this;
78       while (node != null) {
79          if (startsWithRoot) {
80             path.addLast(node.person);
81          } else {
82             path.addFirst(node.person);
83          }
84          node = node.previousNode;
85       }
86       return path;
87    }
88 }
89
90 class BFSData {
91    public Queue<PathNode> toVisit = new LinkedList<PathNode>();
92    public HashMap<Integer, PathNode> visited =
93       new HashMap<Integer, PathNode>();
94
95    public BFSData(Person root) {
96       PathNode sourcePath = new PathNode(root, null);
97       toVisit.add(sourcePath);
98       visited.put(root.getID(), sourcePath);
99    }
100
101   public boolean isFinished() {
102      return toVisit.isEmpty();
103   }
104 }
```

Many people are surprised that this is faster. Some quick math can explain why.

Suppose every person has k friends, and node S and node D have a friend C in common.

- Traditional breadth-first search from S to D: We go through roughly k+k*k nodes: each of S's k friends, and then each of their k friends.

- Bidirectional breadth-first search: We go through 2k nodes: each of S's k friends and each of D's k friends.

Of course, 2k is much less than k+k*k.

Generalizing this to a path of length q, we have this:

- BFS: $O(k^q)$
- Bidirectional BFS: $O(k^{q/2} + k^{q/2})$, which is just $O(k^{q/2})$

If you imagine a path like A->B->C->D->E where each person has 100 friends, this is a big difference. BFS will require looking at 100 million (100^4) nodes. A bidirectional BFS will require looking at only 20,000 nodes (2×100^2).

A bidirectional BFS will generally be faster than the traditional BFS. However, it requires actually having access to both the source node and the destination nodes, which is not always the case.

Step 2: Handle the Millions of Users

When we deal with a service the size of LinkedIn or Facebook, we cannot possibly keep all of our data on one machine. That means that our simple `Person` data structure from above doesn't quite work—our friends may not live on the same machine as we do. Instead, we can replace our list of friends with a list of their IDs, and traverse as follows:

1. For each friend ID: `int machine_index = getMachineIDForUser(personID);`
2. Go to machine `#machine_index`
3. On that machine, do: `Person friend = getPersonWithID(person_id);`

The code below outlines this process. We've defined a class `Server`, which holds a list of all the machines, and a class `Machine`, which represents a single machine. Both classes have hash tables to efficiently lookup data.

```
1  class Server {
2     HashMap<Integer, Machine> machines = new HashMap<Integer, Machine>();
3     HashMap<Integer, Integer> personToMachineMap = new HashMap<Integer, Integer>();
4
5     public Machine getMachineWithId(int machineID) {
6        return machines.get(machineID);
7     }
8
9     public int getMachineIDForUser(int personID) {
10       Integer machineID = personToMachineMap.get(personID);
11       return machineID == null ? -1 : machineID;
12    }
13
14    public Person getPersonWithID(int personID) {
15       Integer machineID = personToMachineMap.get(personID);
16       if (machineID == null) return null;
17
18       Machine machine = getMachineWithId(machineID);
19       if (machine == null) return null;
20
21       return machine.getPersonWithID(personID);
22    }
23 }
24
25 class Person {
```

```
26    private ArrayList<Integer> friends = new ArrayList<Integer>();
27    private int personID;
28    private String info;
29
30    public Person(int id) { this.personID = id; }
31    public String getInfo() { return info; }
32    public void setInfo(String info) { this.info = info; }
33    public ArrayList<Integer> getFriends() { return friends; }
34    public int getID() { return personID; }
35    public void addFriend(int id) { friends.add(id); }
36 }
```

There are more optimizations and follow-up questions here than we could possibly discuss, but here are just a few possibilities.

Optimization: Reduce machine jumps

Jumping from one machine to another is expensive. Instead of randomly jumping from machine to machine with each friend, try to batch these jumps—e.g., if five of my friends live on one machine, I should look them up all at once.

Optimization: Smart division of people and machines

People are much more likely to be friends with people who live in the same country as they do. Rather than randomly dividing people across machines, try to divide them by country, city, state, and so on. This will reduce the number of jumps.

Question: Breadth-first search usually requires "marking" a node as visited. How do you do that in this case?

Usually, in BFS, we mark a node as visited by setting a `visited` flag in its node class. Here, we don't want to do that. There could be multiple searches going on at the same time, so it's a bad idea to just edit our data.

Instead, we could mimic the marking of nodes with a hash table to look up a node id and determine whether it's been visited.

Other Follow-Up Questions:

- In the real world, servers fail. How does this affect you?
- How could you take advantage of caching?
- Do you search until the end of the graph (infinite)? How do you decide when to give up?
- In real life, some people have more friends of friends than others, and are therefore more likely to make a path between you and someone else. How could you use this data to pick where to start traversing?

These are just a few of the follow-up questions you or the interviewer could raise. There are many others.

9.3 Web Crawler: If you were designing a web crawler, how would you avoid getting into infinite loops?

pg 145

SOLUTION

The first thing to ask ourselves in this problem is how an infinite loop might occur. The simplest answer is that, if we picture the web as a graph of links, an infinite loop will occur when a cycle occurs.

To prevent infinite loops, we just need to detect cycles. One way to do this is to create a hash table where we set `hash[v]` to `true` after we visit page `v`.

We can crawl the web using breadth-first search. Each time we visit a page, we gather all its links and insert them at the end of a queue. If we've already visited a page, we ignore it.

This is great—but what does it mean to visit page `v`? Is page `v` defined based on its content or its URL?

If it's defined based on its URL, we must recognize that URL parameters might indicate a completely different page. For example, the page `www.careercup.com/page?pid=microsoft-interview-questions` is totally different from the page `www.careercup.com/page?pid=google-interview-questions`. But, we can also append URL parameters arbitrarily to any URL without truly changing the page, provided it's not a parameter that the web application recognizes and handles. The page `www.careercup.com?foobar=hello` is the same as `www.careercup.com`.

"Okay, then," you might say, "let's define it based on its content." That sounds good too, at first, but it also doesn't quite work. Suppose I have some randomly generated content on the careercup.com home page. Is it a different page each time you visit it? Not really.

The reality is that there is probably no perfect way to define a "different" page, and this is where this problem gets tricky.

One way to tackle this is to have some sort of estimation for degree of similarity. If, based on the content and the URL, a page is deemed to be sufficiently similar to other pages, we *deprioritize* crawling its children. For each page, we would come up with some sort of signature based on snippets of the content and the page's URL.

Let's see how this would work.

We have a database which stores a list of items we need to crawl. On each iteration, we select the highest priority page to crawl. We then do the following:

1. Open up the page and create a signature of the page based on specific subsections of the page and its URL.
2. Query the database to see whether anything with this signature has been crawled recently.
3. If something with this signature has been recently crawled, insert this page back into the database at a low priority.
4. If not, crawl the page and insert its links into the database.

Under the above implementation, we never "complete" crawling the web, but we will avoid getting stuck in a loop of pages. If we want to allow for the possibility of "finishing" crawling the web (which would clearly happen only if the "web" were actually a smaller system, like an intranet), then we can set a minimum priority that a page must have to be crawled.

This is just one, simplistic solution, and there are many others that are equally valid. A problem like this will more likely resemble a conversation with your interviewer which could take any number of paths. In fact, the discussion of this problem could have taken the path of the very next problem.

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

pg 145

SOLUTION

Just how much space do 10 billion URLs take up? If each URL is an average of 100 characters, and each character is 4 bytes, then this list of 10 billion URLs will take up about 4 terabytes. We are probably not going to hold that much data in memory.

But, let's just pretend for a moment that we were miraculously holding this data in memory, since it's useful to first construct a solution for the simple version. Under this version of the problem, we would just create a hash table where each URL maps to `true` if it's already been found elsewhere in the list. (As an alternative solution, we could sort the list and look for the duplicate values that way. That will take a bunch of extra time and offers few advantages.)

Now that we have a solution for the simple version, what happens when we have all 4000 gigabytes of data and we can't store it all in memory? We could solve this either by storing some of the data on disk or by splitting up the data across machines.

Solution #1: Disk Storage

If we stored all the data on one machine, we would do two passes of the document. The first pass would split the list of URLs into 4000 chunks of 1 GB each. An easy way to do that might be to store each URL u in a file named `<x>.txt` where `x = hash(u) % 4000`. That is, we divide up the URLs based on their hash value (modulo the number of chunks). This way, all URLs with the same hash value would be in the same file.

In the second pass, we would essentially implement the simple solution we came up with earlier: load each file into memory, create a hash table of the URLs, and look for duplicates.

Solution #2: Multiple Machines

The other solution is to perform essentially the same procedure, but to use multiple machines. In this solution, rather than storing the data in file `<x>.txt`, we would send the URL to machine `x`.

Using multiple machines has pros and cons.

The main pro is that we can parallelize the operation, such that all 4000 chunks are processed simultaneously. For large amounts of data, this might result in a faster solution.

The disadvantage though is that we are now relying on 4000 different machines to operate perfectly. That may not be realistic (particularly with more data and more machines), and we'll need to start considering how to handle failure. Additionally, we have increased the complexity of the system simply by involving so many machines.

Both are good solutions, though, and both should be discussed with your interviewer.

9.5 **Cache:** 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 cannot guarantee that the same machine will always respond to the same request. The method `processSearch` is very expensive. Design a caching mechanism to cache the results of the most recent queries. Be sure to explain how you would update the cache when data changes.

pg 145

SOLUTION

Before getting into the design of this system, we first have to understand what the question means. Many of the details are somewhat ambiguous, as is expected in questions like this. We will make reasonable assumptions for the purposes of this solution, but you should discuss these details—in depth—with your interviewer.

Assumptions

Here are a few of the assumptions we make for this solution. Depending on the design of your system and how you approach the problem, you may make other assumptions. Remember that while some approaches are better than others, there is no one "correct" approach.

- Other than calling out to `processSearch` as necessary, all query processing happens on the initial machine that was called.
- The number of queries we wish to cache is large (millions).
- Calling between machines is relatively quick.
- The result for a given query is an ordered list of URLs, each of which has an associated 50 character title and 200 character summary.
- The most popular queries are extremely popular, such that they would always appear in the cache.

Again, these aren't the *only* valid assumptions. This is just one reasonable set of assumptions.

System Requirements

When designing the cache, we know we'll need to support two primary functions:

- Efficient lookups given a key.
- Expiration of old data so that it can be replaced with new data.

In addition, we must also handle updating or clearing the cache when the results for a query change. Because some queries are very common and may permanently reside in the cache, we cannot just wait for the cache to naturally expire.

Step 1: Design a Cache for a Single System

A good way to approach this problem is to start by designing it for a single machine. So, how would you create a data structure that enables you to easily purge old data and also efficiently look up a value based on a key?

- A linked list would allow easy purging of old data, by moving "fresh" items to the front. We could implement it to remove the last element of the linked list when the list exceeds a certain size.

- A hash table allows efficient lookups of data, but it wouldn't ordinarily allow easy data purging.

How can we get the best of both worlds? By merging the two data structures. Here's how this works:

Just as before, we create a linked list where a node is moved to the front every time it's accessed. This way, the end of the linked list will always contain the stalest information.

In addition, we have a hash table that maps from a query to the corresponding node in the linked list. This allows us to not only efficiently return the cached results, but also to move the appropriate node to the front of the list, thereby updating its "freshness."

For illustrative purposes, abbreviated code for the cache is below. The code attachment provides the full code for this part. Note that in your interview, it is unlikely that you would be asked to write the full code for this as well as perform the design for the larger system.

```
1   public class Cache {
2     public static int MAX_SIZE = 10;
3     public Node head, tail;
4     public HashMap<String, Node> map;
5     public int size = 0;
6
7     public Cache() {
8       map = new HashMap<String, Node>();
9     }
10
11    /* Moves node to front of linked list */
12    public void moveToFront(Node node) { ... }
13    public void moveToFront(String query) { ... }
14
15    /* Removes node from linked list */
16    public void removeFromLinkedList(Node node) { ... }
17
18    /* Gets results from cache, and updates linked list */
19    public String[] getResults(String query) {
20      if (!map.containsKey(query)) return null;
21
22      Node node = map.get(query);
23      moveToFront(node); // update freshness
24      return node.results;
25    }
26
27    /* Inserts results into linked list and hash */
28    public void insertResults(String query, String[] results) {
29      if (map.containsKey(query)) { // update values
30        Node node = map.get(query);
31        node.results = results;
32        moveToFront(node); // update freshness
33        return;
34      }
35
36      Node node = new Node(query, results);
37      moveToFront(node);
38      map.put(query, node);
39
40      if (size > MAX_SIZE) {
41        map.remove(tail.query);
42        removeFromLinkedList(tail);
43      }
```

```
44      }
45  }
```

Step 2: Expand to Many Machines

Now that we understand how to design this for a single machine, we need to understand how we would design this when queries could be sent to many different machines. Recall from the problem statement that there's no guarantee that a particular query will be consistently sent to the same machine.

The first thing we need to decide is to what extent the cache is shared across machines. We have several options to consider.

Option 1: Each machine has its own cache.

A simple option is to give each machine its own cache. This means that if "foo" is sent to machine 1 twice in a short amount of time, the result would be recalled from the cache on the second time. But, if "foo" is sent first to machine 1 and then to machine 2, it would be treated as a totally fresh query both times.

This has the advantage of being relatively quick, since no machine-to-machine calls are used. The cache, unfortunately, is somewhat less effective as an optimization tool as many repeat queries would be treated as fresh queries.

Option 2: Each machine has a copy of the cache.

On the other extreme, we could give each machine a complete copy of the cache. When new items are added to the cache, they are sent to all machines. The entire data structure—linked list and hash table—would be duplicated.

This design means that common queries would nearly always be in the cache, as the cache is the same everywhere. The major drawback however is that updating the cache means firing off data to N different machines, where N is the size of the response cluster. Additionally, because each item effectively takes up N times as much space, our cache would hold much less data.

Option 3: Each machine stores a segment of the cache.

A third option is to divide up the cache, such that each machine holds a different part of it. Then, when machine `i` needs to look up the results for a query, machine `i` would figure out which machine holds this value, and then ask this other machine (machine `j`) to look up the query in `j`'s cache.

But how would machine `i` know which machine holds this part of the hash table?

One option is to assign queries based on the formula `hash(query) % N`. Then, machine `i` only needs to apply this formula to know that machine `j` should store the results for this query.

So, when a new query comes in to machine `i`, this machine would apply the formula and call out to machine `j`. Machine `j` would then return the value from its cache or call `processSearch(query)` to get the results. Machine `j` would update its cache and return the results back to `i`.

Alternatively, you could design the system such that machine `j` just returns `null` if it doesn't have the `query` in its current cache. This would require machine `i` to call `processSearch` and then forward the results to machine `j` for storage. This implementation actually increases the number of machine-to-machine calls, with few advantages.

Step 3: Updating results when contents change

Recall that some queries may be so popular that, with a sufficiently large cache, they would permanently be cached. We need some sort of mechanism to allow cached results to be refreshed, either periodically or "on-demand" when certain content changes.

To answer this question, we need to consider when results would change (and you need to discuss this with your interviewer). The primary times would be when:

1. The content at a URL changes (or the page at that URL is removed).
2. The ordering of results change in response to the rank of a page changing.
3. New pages appear related to a particular query.

To handle situations #1 and #2, we could create a separate hash table that would tell us which cached queries are tied to a specific URL. This could be handled completely separately from the other caches, and reside on different machines. However, this solution may require a lot of data.

Alternatively, if the data doesn't require instant refreshing (which it probably doesn't), we could periodically crawl through the cache stored on each machine to purge queries tied to the updated URLs.

Situation #3 is substantially more difficult to handle. We could update single word queries by parsing the content at the new URL and purging these one-word queries from the caches. But, this will only handle the one-word queries.

A good way to handle Situation #3 (and likely something we'd want to do anyway) is to implement an "automatic time-out" on the cache. That is, we'd impose a time out where *no* query, regardless of how popular it is, can sit in the cache for more than `x` minutes. This will ensure that all data is periodically refreshed.

Step 4: Further Enhancements

There are a number of improvements and tweaks you could make to this design depending on the assumptions you make and the situations you optimize for.

One such optimization is to better support the situation where some queries are very popular. For example, suppose (as an extreme example) a particular string constitutes 1% of all queries. Rather than machine `i` forwarding the request to machine `j` every time, machine `i` could forward the request just once to `j`, and then `i` could store the results in its own cache as well.

Alternatively, there may also be some possibility of doing some sort of re-architecture of the system to assign queries to machines based on their hash value (and therefore the location of the cache), rather than randomly. However, this decision may come with its own set of trade-offs.

Another optimization we could make is to the "automatic time out" mechanism. As initially described, this mechanism purges any data after `X` minutes. However, we may want to update some data (like current news) much more frequently than other data (like historical stock prices). We could implement timeouts based on topic or based on URLs. In the latter situation, each URL would have a time out value based on how frequently the page has been updated in the past. The time out for the query would be the minimum of the time outs for each URL.

These are just a few of the enhancements we can make. Remember that in questions like this, there is no single correct way to solve the problem. These questions are about having a discussion with your interviewer about design criteria and demonstrating your general approach and methodology.

9.6 **Sales Rank**: A large eCommerce company wishes to list the best-selling products, overall and by category. For example, one product might be the #1056th best-selling product overall but the #13th best-selling product under "Sports Equipment" and the #24th best-selling product under "Safety." Describe how you would design this system.

pg 145

SOLUTION

Let's first start off by making some assumptions to define the problem.

Step 1: Scope the Problem

First, we need to define what exactly we're building.

- We'll assume that we're only being asked to design the components relevant to this question, and not the entire eCommerce system. In this case, we might touch the design of the frontend and purchase components, but only as it impacts the sales rank.
- We should also define what the sales rank means. Is it total sales over all time? Sales in the last month? Last week? Or some more complicated function (such as one involving some sort of exponential decay of sales data)? This would be something to discuss with your interviewer. We will assume that it is simply the total sales over the past week.
- We will assume that each product can be in multiple categories, and that there is no concept of "subcategories."

This part just gives us a good idea of what the problem, or scope of features, is.

Step 2: Make Reasonable Assumptions

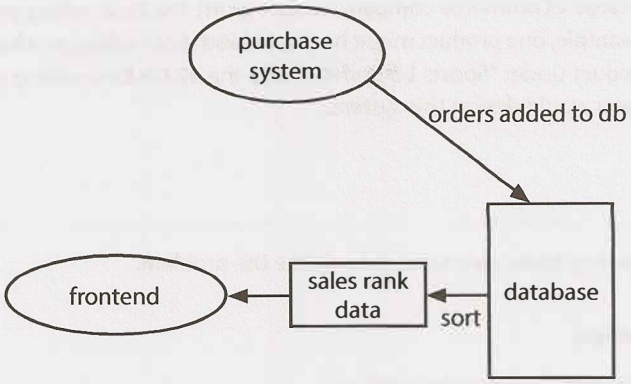
These are the sorts of things you'd want to discuss with your interviewer. Because we don't have an interviewer in front of us, we'll have to make some assumptions.

- We will assume that the stats do not need to be 100% up-to-date. Data can be up to an hour old for the most popular items (for example, top 100 in each category), and up to one day old for the less popular items. That is, few people would care if the #2,809,132th best-selling item should have actually been listed as #2,789,158th instead.
- Precision is important for the most popular items, but a small degree of error is okay for the less popular items.
- We will assume that the data should be updated every hour (for the most popular items), but the time range for this data does not need to be precisely the last seven days (168 hours). If it's sometimes more like 150 hours, that's okay.
- We will assume that the categorizations are based strictly on the origin of the transaction (i.e., the seller's name), not the price or date.

The important thing is not so much which decision you made at each possible issue, but whether it occurred to you that these are assumptions. We should get out as many of these assumptions as possible in the beginning. It's possible you will need to make other assumptions along the way.

Step 3: Draw the Major Components

We should now design just a basic, naive system that describes the major components. This is where you would go up to a whiteboard.



In this simple design, we store every order as soon as it comes into the database. Every hour or so, we pull sales data from the database by category, compute the total sales, sort it, and store it in some sort of sales rank data cache (which is probably held in memory). The frontend just pulls the sales rank from this table, rather than hitting the standard database and doing its own analytics.

Step 4: Identify the Key Issues

Analytics are Expensive

In the naive system, we periodically query the database for the number of sales in the past week for each product. This will be fairly expensive. That's running a query over all sales for all time.

Our database just needs to track the total sales. We'll assume (as noted in the beginning of the solution) that the general storage for purchase history is taken care of in other parts of the system, and we just need to focus on the sales data analytics.

Instead of listing every purchase in our database, we'll store just the total sales from the last week. Each purchase will just update the total weekly sales.

Tracking the total sales takes a bit of thought. If we just use a single column to track the total sales over the past week, then we'll need to re-compute the total sales every day (since the specific days covered in the last seven days change with each day). That is unnecessarily expensive.

Instead, we'll just use a table like this.

Prod ID	Total	Sun	Mon	Tues	Wed	Thurs	Fri	Sat

This is essentially like a circular array. Each day, we clear out the corresponding day of the week. On each purchase, we update the total sales count for that product on that day of the week, as well as the total count.

We will also need a separate table to store the associations of product IDs and categories.

Prod ID	Category ID

To get the sales rank per category, we'll need to join these tables.

Database Writes are Very Frequent

Even with this change, we'll still be hitting the database very frequently. With the amount of purchases that could come in every second, we'll probably want to batch up the database writes.

Instead of immediately committing each purchase to the database, we could store purchases in some sort of in-memory cache (as well as to a log file as a backup). Periodically, we'll process the log / cache data, gather the totals, and update the database.

> We should quickly think about whether or not it's feasible to hold this in memory. If there are 10 million products in the system, can we store each (along with a count) in a hash table? Yes. If each product ID is four bytes (which is big enough to hold up to 4 billion unique IDs) and each count is four bytes (more than enough), then such a hash table would only take about 40 megabytes. Even with some additional overhead and substantial system growth, we would still be able to fit this all in memory.

After updating the database, we can re-run the sales rank data.

We need to be a bit careful here, though. If we process one product's logs before another's, and re-run the stats in between, we could create a bias in the data (since we're including a larger timespan for one product than its "competing" product).

We can resolve this by either ensuring that the sales rank doesn't run until all the stored data is processed (difficult to do when more and more purchases are coming in), or by dividing up the in-memory cache by some time period. If we update the database for all the stored data up to a particular moment in time, this ensures that the database will not have biases.

Joins are Expensive

We have potentially tens of thousands of product categories. For each category, we'll need to first pull the data for its items (possibly through an expensive join) and then sort those.

Alternatively, we could just do one join of products and categories, such that each product will be listed once per category. Then, if we sorted that on category and then product ID, we could just walk the results to get the sales rank for each category.

Prod ID	Category	Total	Sun	Mon	Tues	Wed	Thurs	Fri	Sat
1423	sportseq	13	4	1	4	19	322	32	232
1423	safety	13	4	1	4	19	322	32	232

Rather than running thousands of queries (one for each category), we could sort the data on the category first and then the sales volume. Then, if we walked those results, we would get the sales rank for each category. We would also need to do one sort of the entire table on just sales number, to get the overall rank.

We could also just keep the data in a table like this from the beginning, rather than doing joins. This would require us to update multiple rows for each product.

Database Queries Might Still Be Expensive

Alternatively, if the queries and writes get very expensive, we could consider forgoing a database entirely and just using log files. This would allow us to take advantage of something like MapReduce.

Under this system, we would write a purchase to a simple text file with the product ID and time stamp. Each category has its own directory, and each purchase gets written to all the categories associated with that product.

We would run frequent jobs to merge files together by product ID and time ranges, so that eventually all purchases in a given day (or possibly hour) were grouped together.

```
/sportsequipment
    1423,Dec 13 08:23-Dec 13 08:23,1
    4221,Dec 13 15:22-Dec 15 15:45,5
    ...
/safety
    1423,Dec 13 08:23-Dec 13 08:23,1
    5221,Dec 12 03:19-Dec 12 03:28,19
...
```

To get the best-selling products within each category, we just need to sort each directory.

How do we get the overall ranking? There are two good approaches:

- We could treat the general category as just another directory, and write every purchase to that directory. That would mean a lot of files in this directory.
- Or, since we'll already have the products sorted by sales volume order for each category, we can also do an N-way merge to get the overall rank.

Alternatively, we can take advantage of the fact that the data doesn't need (as we assumed earlier) to be 100% up-to-date. We just need the most popular items to be up-to-date.

We can merge the most popular items from each category in a pairwise fashion. So, two categories get paired together and we merge the most popular items (the first 100 or so). After we have 100 items in this sorted order, we stop merging this pair and move onto the next pair.

To get the ranking for all products, we can be much lazier and only run this work once a day.

One of the advantages of this is that it scales nicely. We can easily divide up the files across multiple servers, as they aren't dependent on each other.

Follow Up Questions

The interviewer could push this design in any number of directions.

- Where do you think you'd hit the next bottlenecks? What would you do about that?
- What if there were subcategories as well? So items could be listed under "Sports" and "Sports Equipment" (or even "Sports" > "Sports Equipment" > "Tennis" > "Rackets")?
- What if data needed to be more accurate? What if it needed to be accurate within 30 minutes for all products?

Think through your design carefully and analyze it for the tradeoffs. You might also be asked to go into more detail on any specific aspect of the product.

9.7 **Personal Financial Manager**: Explain how you would design a personal financial manager (like Mint.com). This system would connect to your bank accounts, analyze your spending habits, and make recommendations.

pg 145

SOLUTION

The first thing we need to do is define what it is exactly that we are building.

Step 1: Scope the Problem

Ordinarily, you would clarify this system with your interviewer. We'll scope the problem as follows:

- You create an account and add your bank accounts. You can add multiple bank accounts. You can also add them at a later point in time.
- It pulls in all your financial history, or as much of it as your bank will allow.
- This financial history includes outgoing money (things you bought or paid for), incoming money (salary and other payments), and your current money (what's in your bank account and investments).
- Each payment transaction has a "category" associated with it (food, travel, clothing, etc.).
- There is some sort of data source provided that tells the system, with some reliability, which category a transaction is associated with. The user might, in some cases, override the category when it's improperly assigned (e.g., eating at the cafe of a department store getting assigned to "clothing" rather than "food").
- Users will use the system to get recommendations on their spending. These recommendations will come from a mix of "typical" users ("people generally shouldn't spend more than X% of their income on clothing"), but can be overridden with custom budgets. This will not be a primary focus right now.
- We assume this is just a website for now, although we could potentially talk about a mobile app as well.
- We probably want email notifications either on a regular basis, or on certain conditions (spending over a certain threshold, hitting a budget max, etc.).
- We'll assume that there's no concept of user-specified rules for assigning categories to transactions.

This gives us a basic goal for what we want to build.

Step 2: Make Reasonable Assumptions

Now that we have the basic goal for the system, we should define some further assumptions about the characteristics of the system.

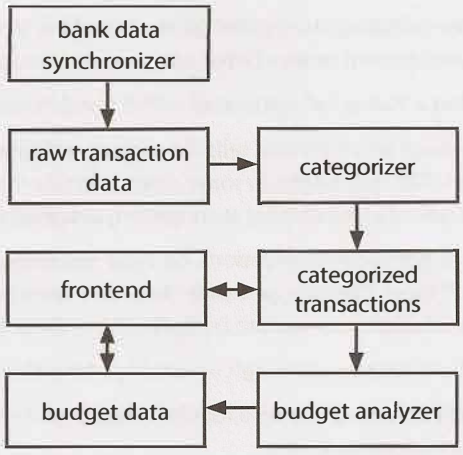
- Adding or removing bank accounts is relatively unusual.
- The system is write-heavy. A typical user may make several new transactions daily, although few users would access the website more than once a week. In fact, for many users, their primary interaction might be through email alerts.
- Once a transaction is assigned to a category, it will only be changed if the user asks to change it. The system will never reassign a transaction to a different category "behind the scenes", even if the rules change. This means that two otherwise identical transactions could be assigned to different categories if the rules changed in between each transaction's date. We do this because it may confuse users if their spending per category changes with no action on their part.
- The banks probably won't push data to our system. Instead, we will need to pull data from the banks.
- Alerts on users exceeding budgets probably do not need to be sent instantaneously. (That wouldn't be realistic anyway, since we won't get the transaction data instantaneously.) It's probably pretty safe for them to be up to 24 hours delayed.

It's okay to make different assumptions here, but you should explicitly state them to your interviewer.

Step 3: Draw the Major Components

The most naive system would be one that pulls bank data on each login, categorizes all the data, and then analyzes the user's budget. This wouldn't quite fit the requirements, though, as we want email notifications on particular events.

We can do a bit better.



With this basic architecture, the bank data is pulled at periodic times (hourly or daily). The frequency may depend on the behavior of the users. Less active users may have their accounts checked less frequently.

Once new data arrives, it is stored in some list of raw, unprocessed transactions. This data is then pushed to the categorizer, which assigns each transaction to a category and stores these categorized transactions in another datastore.

The budget analyzer pulls in the categorized transactions, updates each user's budget per category, and stores the user's budget.

The frontend pulls data from both the categorized transactions datastore as well as from the budget datastore. Additionally, a user could also interact with the frontend by changing the budget or the categorization of their transactions.

Step 4: Identify the Key Issues

We should now reflect on what the major issues here might be.

This will be a very data-heavy system. We want it to feel snappy and responsive, though, so we'll want as much processing as possible to be asynchronous.

We will almost certainly want at least one task queue, where we can queue up work that needs to be done. This work will include tasks such as pulling in new bank data, re-analyzing budgets, and categorizing new bank data. It would also include re-trying tasks that failed.

These tasks will likely have some sort of priority associated with them, as some need to be performed more often than others. We want to build a task queue system that can prioritize some task types over others, while still ensuring that all tasks will be performed eventually. That is, we wouldn't want a low priority task to essentially "starve" because there are always higher priority tasks.

One important part of the system that we haven't yet addressed will be the email system. We could use a task to regularly crawl user's data to check if they're exceeding their budget, but that means checking every

single user daily. Instead, we'll want to queue a task whenever a transaction occurs that potentially exceeds a budget. We can store the current budget totals by category to make it easy to understand if a new transaction exceeds the budget.

We should also consider incorporating the knowledge (or assumption) that a system like this will probably have a large number of inactive users—users who signed up once and then haven't touched the system since. We may want to either remove them from the system entirely or deprioritize their accounts. We'll want some system to track their account activity and associate priority with their accounts.

The biggest bottleneck in our system will likely be the massive amount of data that needs to be pulled and analyzed. We should be able to fetch the bank data asynchronously and run these tasks across many servers. We should drill a bit deeper into how the categorizer and budget analyzer work.

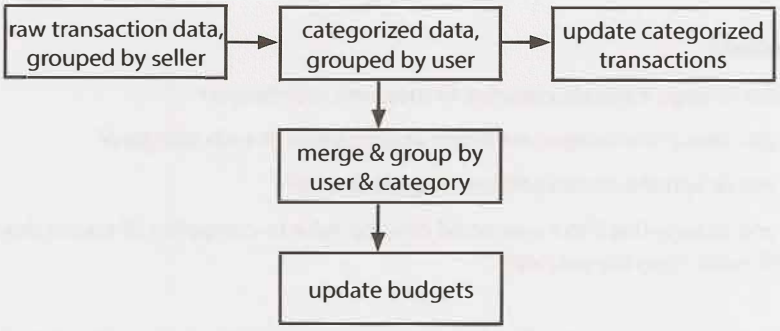
Categorizer and Budget Analyzer

One thing to note is that transactions are not dependent on each other. As soon as we get a transaction for a user, we can categorize it and integrate this data. It might be inefficient to do so, but it won't cause any inaccuracies.

Should we use a standard database for this? With lots of transactions coming in at once, that might not be very efficient. We certainly don't want to do a bunch of joins.

It may be better instead to just store the transactions to a set of flat text files. We assumed earlier that the categorizations are based on the seller's name alone. If we're assuming a lot of users, then there will be a lot of duplicates across the sellers. If we group the transaction files by seller's name, we can take advantage of these duplicates.

The categorizer can do something like this:



It first gets the raw transaction data, grouped by seller. It picks the appropriate category for the seller (which might be stored in a cache for the most common sellers), and then applies that category to all those transactions.

After applying the category, it re-groups all the transactions by user. Then, those transactions are inserted into the datastore for this user.

before categorizer	after categorizer

amazon/
  user121,$5.43,Aug 13
  user922,$15.39,Aug 27
  ...
comcast/
  user922,$9.29,Aug 24
  user248,$40.13,Aug 18
  ...
	user121/
	  amazon,shopping,$5.43,Aug 13
	  ...
	user922/
	  amazon,shopping,$15.39,Aug 27
	  comcast,utilities,$9.29,Aug 24
	  ...
	user248/
	  comcast,utilities,$40.13,Aug 18
	  ...

Then, the budget analyzer comes in. It takes the data grouped by user, merges it across categories (so all Shopping tasks for this user in this timespan are merged), and then updates the budget.

Most of these tasks will be handled in simple log files. Only the final data (the categorized transactions and the budget analysis) will be stored in a database. This minimizes writing and reading from the database.

User Changing Categories

The user might selectively override particular transactions to assign them to a different category. In this case, we would update the datastore for the categorized transactions. It would also signal a quick recomputation of the budget to decrement the item from the old category and increment the item in the other category.

We could also just recompute the budget from scratch. The budget analyzer is fairly quick as it just needs to look over the past few weeks of transactions for a single user.

Follow Up Questions

- How would this change if you also needed to support a mobile app?
- How would you design the component which assigns items to each category?
- How would you design the recommended budgets feature?
- How would you change this if the user could develop rules to categorize all transactions from a particular seller differently than the default?

9.8 **Pastebin**: Design a system like Pastebin, where a user can enter a piece of text and get a randomly generated URL for public access.

pg 145

SOLUTION

We can start with clarifying the specifics of this system.

Step 1: Scope the Problem

- The system does not support user accounts or editing documents.
- The system tracks analytics of how many times each page is accessed.
- Old documents get deleted after not being accessed for a sufficiently long period of time.
- While there isn't true authentication on accessing documents, users should not be able to "guess" docu-

ment URLs easily.

- The system has a frontend as well as an API.
- The analytics for each URL can be accessed through a "stats" link on each page. It is not shown by default, though.

Step 2: Make Reasonable Assumptions

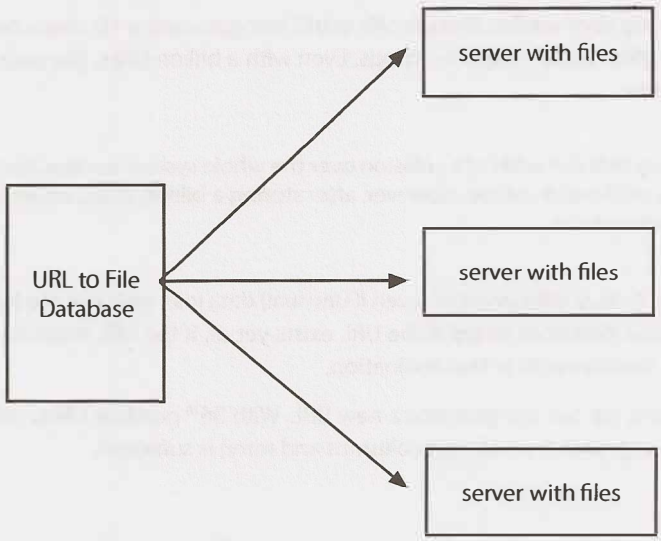
- The system gets heavy traffic and contains many millions of documents.
- Traffic is not equally distributed across documents. Some documents get much more access than others.

Step 3: Draw the Major Components

We can sketch out a simple design. We'll need to keep track of URLs and the files associated with them, as well as analytics for how often the files have been accessed.

How should we store the documents? We have two options: we can store them in a database or we can store them on a file. Since the documents can be large and it's unlikely we need searching capabilities, storing them on a file is probably the better choice.

A simple design like this might work well:



Here, we have a simple database that looks up the location (server and path) of each file. When we have a request for a URL, we look up the location of the URL within the datastore and then access the file.

Additionally, we will need a database that tracks analytics. We can do this with a simple datastore that adds each visit (including timestamp, IP address, and location) as a row in a database. When we need to access the stats of each visit, we pull the relevant data in from this database.

Step 4: Identify the Key Issues

The first issue that comes to mind is that some documents will be accessed much more frequently than others. Reading data from the filesystem is relatively slow compared with reading from data in memory. Therefore, we probably want to use a cache to store the most recently accessed documents. This will ensure

that items accessed very frequently (or very recently) will be quickly accessible. Since documents cannot be edited, we will not need to worry about invalidating this cache.

We should also potentially consider sharding the database. We can shard it using some mapping from the URL (for example, the URL's hash code modulo some integer), which will allow us to quickly locate the database which contains this file.

In fact, we could even take this a step further. We could skip the database entirely and just let a hash of the URL indicate which server contains the document. The URL itself could reflect the location of the document. One potential issue from this is that if we need to add servers, it could be difficult to redistribute the documents.

Generating URLs

We have not yet discussed how to actually generate the URLs. We probably do not want a monotonically increasing integer value, as this would be easy for a user to "guess." We want URLs to be difficult to access without being provided the link.

One simple path is to generate a random GUID (e.g., 5d50e8ac-57cb-4a0d-8661-bcdee2548979). This is a 128-bit value that, while not strictly guaranteed to be unique, has low enough odds of a collision that we can treat it as unique. The drawback of this plan is that such a URL is not very "pretty" to the user. We could hash it to a smaller value, but then that increases the odds of collision.

We could do something very similar, though. We could just generate a 10-character sequence of letters and numbers, which gives us 36^{10} possible strings. Even with a billion URLs, the odds of a collision on any specific URL are very low.

> This is not to say that the odds of a collision over the whole system are low. They are not. Any one specific URL is unlikely to collide. However, after storing a billion URLs, we are very likely to have a collision at some point.

Assuming that we aren't okay with periodic (even if unusual) data loss, we'll need to handle these collisions. We can either check the datastore to see if the URL exists yet or, if the URL maps to a specific server, just detect whether a file already exists at the destination.

When a collision occurs, we can just generate a new URL. With 36^{10} possible URLs, collisions would be rare enough that the lazy approach here (detect collisions and retry) is sufficient.

Analytics

The final component to discuss is the analytics piece. We probably want to display the number of visits, and possibly break this down by location or time.

We have two options here:

- Store the raw data from each visit.
- Store just the data we know we'll use (number of visits, etc.).

You can discuss this with your interviewer, but it probably makes sense to store the raw data. We never know what features we'll add to the analytics down the road. The raw data allows us flexibility.

This does not mean that the raw data needs to be easily searchable or even accessible. We can just store a log of each visit in a file, and back this up to other servers.

One issue here is that this amount of data could be substantial. We could potentially reduce the space usage considerably by storing data only probabilistically. Each URL would have a `storage_probability` associated with it. As the popularity of a site goes up, the `storage_probability` goes down. For example, a popular document might have data logged only one out of every ten times, at random. When we look up the number of visits for the site, we'll need to adjust the value based on the probability (for example, by multiplying it by 10). This will of course lead to a small inaccuracy, but that may be acceptable.

The log files are not designed to be used frequently. We will want to also store this precomputed data in a datastore. If the analytics just displays the number of visits plus a graph over time, this could be kept in a separate database.

URL	Month and Year	Visits
12ab31b92p	December 2013	242119
12ab31b92p	January 2014	429918
...	...	...

Every time a URL is visited, we can increment the appropriate row and column. This datastore can also be sharded by the URL.

As the stats are not listed on the regular pages and would generally be of less interest, it should not face as heavy of a load. We could still cache the generated HTML on the frontend servers, so that we don't continuously reaccess the data for the most popular URLs.

Follow-Up Questions

- How would you support user accounts?
- How would you add a new piece of analytics (e.g., referral source) to the stats page?
- How would your design change if the stats were shown with each document?

10

Solutions to Sorting and Searching

10.1 Sorted Merge: 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 149

SOLUTION

Since we know that A has enough buffer at the end, we won't need to allocate additional space. Our logic should involve simply comparing elements of A and B and inserting them in order, until we've exhausted all elements in A and in B.

The only issue with this is that if we insert an element into the front of A, then we'll have to shift the existing elements backwards to make room for it. It's better to insert elements into the back of the array, where there's empty space.

The code below does just that. It works from the back of A and B, moving the largest elements to the back of A.

```
1   void merge(int[] a, int[] b, int lastA, int lastB) {
2      int indexA = lastA - 1; /* Index of last element in array a */
3      int indexB = lastB - 1; /* Index of last element in array b */
4      int indexMerged = lastB + lastA - 1; /* end of merged array */
5
6      /* Merge a and b, starting from the last element in each */
7      while (indexB >= 0) {
8         /* end of a is > than end of b */
9         if (indexA >= 0 && a[indexA] > b[indexB]) {
10           a[indexMerged] = a[indexA]; // copy element
11           indexA--;
12        } else {
13           a[indexMerged] = b[indexB]; // copy element
14           indexB--;
15        }
16        indexMerged--; // move indices
17     }
18  }
```

Note that you don't need to copy the contents of A after running out of elements in B. They are already in place.

10.2 Group Anagrams: Write a method to sort an array of strings so that all the anagrams are next to each other.

pg 150

SOLUTION

This problem asks us to group the strings in an array such that the anagrams appear next to each other. Note that no specific ordering of the words is required, other than this.

We need a quick and easy way of determining if two strings are anagrams of each other. What defines if two words are anagrams of each other? Well, anagrams are words that have the same characters but in different orders. It follows then that if we can put the characters in the same order, we can easily check if the new words are identical.

One way to do this is to just apply any standard sorting algorithm, like merge sort or quick sort, and modify the comparator. This comparator will be used to indicate that two strings which are anagrams of each other are equivalent.

What's the easiest way of checking if two words are anagrams? We could count the occurrences of the distinct characters in each string and return `true` if they match. Or, we could just sort the string. After all, two words which are anagrams will look the same once they're sorted.

The code below implements the comparator.

```
1  class AnagramComparator implements Comparator<String> {
2    public String sortChars(String s) {
3      char[] content = s.toCharArray();
4      Arrays.sort(content);
5      return new String(content);
6    }
7
8    public int compare(String s1, String s2) {
9      return sortChars(s1).compareTo(sortChars(s2));
10   }
11 }
```

Now, just sort the arrays using this `compareTo` method instead of the usual one.

```
12 Arrays.sort(array, new AnagramComparator());
```

This algorithm will take `O(n log(n))` time.

This may be the best we can do for a general sorting algorithm, but we don't actually need to fully sort the array. We only need to *group* the strings in the array by anagram.

We can do this by using a hash table which maps from the sorted version of a word to a list of its anagrams. So, for example, `acre` will map to the list `{acre, race, care}`. Once we've grouped all the words into these lists by anagram, we can then put them back into the array.

The code below implements this algorithm.

```
1  void sort(String[] array) {
2    HashMapList<String, String> mapList = new HashMapList<String, String>();
3
4    /* Group words by anagram */
5    for (String s : array) {
6      String key = sortChars(s);
7      mapList.put(key, s);
8    }
```

```
9
10    / *Convert hash table to array */
11    int index = 0;
12    for (String key : mapList.keySet()) {
13       ArrayList<String> list = mapList.get(key);
14       for (String t : list) {
15          array[index] = t;
16          index++;
17       }
18    }
19 }
20
21 String sortChars(String s) {
22    char[] content = s.toCharArray();
23    Arrays.sort(content);
24    return new String(content);
25 }
26
27 / *HashMapList<String, Integer> is a HashMap that maps from Strings to
28  * ArrayList<Integer>. See appendix for implementation. */
```

You may notice that the algorithm above is a modification of bucket sort.

10.3 Search in Rotated Array: 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 150

SOLUTION

If this problem smells like binary search to you, you're right!

In classic binary search, we compare x with the midpoint to figure out if x belongs on the left or the right side. The complication here is that the array is rotated and may have an inflection point. Consider, for example, the following two arrays:

```
Array1: {10, 15, 20,  0,  5}
Array2: {50,  5, 20, 30, 40}
```

Note that both arrays have a midpoint of 20, but 5 appears on the left side of one and on the right side of the other. Therefore, comparing x with the midpoint is insufficient.

However, if we look a bit deeper, we can see that one half of the array must be ordered normally (in increasing order). We can therefore look at the normally ordered half to determine whether we should search the left or right half.

For example, if we are searching for 5 in `Array1`, we can look at the left element (10) and middle element (20). Since 10 < 20, the left half must be ordered normally. And, since 5 is not between those, we know that we must search the right half.

In Array2, we can see that since 50 > 20, the right half must be ordered normally. We turn to the middle (20) and right (40) element to check if 5 would fall between them. The value 5 would not; therefore, we search the left half.

The tricky condition is if the left and the middle are identical, as in the example array {2, 2, 2, 3, 4, 2}. In this case, we can check if the rightmost element is different. If it is, we can search just the right side. Otherwise, we have no choice but to search both halves.

```
1  int search(int a[], int left, int right, int x) {
2     int mid = (left + right) / 2;
3     if (x == a[mid]) { // Found element
4        return mid;
5     }
6     if (right < left) {
7        return -1;
8     }
9
10    /* Either the left or right half must be normally ordered. Find out which side
11     * is normally ordered, and then use the normally ordered half to figure out
12     * which side to search to find x. */
13    if (a[left] < a[mid]) { // Left is normally ordered.
14       if (x >= a[left] && x < a[mid]) {
15          return search(a, left, mid - 1, x); // Search left
16       } else {
17          return search(a, mid + 1, right, x); // Search right
18       }
19    } else if (a[mid] < a[left]) { // Right is normally ordered.
20       if (x > a[mid] && x <= a[right]) {
21          return search(a, mid + 1, right, x); // Search right
22       } else {
23          return search(a, left, mid - 1, x); // Search left
24       }
25    } else if (a[left] == a[mid]) { // Left or right half is all repeats
26       if (a[mid] != a[right]) { // If right is different, search it
27          return search(a, mid + 1, right, x); // search right
28       } else { // Else, we have to search both halves
29          int result = search(a, left, mid - 1, x); // Search left
30          if (result == -1) {
31             return search(a, mid + 1, right, x); // Search right
32          } else {
33             return result;
34          }
35       }
36    }
37    return -1;
38 }
```

This code will run in O(log n) if all the elements are unique. However, with many duplicates, the algorithm is actually O(n). This is because with many duplicates, we will often have to search both the left and right sides of the array (or subarrays).

Note that while this problem is not conceptually very complex, it is actually very difficult to implement flawlessly. Don't feel bad if you had trouble implementing it without a few bugs. Because of the ease of making off-by-one and other minor errors, you should make sure to test your code very thoroughly.

10.4 Sorted Search, No Size: You are given an array-like data structure `Listy` which lacks a size method. It does, however, have an `elementAt(i)` method that returns the element at index `i` in `O(1)` time. If `i` is beyond the bounds of the data structure, it returns `-1`. (For this reason, the data structure only supports positive integers.) Given a `Listy` which contains sorted, positive integers, find the index at which an element x occurs. If x occurs multiple times, you may return any index.

pg 150

SOLUTION

Our first thought here should be binary search. The problem is that binary search requires us knowing the length of the list, so that we can compare it to the midpoint. We don't have that here.

Could we compute the length? Yes!

We know that `elementAt` will return `-1` when `i` is too large. We can therefore just try bigger and bigger values until we exceed the size of the list.

But how much bigger? If we just went through the list linearly—1, then 2, then 3, then 4, and so on—we'd wind up with a linear time algorithm. We probably want something faster than this. Otherwise, why would the interviewer have specified the list is sorted?

It's better to back off exponentially. Try 1, then 2, then 4, then 8, then 16, and so on. This ensures that, if the list has length n, we'll find the length in at most `O(log n)` time.

> Why `O(log n)`? Imagine we start with pointer q at `q = 1`. At each iteration, this pointer q doubles, until q is bigger than the length n. How many times can q double in size before it's bigger than n? Or, in other words, for what value of k does $2^k = n$? This expression is equal when `k = log n`, as this is precisely what `log` means. Therefore, it will take `O(log n)` steps to find the length.

Once we find the length, we just perform a (mostly) normal binary search. I say "mostly" because we need to make one small tweak. If the mid point is -1, we need to treat this as a "too big" value and search left. This is on line 16 below.

There's one more little tweak. Recall that the way we figure out the length is by calling `elementAt` and comparing it to -1. If, in the process, the element is bigger than the value x (the one we're searching for), we'll jump over to the binary search part early.

```
1   int search(Listy list, int value) {
2     int index = 1;
3     while (list.elementAt(index) != -1 && list.elementAt(index) < value) {
4       index *= 2;
5     }
6     return binarySearch(list, value, index / 2, index);
7   }
8
9   int binarySearch(Listy list, int value, int low, int high) {
10    int mid;
11
12    while (low <= high) {
13      mid = (low + high) / 2;
14      int middle = list.elementAt(mid);
15      if (middle > value || middle == -1) {
16        high = mid - 1;
17      } else if (middle < value) {
```

```
18          low = mid + 1;
19        } else {
20          return mid;
21        }
22      }
23      return -1;
24    }
```

It turns out that not knowing the length didn't impact the runtime of the search algorithm. We find the length in $O(\log n)$ time and then do the search in $O(\log n)$ time. Our overall runtime is $O(\log n)$, just as it would be in a normal array.

10.5 Sparse Search: Given a sorted array of strings that is interspersed with empty strings, write a method to find the location of a given string.

EXAMPLE

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

Output: 4

pg 150

SOLUTION

If it weren't for the empty strings, we could simply use binary search. We would compare the string to be found, `str`, with the midpoint of the array, and go from there.

With empty strings interspersed, we can implement a simple modification of binary search. All we need to do is fix the comparison against `mid`, in case `mid` is an empty string. We simply move `mid` to the closest non-empty string.

The recursive code below to solve this problem can easily be modified to be iterative. We provide such an implementation in the code attachment.

```
1   int search(String[] strings, String str, int first, int last) {
2     if (first > last) return -1;
3     /* Move mid to the middle * /
4     int mid = (last + first) / 2;
5
6     /* If mid is empty, find closest non-empty string. * /
7     if (strings[mid].isEmpty()) {
8       int left = mid - 1;
9       int right = mid + 1;
10      while (true) {
11        if (left < first && right > last) {
12          return -1;
13        } else if (right <= last && !strings[right].isEmpty()) {
14          mid = right;
15          break;
16        } else if (left >= first && !strings[left].isEmpty()) {
17          mid = left;
18          break;
19        }
20        right++;
21        left--;
22      }
23    }
```

```
24
25    /* Check for string, and recurse if necessary */
26    if (str.equals(strings[mid])) { // Found it!
27       return mid;
28    } else if (strings[mid].compareTo(str) < 0) { // Search right
29       return search(strings, str, mid + 1, last);
30    } else { // Search left
31       return search(strings, str, first, mid - 1);
32    }
33 }
34
35 int search(String[] strings, String str) {
36    if (strings == null || str == null || str == "") {
37       return -1;
38    }
39    return search(strings, str, 0, strings.length - 1);
40 }
```

The worst-case runtime for this algorithm is O(n). In fact, it's impossible to have an algorithm for this problem that is better than O(n) in the worst case. After all, you could have an array of all empty strings except for one non-empty string. There is no "smart" way to find this non-empty string. In the worst case, you will need to look at every element in the array.

Careful consideration should be given to the situation when someone searches for the empty string. Should we find the location (which is an O(n) operation)? Or should we handle this as an error?

There's no correct answer here. This is an issue you should raise with your interviewer. Simply asking this question will demonstrate that you are a careful coder.

10.6 **Sort Big File:** Imagine you have a 20 GB file with one string per line. Explain how you would sort the file.

pg 150

SOLUTION

When an interviewer gives a size limit of 20 gigabytes, it should tell you something. In this case, it suggests that they don't want you to bring all the data into memory.

So what do we do? We only bring part of the data into memory.

We'll divide the file into chunks, which are x megabytes each, where x is the amount of memory we have available. Each chunk is sorted separately and then saved back to the file system.

Once all the chunks are sorted, we merge the chunks, one by one. At the end, we have a fully sorted file.

This algorithm is known as external sort.

10.7 **Missing Int:** Given an input file with four billion non-negative integers, provide an algorithm to generate an integer that 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 150

SOLUTION

There are a total of 2^{32}, or 4 billion, distinct integers possible and 2^{31} non-negative integers. Therefore, we know the input file (assuming it is `int`s rather than `long`s) contains some duplicates.

We have 1 GB of memory, or 8 billion bits. Thus, with 8 billion bits, we can map all possible integers to a distinct bit with the available memory. The logic is as follows:

1. Create a bit vector (BV) with 4 billion bits. Recall that a bit vector is an array that compactly stores boolean values by using an array of ints (or another data type). Each int represents 32 boolean values.
2. Initialize BV with all 0s.
3. Scan all numbers (num) from the file and call `BV.set(num, 1)`.
4. Now scan again BV from the 0th index.
5. Return the first index which has a value of 0.

The following code demonstrates our algorithm.

```
1  long numberOfInts = ((long) Integer.MAX_VALUE) + 1;
2  byte[] bitfield = new byte [(int) (numberOfInts / 8)];
3  String filename = ...
4
5  void findOpenNumber() throws FileNotFoundException {
6     Scanner in = new Scanner(new FileReader(filename));
7     while (in.hasNextInt()) {
8        int n = in.nextInt ();
9        /* Finds the corresponding number in the bitfield by using the OR operator to
10        * set the nth bit of a byte (e.g., 10 would correspond to the 2nd bit of
11        * index 2 in the byte array). */
12       bitfield [n / 8] |= 1 << (n % 8);
13    }
14
15    for (int i = 0; i < bitfield.length; i++) {
16       for (int j = 0; j < 8; j++) {
17          /* Retrieves the individual bits of each byte. When 0 bit is found, print
18           * the corresponding value. */
19          if ((bitfield[i] & (1 << j)) == 0) {
20             System.out.println (i * 8 + j);
21             return;
22          }
23       }
24    }
25 }
```

Follow Up: What if we have only 10 MB memory?

It's possible to find a missing integer with two passes of the data set. We can divide up the integers into blocks of some size (we'll discuss how to decide on a size later). Let's just assume that we divide up the integers into blocks of 1000. So, block 0 represents the numbers 0 through 999, block 1 represents numbers 1000 - 1999, and so on.

Since all the values are distinct, we know how many values we *should* find in each block. So, we search through the file and count how many values are between 0 and 999, how many are between 1000 and 1999, and so on. If we count only 999 values in a particular range, then we know that a missing int must be in that range.

In the second pass, we'll actually look for which number in that range is missing. We use the bit vector approach from the first part of this problem. We can ignore any number outside of this specific range.

The question, now, is what is the appropriate block size? Let's define some variables as follows:

- Let `rangeSize` be the size of the ranges that each block in the first pass represents.
- Let `arraySize` represent the number of blocks in the first pass. Note that `arraySize` = $2^{31}/\texttt{rangeSize}$ since there are 2^{31} non-negative integers.

We need to select a value for `rangeSize` such that the memory from the first pass (the array) and the second pass (the bit vector) fit.

First Pass: The Array

The array in the first pass can fit in 10 megabytes, or roughly 2^{23} bytes, of memory. Since each element in the array is an int, and an int is 4 bytes, we can hold an array of at most about 2^{21} elements. So, we can deduce the following:

$$\texttt{arraySize} = \frac{2^{31}}{\texttt{rangeSize}} \leq 2^{21}$$

$$\texttt{rangeSize} \geq \frac{2^{31}}{2^{21}}$$

$$\texttt{rangeSize} \geq 2^{10}$$

Second Pass: The Bit Vector

We need to have enough space to store `rangeSize` bits. Since we can fit 2^{23} bytes in memory, we can fit 2^{26} bits in memory. Therefore, we can conclude the following:

$$2^{11} <= \texttt{rangeSize} <= 2^{26}$$

These conditions give us a good amount of "wiggle room," but the nearer to the middle that we pick, the less memory will be used at any given time.

The below code provides one implementation for this algorithm.

```
1  int findOpenNumber(String filename) throws FileNotFoundException {
2    int rangeSize = (1 << 20); // 2^20 bits (2^17 bytes)
3
4    /* Get count of number of values within each block. */
5    int[] blocks = getCountPerBlock(filename, rangeSize);
6
7    /* Find a block with a missing value. */
8    int blockIndex = findBlockWithMissing(blocks, rangeSize);
9    if (blockIndex < 0) return -1;
```

```
10
11    /* Create bit vector for items within this range. */
12    byte[] bitVector = getBitVectorForRange(filename, blockIndex, rangeSize);
13
14    /* Find a zero in the bit vector */
15    int offset = findZero(bitVector);
16    if (offset < 0) return -1;
17
18    /* Compute missing value. */
19    return blockIndex * rangeSize + offset;
20  }
21
22  /* Get count of items within each range. */
23  int[] getCountPerBlock(String filename, int rangeSize)
24        throws FileNotFoundException {
25    int arraySize = Integer.MAX_VALUE / rangeSize + 1;
26    int[] blocks = new int[arraySize];
27
28    Scanner in = new Scanner (new FileReader(filename));
29    while (in.hasNextInt()) {
30        int value = in.nextInt();
31        blocks[value / rangeSize]++;
32    }
33    in.close();
34    return blocks;
35  }
36
37  /* Find a block whose count is low. */
38  int findBlockWithMissing(int[] blocks, int rangeSize) {
39    for (int i = 0; i < blocks.length; i++) {
40       if (blocks[i] < rangeSize){
41          return i;
42       }
43    }
44    return -1;
45  }
46
47  /* Create a bit vector for the values within a specific range. */
48  byte[] getBitVectorForRange(String filename, int blockIndex, int rangeSize)
49        throws FileNotFoundException {
50    int startRange = blockIndex * rangeSize;
51    int endRange = startRange + rangeSize;
52    byte[] bitVector = new byte[rangeSize/Byte.SIZE];
53
54    Scanner in = new Scanner(new FileReader(filename));
55    while (in.hasNextInt()) {
56       int value = in.nextInt();
57       /* If the number is inside the block that's missing numbers, we record it */
58       if (startRange <= value && value < endRange) {
59          int offset = value - startRange;
60          int mask = (1 << (offset % Byte.SIZE));
61          bitVector[offset / Byte.SIZE] |= mask;
62       }
63    }
64    in.close();
65    return bitVector;
```

```
}

/* Find bit index that is 0 within byte. */
int findZero(byte b) {
   for (int i = 0; i < Byte.SIZE; i++) {
      int mask = 1 << i;
      if ((b & mask) == 0) {
         return i;
      }
   }
   return -1;
}

/* Find a zero within the bit vector and return the index. */
int findZero(byte[] bitVector) {
   for (int i = 0; i < bitVector.length; i++) {
      if (bitVector[i] != ~0) { // If not all 1s
         int bitIndex = findZero(bitVector[i]);
         return i * Byte.SIZE + bitIndex;
      }
   }
   return -1;
}
```

What if, as a follow up question, you are asked to solve the problem with even less memory? In this case, we can do repeated passes using the approach from the first step. We'd first check to see how many integers are found within each sequence of a million elements. Then, in the second pass, we'd check how many integers are found in each sequence of a thousand elements. Finally, in the third pass, we'd apply the bit vector.

10.8 Find Duplicates: 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 151

SOLUTION

We have 4 kilobytes of memory which means we can address up to $8 * 4 * 2^{10}$ bits. Note that $32 * 2^{10}$ bits is greater than 32000. We can create a bit vector with 32000 bits, where each bit represents one integer.

Using this bit vector, we can then iterate through the array, flagging each element v by setting bit v to 1. When we come across a duplicate element, we print it.

```
void checkDuplicates(int[] array) {
   BitSet bs = new BitSet(32000);
   for (int i = 0; i < array.length; i++) {
      int num = array[i];
      int num0 = num - 1; // bitset starts at 0, numbers start at 1
      if (bs.get(num0)) {
         System.out.println(num);
      } else {
         bs.set(num0);
      }
   }
}

class BitSet {
```

```
15    int[] bitset;
16
17    public BitSet(int size) {
18       bitset = new int[(size >> 5) + 1]; // divide by 32
19    }
20
21    boolean get(int pos) {
22       int wordNumber = (pos >> 5); // divide by 32
23       int bitNumber = (pos & 0x1F); // mod 32
24       return (bitset[wordNumber] & (1 << bitNumber)) != 0;
25    }
26
27    void set(int pos) {
28       int wordNumber = (pos >> 5); // divide by 32
29       int bitNumber = (pos & 0x1F); // mod 32
30       bitset[wordNumber] |= 1 << bitNumber;
31    }
32 }
```

Note that while this isn't an especially difficult problem, it's important to implement this cleanly. This is why we defined our own bit vector class to hold a large bit vector. If our interviewer lets us (she may or may not), we could have of course used Java's built in `BitSet` class.

10.9 Sorted Matrix Search: 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 151

SOLUTION

We can approach this in two ways: a more naive solution that only takes advantage of part of the sorting, and a more optimal way that takes advantage of both parts of the sorting.

Solution #1: Naive Solution

As a first approach, we can do binary search on every row to find the element. This algorithm will be `O(M log(N))`, since there are `M` rows and it takes `O(log(N))` time to search each one. This is a good approach to mention to your interviewer before you proceed with generating a better algorithm.

To develop an algorithm, let's start with a simple example.

15	20	40	85
20	35	80	95
30	55	95	105
40	80	100	120

Suppose we are searching for the element 55. How can we identify where it is?

If we look at the start of a row or the start of a column, we can start to deduce the location. If the start of a column is greater than 55, we know that 55 can't be in that column, since the start of the column is always the minimum element. Additionally, we know that 55 can't be in any columns on the right, since the first element of each column must increase in size from left to right. Therefore, if the start of the column is greater than the element `x` that we are searching for, we know that we need to move further to the left.

For rows, we use identical logic. If the start of a row is bigger than `x`, we know we need to move upwards.

Observe that we can also make a similar conclusion by looking at the ends of columns or rows. If the end of a column or row is less than x, then we know that we must move down (for rows) or to the right (for columns) to find x. This is because the end is always the maximum element.

We can bring these observations together into a solution. The observations are the following:

- If the start of a column is greater than x, then x is to the left of the column.
- If the end of a column is less than x, then x is to the right of the column.
- If the start of a row is greater than x, then x is above that row.
- If the end of a row is less than x, then x is below that row.

We can begin in any number of places, but let's begin with looking at the starts of columns.

We need to start with the greatest column and work our way to the left. This means that our first element for comparison is `array[0][c-1]`, where c is the number of columns. By comparing the start of columns to x (which is 55), we'll find that x must be in columns 0, 1, or 2. We will have stopped at `array[0][2]`.

This element may not be the end of a row in the full matrix, but it is an end of a row of a submatrix. The same conditions apply. The value at `array[0][2]`, which is 40, is less than 55, so we know we can move downwards.

We now have a submatrix to consider that looks like the following (the gray squares have been eliminated).

15	20	40	85
20	35	80	95
30	55	95	105
40	80	100	120

We can repeatedly apply these conditions to search for 55. Note that the only conditions we actually use are conditions 1 and 4.

The code below implements this elimination algorithm.

```
1  boolean findElement(int[][] matrix, int elem) {
2     int row = 0;
3     int col = matrix[0].length - 1;
4     while (row < matrix.length && col >= 0) {
5        if (matrix[row][col] == elem) {
6           return true;
7        } else if (matrix[row][col] > elem) {
8           col--;
9        } else {
10          row++;
11       }
12    }
13    return false;
14 }
```

Alternatively, we can apply a solution that more directly looks like binary search. The code is considerably more complicated, but it applies many of the same learnings.

Solution #2: Binary Search

Let's again look at a simple example.

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

We want to be able to leverage the sorting property to more efficiently find an element. So, we might ask ourselves, what does the unique ordering property of this matrix imply about where an element might be located?

We are told that every row and column is sorted. This means that element `a[i][j]` will be greater than the elements in row `i` between columns 0 and `j - 1` and the elements in column `j` between rows 0 and `i - 1`.

Or, in other words:

```
a[i][0] <= a[i][1] <= ... <= a[i][j-1] <= a[i][j]
a[0][j] <= a[1][j] <= ... <= a[i-1][j] <= a[i][j]
```

Looking at this visually, the dark gray element below is bigger than all the light gray elements.

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

The light gray elements also have an ordering to them: each is bigger than the elements to the left of it, as well as the elements above it. So, by transitivity, the dark gray element is bigger than the entire square.

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

This means that for any rectangle we draw in the matrix, the bottom right hand corner will always be the biggest.

Likewise, the top left hand corner will always be the smallest. The colors below indicate what we know about the ordering of elements (light gray < dark gray < black):

15	20	70	85
20	35	80	95
30	55	95	105
40	80	120	120

Let's return to the original problem: suppose we were searching for the value 85. If we look along the diagonal, we'll find the elements 35 and 95. What does this tell us about the location of 85?

15	20	70	85
25	35	80	95
30	55	95	105
40	80	120	120

85 can't be in the black area, since 95 is in the upper left hand corner and is therefore the smallest element in that square.

85 can't be in the light gray area either, since 35 is in the lower right hand corner of that square.

85 must be in one of the two white areas.

So, we partition our grid into four quadrants and recursively search the lower left quadrant and the upper right quadrant. These, too, will get divided into quadrants and searched.

Observe that since the diagonal is sorted, we can efficiently search it using binary search.

The code below implements this algorithm.

```
1  Coordinate findElement(int[][] matrix, Coordinate origin, Coordinate dest, int x){
2    if (!origin.inbounds(matrix) || !dest.inbounds(matrix)) {
3      return null;
4    }
5    if (matrix[origin.row][origin.column] == x) {
6      return origin;
7    } else if (!origin.isBefore(dest)) {
8      return null;
9    }
10
11   /* Set start to start of diagonal and end to the end of the diagonal. Since the
12    * grid may not be square, the end of the diagonal may not equal dest. */
13   Coordinate start = (Coordinate) origin.clone();
14   int diagDist = Math.min(dest.row - origin.row, dest.column - origin.column);
15   Coordinate end = new Coordinate(start.row + diagDist, start.column + diagDist);
16   Coordinate p = new Coordinate(0, 0);
17
18   /* Do binary search on the diagonal, looking for the first element > x */
19   while (start.isBefore(end)) {
20     p.setToAverage(start, end);
21     if (x > matrix[p.row][p.column]) {
22       start.row = p.row + 1;
23       start.column = p.column + 1;
24     } else {
25       end.row = p.row - 1;
26       end.column = p.column - 1;
27     }
28   }
29
30   /* Split the grid into quadrants. Search the bottom left and the top right. */
31   return partitionAndSearch(matrix, origin, dest, start, x);
32 }
33
34 Coordinate partitionAndSearch(int[][] matrix, Coordinate origin, Coordinate dest,
35                               Coordinate pivot, int x) {
36   Coordinate lowerLeftOrigin = new Coordinate(pivot.row, origin.column);
37   Coordinate lowerLeftDest = new Coordinate(dest.row, pivot.column - 1);
38   Coordinate upperRightOrigin = new Coordinate(origin.row, pivot.column);
39   Coordinate upperRightDest = new Coordinate(pivot.row - 1, dest.column);
40
41   Coordinate lowerLeft = findElement(matrix, lowerLeftOrigin, lowerLeftDest, x);
42   if (lowerLeft == null) {
43     return findElement(matrix, upperRightOrigin, upperRightDest, x);
44   }
```

```
45    return lowerLeft;
46  }
47
48  Coordinate findElement(int[][] matrix, int x) {
49    Coordinate origin = new Coordinate(0, 0);
50    Coordinate dest = new Coordinate(matrix.length - 1, matrix[0].length - 1);
51    return findElement(matrix, origin, dest, x);
52  }
53
54  public class Coordinate implements Cloneable {
55    public int row, column;
56    public Coordinate(int r, int c) {
57      row = r;
58      column = c;
59    }
60
61    public boolean inbounds(int[][] matrix) {
62      return  row >= 0 && column >= 0 &&
63              row < matrix.length && column < matrix[0].length;
64    }
65
66    public boolean isBefore(Coordinate p) {
67      return row <= p.row && column <= p.column;
68    }
69
70    public Object clone() {
71      return new Coordinate(row, column);
72    }
73
74    public void setToAverage(Coordinate min, Coordinate max) {
75      row = (min.row + max.row) / 2;
76      column = (min.column + max.column) / 2;
77    }
78  }
```

If you read all this code and thought, "there's no way I could do all this in an interview!" you're probably right. You couldn't. But, your performance on any problem is evaluated compared to other candidates on the same problem. So while you couldn't implement all this, neither could they. You are at no disadvantage when you get a tricky problem like this.

You help yourself out a bit by separating code out into other methods. For example, by pulling `partitionAndSearch` out into its own method, you will have an easier time outlining key aspects of the code. You can then come back to fill in the body for `partitionAndSearch` if you have time.

10.10 Rank from Stream: 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 151

SOLUTION

A relatively easy way to implement this would be to have an array that holds all the elements in sorted order. When a new element comes in, we would need to shift the other elements to make room. Implementing `getRankOfNumber` would be quite efficient, though. We would simply perform a binary search for n, and return the index.

However, this is very inefficient for inserting elements (that is, the `track(int x)` function). We need a data structure which is good at keeping relative ordering, as well as updating when we insert new elements. A binary search tree can do just that.

Instead of inserting elements into an array, we insert elements into a binary search tree. The method `track(int x)` will run in `O(log n)` time, where n is the size of the tree (provided, of course, that the tree is balanced).

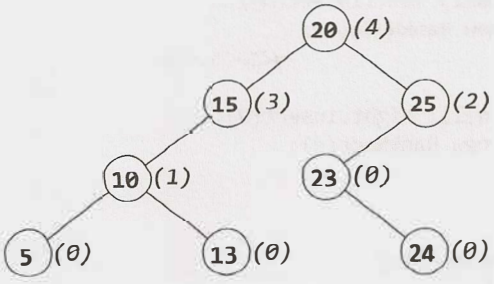
To find the rank of a number, we could do an in-order traversal, keeping a counter as we traverse. The goal is that, by the time we find x, `counter` will equal the number of elements less than x.

As long as we're moving left during searching for x, the counter won't change. Why? Because all the values we're skipping on the right side are greater than x. After all, the very smallest element (with rank of 1) is the leftmost node.

When we move to the right though, we skip over a bunch of elements on the left. All of these elements are less than x, so we'll need to increment `counter` by the number of elements in the left subtree.

Rather than counting the size of the left subtree (which would be inefficient), we can track this information as we add new elements to the tree.

Let's walk through an example on the following tree. In the below example, the value in parentheses indicates the number of nodes in the left subtree (or, in other words, the rank of the node *relative* to its subtree).



Suppose we want to find the rank of 24 in the tree above. We would compare 24 with the root, 20, and find that 24 must reside on the right. The root has 4 nodes in its left subtree, and when we include the root itself, this gives us five total nodes smaller than 24. We set `counter` to 5.

Then, we compare 24 with node 25 and find that 24 must be on the left. The value of `counter` does not update, since we're not "passing over" any smaller nodes. The value of `counter` is still 5.

Next, we compare 24 with node 23, and find that 24 must be on the right. `Counter` gets incremented by just 1 (to 6), since 23 has no left nodes.

Finally, we find 24 and we return `counter`: 6.

Recursively, the algorithm is the following:

```
int getRank(Node node, int x) {
  if x is node.data, return node.leftSize()
  if x is on left of node, return getRank(node.left, x)
  if x is on right of node, return node.leftSize() + 1 + getRank(node.right, x)
}
```

The full code for this is below.

```
RankNode root = null;

void track(int number) {
  if (root == null) {
    root = new RankNode(number);
  } else {
    root.insert(number);
  }
}

int getRankOfNumber(int number) {
  return root.getRank(number);
}


public class RankNode {
  public int left_size = 0;
  public RankNode left, right;
  public int data = 0;
  public RankNode(int d) {
    data = d;
  }

  public void insert(int d) {
    if (d <= data) {
```

```
26          if (left != null) left.insert(d);
27          else left = new RankNode(d);
28          left_size++;
29        } else {
30          if (right != null) right.insert(d);
31          else right = new RankNode(d);
32        }
33      }
34
35      public int getRank(int d) {
36        if (d == data) {
37          return left_size;
38        } else if (d < data) {
39          if (left == null) return -1;
40          else return left.getRank(d);
41        } else {
42          int right_rank = right == null ? -1 : right.getRank(d);
43          if (right_rank == -1) return -1;
44          else return left_size + 1 + right_rank;
45        }
46      }
47  }
```

The `track` method and the `getRankOfNumber` method will both operate in `O(log N)` on a balanced tree and `O(N)` on an unbalanced tree.

Note how we've handled the case in which d is not found in the tree. We check for the -1 return value, and, when we find it, return -1 up the tree. It is important that you handle cases like this.

10.11 Peaks and Valleys: In an array of integers, a "peak" is an element which is greater than or equal to the adjacent integers and a "valley" is an element which is less than or equal to the adjacent integers. For example, in the array {5, 8, 6, 2, 3, 4, 6}, {8, 6} are peaks and {5, 2} are valleys. Given an array of integers, sort the array into an alternating sequence of peaks and valleys.

EXAMPLE

Input: {5, 3, 1, 2, 3}

Output: {5, 1, 3, 2, 3}

pg 151

SOLUTION

Since this problem asks us to sort the array in a particular way, one thing we can try is doing a normal sort and then "fixing" the array into an alternating sequence of peaks and valleys.

Suboptimal Solution

Imagine we were given an unsorted array and then sort it to become the following:

```
0  1  4  7  8  9
```

We now have an ascending list of integers.

How can we rearrange this into a proper alternating sequence of peaks and valleys? Let's walk through it and try to do that.

- The `0` is okay.

- The 1 is in the wrong place. We can swap it with either the 0 or 4. Let's swap it with the 0.

  ```
  1  0  4  7  8  9
  ```

- The 4 is okay.
- The 7 is in the wrong place. We can swap it with either the 4 or the 8. Let's swap it with the 4.

  ```
  1  0  7  4  8  9
  ```

- The 9 is in the wrong place. Let's swap it with the 8.

  ```
  1  0  7  4  9  8
  ```

Observe that there's nothing special about the array having these values. The relative order of the elements matters, but all sorted arrays will have the same relative order. Therefore, we can take this same approach on any sorted array.

Before coding, we should clarify the exact algorithm, though.

1. Sort the array in ascending order.
2. Iterate through the elements, starting from index 1 (not 0) and jumping two elements at a time.
3. At each element, swap it with the previous element. Since every three elements appear in the order `small <= medium <= large`, swapping these elements will always put medium as a peak: `medium <= small <= large`.

This approach will ensure that the peaks are in the right place: indexes 1, 3, 5, and so on. As long as the odd-numbered elements (the peaks) are bigger than the adjacent elements, then the even-numbered elements (the valleys) must be smaller than the adjacent elements.

The code to implement this is below.

```
1  void sortValleyPeak(int[] array) {
2    Arrays.sort(array);
3    for (int i = 1; i < array.length; i += 2) {
4      swap(array, i - 1, i);
5    }
6  }
7
8  void swap(int[] array, int left, int right) {
9    int temp = array[left];
10   array[left] = array[right];
11   array[right] = temp;
12 }
```

This algorithm runs in `O(n log n)` time.

Optimal Solution

To optimize past the prior solution, we need to cut out the sorting step. The algorithm must operate on an unsorted array.

Let's revisit an example.

```
9  1  0  4  8  7
```

For each element, we'll look at the adjacent elements. Let's imagine some sequences. We'll just use the numbers 0, 1 and 2. The specific values don't matter.

```
0  1  2
0  2  1       // peak
1  0  2
1  2  0       // peak
2  1  0
```

```
2   0   1
```

If the center element needs to be a peak, then two of those sequences work. Can we fix the other ones to make the center element a peak?

Yes. We can fix this sequence by swapping the center element with the largest adjacent element.

```
0   1   2   ->  0   2   1
0   2   1       // peak
1   0   2   ->  1   2   0
1   2   0   // peak
2   1   0   ->  1   2   0
2   0   1   ->  0   2   1
```

As we noted before, if we make sure the peaks are in the right place then we know the valleys are in the right place.

> We should be a little cautious here. Is it possible that one of these swaps could "break" an earlier part of the sequence that we'd already processed? This is a good thing to worry about, but it's not an issue here. If we're swapping `middle` with `left`, then `left` is currently a valley. `Middle` is smaller than `left`, so we're putting an even smaller element as a valley. Nothing will break. All is good!

The code to implement this is below.

```
1   void sortValleyPeak(int[] array) {
2     for (int i = 1; i < array.length; i += 2) {
3       int biggestIndex = maxIndex(array, i - 1, i, i + 1);
4       if (i != biggestIndex) {
5         swap(array, i, biggestIndex);
6       }
7     }
8   }
9
10  int maxIndex(int[] array, int a, int b, int c) {
11    int len = array.length;
12    int aValue = a >= 0 && a < len ? array[a] : Integer.MIN_VALUE;
13    int bValue = b >= 0 && b < len ? array[b] : Integer.MIN_VALUE;
14    int cValue = c >= 0 && c < len ? array[c] : Integer.MIN_VALUE;
15
16    int max = Math.max(aValue, Math.max(bValue, cValue));
17    if (aValue == max) return a;
18    else if (bValue == max) return b;
19    else return c;
20  }
```

This algorithm takes O(n) time.

11

Solutions to Testing

11.1 **Mistake:** Find the mistake(s) in the following code:

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

pg 157

SOLUTION

There are two mistakes in this code.

First, note that an `unsigned int` is, by definition, always greater than or equal to zero. The for loop condition will therefore always be true, and it will loop infinitely.

The correct code to print all numbers from 100 to 1, is `i > 0`. If we truly wanted to print zero, we could add an additional `printf` statement after the for loop.

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

One additional correction is to use %u in place of %d, as we are printing `unsigned int`.

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

This code will now correctly print the list of all numbers from 100 to 1, in descending order.

11.2 **Random Crashes:** 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 157

SOLUTION

The question largely depends on the type of application being diagnosed. However, we can give some general causes of random crashes.

1. *"Random Variable:"* The application may use some random number or variable component that may not be fixed for every execution of the program. Examples include user input, a random number generated by the program, or the time of day.

2. *Uninitialized Variable:* The application could have an uninitialized variable which, in some languages, may cause it to take on an arbitrary value. The values of this variable could result in the code taking a slightly different path each time.
3. *Memory Leak:* The program may have run out of memory. Other culprits are totally random for each run since it depends on the number of processes running at that particular time. This also includes heap overflow or corruption of data on the stack.
4. *External Dependencies:* The program may depend on another application, machine, or resource. If there are multiple dependencies, the program could crash at any point.

To track down the issue, we should start with learning as much as possible about the application. Who is running it? What are they doing with it? What kind of application is it?

Additionally, although the application doesn't crash in exactly the same place, it's possible that it is linked to specific components or scenarios. For example, it could be that the application never crashes if it's simply launched and left untouched, and that crashes only appear at some point after loading a file. Or, it may be that all the crashes take place within the lower level components, such as file I/O.

It may be useful to approach this by elimination. Close down all other applications on the system. Track resource use very carefully. If there are parts of the program we can disable, do so. Run it on a different machine and see if we experience the same issue. The more we can eliminate (or change), the easier we can track down the issue.

Additionally, we may be able to use tools to check for specific situations. For example, to investigate issue #2, we can utilize runtime tools which check for uninitialized variables.

These problems are as much about your brainstorming ability as they are about your approach. Do you jump all over the place, shouting out random suggestions? Or do you approach it in a logical, structured manner? Hopefully, it's the latter.

11.3 Chess Test: 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 157

SOLUTION

In this problem, there are two primary types of testing: extreme case validation (ensuring that the program doesn't crash on bad input), and general case testing. We'll start with the first type.

Testing Type #1: Extreme Case Validation

We need to ensure that the program handles bad or unusual input gracefully. This means checking the following conditions:

- Test with negative numbers for x and y
- Test with x larger than the width
- Test with y larger than the height
- Test with a completely full board
- Test with an empty or nearly empty board
- Test with far more white pieces than black

- Test with far more black pieces than white

For the error cases above, we should ask our interviewer whether we want to return false or throw an exception, and we should test accordingly.

Testing Type #2: General Testing:

General testing is much more expansive. Ideally, we would test every possible board, but there are far too many boards. We can, however, perform a reasonable coverage of different boards.

There are 6 pieces in chess, so we can test each piece against every other piece, in every possible direction. This would look something like the below code:

```
1  foreach piece a:
2    for each other type of piece b (6 types + empty space)
3      foreach direction d
4        Create a board with piece a.
5        Place piece b in direction d.
6        Try to move - check return value.
```

The key to this problem is recognizing that we can't test every possible scenario, even if we would like to. So, instead, we must focus on the essential areas.

11.4 No Test Tools: How would you load test a webpage without using any test tools?

pg 157

SOLUTION

Load testing helps to identify a web application's maximum operating capacity, as well as any bottlenecks that may interfere with its performance. Similarly, it can check how an application responds to variations in load.

To perform load testing, we must first identify the performance critical scenarios and the metrics which fulfill our performance objectives. Typical criteria include:

- Response time
- Throughput
- Resource utilization
- Maximum load that the system can bear.

Then, we design tests to simulate the load, taking care to measure each of these criteria.

In the absence of formal testing tools, we can basically create our own. For example, we could simulate concurrent users by creating thousands of virtual users. We would write a multi-threaded program with thousands of threads, where each thread acts as a real-world user loading the page. For each user, we would programmatically measure response time, data I/O, etc.

We would then analyze the results based on the data gathered during the tests and compare it with the accepted values.

11.5 Test a Pen: How would you test a pen?

pg 157

SOLUTION

This problem is largely about understanding the constraints and approaching the problem in a structured manner.

To understand the constraints, you should ask a lot of questions to understand the "who, what, where, when, how and why" of a problem (or as many of those as apply to the problem). Remember that a good tester understands exactly what he is testing before starting the work.

To illustrate the technique in this problem, let us guide you through a mock conversation.

- **Interviewer:** How would you test a pen?
- **Candidate:** Let me find out a bit about the pen. Who is going to use the pen?
- **Interviewer:** Probably children.
- **Candidate:** Okay, that's interesting. What will they be doing with it? Will they be writing, drawing, or doing something else with it?
- **Interviewer:** Drawing.
- **Candidate:** Okay, great. On what? Paper? Clothing? Walls?
- **Interviewer:** On clothing.
- **Candidate:** Great. What kind of tip does the pen have? Felt? Ballpoint? Is it intended to wash off, or is it intended to be permanent?
- **Interviewer:** It's intended to wash off.

Many questions later, you may get to this:

- **Candidate:** Okay, so as I understand it, we have a pen that is being targeted at 5 to 10-year-olds. The pen has a felt tip and comes in red, green, blue and black. It's intended to wash off when clothing is washed. Is that correct?

The candidate now has a problem that is significantly different from what it initially seemed to be. This is not uncommon. In fact, many interviewers intentionally give a problem that seems clear (everyone knows what a pen is!), only to let you discover that it's quite a different problem from what it seemed. Their belief is that users do the same thing, though users do so accidentally.

Now that you understand what you're testing, it's time to come up with a plan of attack. The key here is *structure*.

Consider what the different components of the object or problem, and go from there. In this case, the components might be:

- *Fact check:* Verify that the pen is felt tip and that the ink is one of the allowed colors.
- *Intended use:* Drawing. Does the pen write properly on clothing?
- *Intended use:* Washing. Does it wash off of clothing (even if it's been there for an extended period of time)? Does it wash off in hot, warm and cold water?
- *Safety:* Is the pen safe (non-toxic) for children?
- *Unintended uses:* How else might children use the pen? They might write on other surfaces, so you need to check whether the behavior there is correct. They might also stomp on the pen, throw it, and so on.

You'll need to make sure that the pen holds up under these conditions.

Remember that in any testing question, you need to test both the intended and unintended scenarios. People don't always use the product the way you want them to.

11.6 Test an ATM: How would you test an ATM in a distributed banking system?

pg 157

SOLUTION

The first thing to do on this question is to clarify assumptions. Ask the following questions:

- Who is going to use the ATM? Answers might be "anyone," or it might be "blind people," or any number of other answers.
- What are they going to use it for? Answers might be "withdrawing money," "transferring money," "checking their balance," or many other answers.
- What tools do we have to test? Do we have access to the code, or just to the ATM?

Remember: a good tester makes sure she knows what she's testing!

Once we understand what the system looks like, we'll want to break down the problem into different testable components. These components include:

- Logging in
- Withdrawing money
- Depositing money
- Checking balance
- Transferring money

We would probably want to use a mix of manual and automated testing.

Manual testing would involve going through the steps above, making sure to check for all the error cases (low balance, new account, nonexistent account, and so on).

Automated testing is a bit more complex. We'll want to automate all the standard scenarios, as shown above, and we also want to look for some very specific issues, such as race conditions. Ideally, we would be able to set up a closed system with fake accounts and ensure that, even if someone withdraws and deposits money rapidly from different locations, he never gets money or loses money that he shouldn't.

Above all, we need to prioritize security and reliability. People's accounts must always be protected, and we must make sure that money is always properly accounted for. No one wants to unexpectedly lose money! A good tester understands the system priorities.

12

Solutions to C and C++

12.1 Last K Lines: Write a method to print the last K lines of an input file using C++.

pg 163

SOLUTION

One brute force way could be to count the number of lines (N) and then print from N-K to Nth line. But this requires two reads of the file, which is unnecessarily costly. We need a solution which allows us to read just once and be able to print the last K lines.

We can allocate an array for all K lines and the last K lines we've read in the array. , and so on. Each time that we read a new line, we purge the oldest line from the array.

But—you might ask—wouldn't this require shifting elements in the array, which is also very expensive? No, not if we do it correctly. Instead of shifting the array each time, we will use a circular array.

With a circular array, we always replace the oldest item when we read a new line. The oldest item is tracked in a separate variable, which adjusts as we add new items.

The following is an example of a circular array:

```
step 1 (initially): array = {a, b, c, d, e, f}. p = 0
step 2 (insert g):  array = {g, b, c, d, e, f}. p = 1
step 3 (insert h):  array = {g, h, c, d, e, f}. p = 2
step 4 (insert i):  array = {g, h, i, d, e, f}. p = 3
```

The code below implements this algorithm.

```
1  void printLast10Lines(char* fileName) {
2    const int K = 10;
3    ifstream file (fileName);
4    string L[K];
5    int size = 0;
6
7    /* read file line by line into circular array */
8    /* peek() so an EOF following a line ending is not considered a separate line */
9    while (file.peek() != EOF) {
10     getline(file, L[size % K]);
11     size++;
12   }
13
14   /* compute start of circular array, and the size of it */
15   int start = size > K ? (size % K) : 0;
16   int count = min(K, size);
17
```

```
    /* print elements in the order they were read */
    for (int i = 0; i < count; i++) {
       cout << L[(start + i) % K] << endl;
    }
}
```

This solution will require reading in the whole file, but only ten lines will be in memory at any given point.

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

pg 163

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.

```
void reverse(char *str) {
   char* end = str;
   char tmp;
   if (str) {
      while (*end) { /* find end of the string */
         ++end;
      }
      --end; /* set one char back, since last char is null */

      /* swap characters from start of string with the end of the string, until the
       * pointers meet in middle. */
      while (str < end) {
         tmp = *str;
         *str++ = *end;
         *end-- = tmp;
      }
   }
}
```

This is just one of many ways to implement this solution. We could even implement this code recursively (but we wouldn't recommend it).

12.3 Hash Table vs STL Map: 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 163

SOLUTION

In a hash table, a value is stored by calling a hash function on a key. Values are not stored in sorted order. Additionally, since hash tables use the key to find the index that will store the value, an insert or lookup can be done in amortized `O(1)` time (assuming few collisions in the hash table). In a hash table, one must also handle potential collisions. This is often done by chaining, which means to create a linked list of all the values whose keys map to a particular index.

An STL map inserts the key/value pairs into a binary search tree based on the keys. There is no need to handle collisions, and, since the tree is balanced, the insert and lookup time is guaranteed to be `O(log N)`.

How is a hash table implemented?

A hash table is traditionally implemented with an array of linked lists. When we want to insert a key/value pair, we map the key to an index in the array using a hash function. The value is then inserted into the linked list at that position.

Note that the elements in a linked list at a particular index of the array do not have the same key. Rather, `hashFunction(key)` is the same for these values. Therefore, in order to retrieve the value for a specific key, we need to store in each node both the exact key and the value.

To summarize, the hash table will be implemented with an array of linked lists, where each node in the linked list holds two pieces of data: the value and the original key. In addition, we will want to note the following design criteria:

1. We want to use a good hash function to ensure that the keys are well distributed. If they are not well distributed, then we would get a lot of collisions and the speed to find an element would decline.
2. No matter how good our hash function is, we will still have collisions, so we need a method for handling them. This often means chaining via a linked list, but it's not the only way.
3. We may also wish to implement methods to dynamically increase or decrease the hash table size depending on capacity. For example, when the ratio of the number of elements to the table size exceeds a certain threshold, we may wish to increase the hash table size. This would mean creating a new hash table and transferring the entries from the old table to the new table. Because this is an expensive operation, we want to be careful to not do it too often.

What can be used instead of a hash table, if the number of inputs is small?

You can use an STL map or a binary tree. Although this takes `O(log(n))` time, the number of inputs may be small enough to make this time negligible.

12.4 **Virtual Functions:** How do virtual functions work in C++?

pg 164

SOLUTION

A virtual function depends on a "vtable" or "Virtual Table." If any function of a class is declared to be virtual, a vtable is constructed which stores addresses of the virtual functions of this class. The compiler also adds a hidden `vptr` variable in all such classes which points to the vtable of that class. If a virtual function is not overridden in the derived class, the vtable of the derived class stores the address of the function in its parent class. The vtable is used to resolve the address of the function when the virtual function is called. Dynamic binding in C++ is performed through the vtable mechanism.

Thus, when we assign the derived class object to the base class pointer, the `vptr` variable points to the vtable of the derived class. This assignment ensures that the most derived virtual function gets called.

Consider the following code.

```
1  class Shape {
2    public:
3     int edge_length;
4     virtual int circumference () {
```

```
5       cout << "Circumference of Base Class\n";
6       return 0;
7     }
8   };
9
10  class Triangle: public Shape {
11    public:
12    int circumference () {
13       cout<< "Circumference of Triangle Class\n";
14       return 3 * edge_length;
15    }
16  };
17
18  void main() {
19    Shape * x = new Shape();
20    x->circumference(); // "Circumference of Base Class"
21    Shape *y = new Triangle();
22    y->circumference(); // "Circumference of Triangle Class"
23  }
```

In the previous example, `circumference` is a virtual function in the Shape class, so it becomes virtual in each of the derived classes (`Triangle`, etc). C++ non-virtual function calls are resolved at compile time with static binding, while virtual function calls are resolved at runtime with dynamic binding.

12.5 Shallow vs Deep Copy: What is the difference between deep copy and shallow copy? Explain how you would use each.

pg 164

SOLUTION

A shallow copy copies all the member values from one object to another. A deep copy does all this and also deep copies any pointer objects.

An example of shallow and deep copy is below.

```
1   struct Test {
2     char * ptr;
3   };
4
5   void shallow_copy(Test & src, Test & dest) {
6     dest.ptr = src.ptr;
7   }
8
9   void deep_copy(Test & src, Test & dest) {
10    dest.ptr = (char*)malloc(strlen(src.ptr) + 1);
11    strcpy(dest.ptr, src.ptr);
12  }
```

Note that `shallow_copy` may cause a lot of programming runtime errors, especially with the creation and deletion of objects. Shallow copy should be used very carefully and only when a programmer really understands what he wants to do. In most cases, shallow copy is used when there is a need to pass information about a complex structure without actual duplication of data. One must also be careful with destruction of objects in a shallow copy.

In real life, shallow copy is rarely used. Deep copy should be used in most cases, especially when the size of the copied structure is small.

12.6 Volatile: What is the significance of the keyword "volatile" in C?

pg 164

SOLUTION

The keyword `volatile` informs the compiler that the value of variable it is applied to can change from the outside, without any update done by the code. This may be done by the operating system, the hardware, or another thread. Because the value can change unexpectedly, the compiler will therefore reload the value each time from memory.

A volatile integer can be declared by either of the following statements:

```
int volatile x;
volatile int x;
```

To declare a pointer to a volatile integer, we do the following:

```
volatile int * x;
int volatile * x;
```

A volatile pointer to non-volatile data is rare, but can be done.

```
int * volatile x;
```

If you wanted to declare a volatile variable pointer for volatile memory (both pointer address and memory contained are volatile), you would do the following:

```
int volatile * volatile x;
```

Volatile variables are not optimized, which can be very useful. Imagine this function:

```
1  int opt = 1;
2  void Fn(void) {
3    start:
4      if (opt == 1) goto start;
5      else break;
6  }
```

At first glance, our code appears to loop infinitely. The compiler may try to optimize it to:

```
1  void Fn(void) {
2    start:
3      int opt = 1;
4      if (true)
5      goto start;
6  }
```

This becomes an infinite loop. However, an external operation might write '0' to the location of variable `opt`, thus breaking the loop.

To prevent the compiler from performing such optimization, we want to signal that another element of the system could change the variable. We do this using the `volatile` keyword, as shown below.

```
1  volatile int opt = 1;
2  void Fn(void) {
3    start:
4      if (opt == 1) goto start;
5      else break;
6  }
```

Volatile variables are also useful when multi-threaded programs have global variables and any thread can modify these shared variables. We may not want optimization on these variables.

12.7 *Virtual Base Class:* Why does a destructor in base class need to be declared virtual?

pg 164

SOLUTION

Let's think about why we have virtual methods to start with. Suppose we have the following code:

```
1  class Foo {
2   public:
3     void f();
4  };
5
6  class Bar : public Foo {
7   public:
8     void f();
9  }
10
11 Foo * p = new Bar();
12 p->f();
```

Calling `p->f()` will result in a call to `Foo::f()`. This is because `p` is a pointer to `Foo`, and `f()` is not virtual.

To ensure that `p->f()` will invoke the most derived implementation of `f()`, we need to declare `f()` to be a virtual function.

Now, let's go back to our destructor. Destructors are used to clean up memory and resources. If Foo's destructor were not virtual, then Foo's destructor would be called, even when `p` is *really* of type `Bar`.

This is why we declare destructors to be virtual; we want to ensure that the destructor for the most derived class is called.

12.8 Copy Node: 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 164

SOLUTION

The algorithm will maintain a mapping from a node address in the original structure to the corresponding node in the new structure. This mapping will allow us to discover previously copied nodes during a traditional depth-first traversal of the structure. Traversals often mark visited nodes—the mark can take many forms and does not necessarily need to be stored in the node.

Thus, we have a simple recursive algorithm:

```
1  typedef map<Node*, Node*> NodeMap;
2
3  Node * copy_recursive(Node * cur, NodeMap & nodeMap) {
4    if (cur == NULL) {
5      return NULL;
6    }
7
8    NodeMap::iterator i = nodeMap.find(cur);
9    if (i != nodeMap.end()) {
10     // we've been here before, return the copy
11     return i->second;
12   }
```

```
13
14    Node * node = new Node;
15    nodeMap[cur] = node; // map current before traversing links
16    node->ptr1 = copy_recursive(cur->ptr1, nodeMap);
17    node->ptr2 = copy_recursive(cur->ptr2, nodeMap);
18    return node;
19  }
20
21  Node * copy_structure(Node * root) {
22    NodeMap nodeMap; // we will need an empty map
23    return copy_recursive(root, nodeMap);
24  }
```

12.9 Smart Pointer: 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 164

SOLUTION

A smart pointer is the same as a normal pointer, but it provides safety via automatic memory management. It avoids issues like dangling pointers, memory leaks and allocation failures. The smart pointer must maintain a single reference count for all references to a given object.

This is one of those problems that seems at first glance pretty overwhelming, especially if you're not a C++ expert. One useful way to approach the problem is to divide the problem into two parts: (1) outline the pseudocode and approach and then (2) implement the detailed code.

In terms of the approach, we need a reference count variable that is incremented when we add a new reference to the object and decremented when we remove a reference. The code should look something like the below pseudocode:

```
1   template <class T> class SmartPointer {
2     /* The smart pointer class needs pointers to both the object itself and to the
3      * ref count. These must be pointers, rather than the actual object or ref count
4      * value, since the goal of a smart pointer is that the reference count is
5      * tracked across multiple smart pointers to one object. */
6     T * obj;
7     unsigned * ref_count;
8   }
```

We know we need constructors and a single destructor for this class, so let's add those first.

```
1   SmartPointer(T * object) {
2     /* We want to set the value of T * obj, and set the reference counter to 1. */
3   }
4
5   SmartPointer(SmartPointer<T>& sptr) {
6     /* This constructor creates a new smart pointer that points to an existing
7      * object. We will need to first set obj and ref_count to pointer to sptr's obj
8      * and ref_count. Then, because we created a new reference to obj, we need to
9      * increment ref_count. */
10  }
11
12  ~SmartPointer(SmartPointer<T> sptr) {
13    /* We are destroying a reference to the object. Decrement ref_count. If
```

```
     * ref_count is 0, then free the memory created by the integer and destroy the
     * object. */
}
```

There's one additional way that references can be created: by setting one `SmartPointer` equal to another. We'll want to override the `equal` operator to handle this, but for now, let's sketch the code like this.

```
onSetEquals(SmartPoint<T> ptr1, SmartPoint<T> ptr2) {
   /* If ptr1 has an existing value, decrement its reference count. Then, copy the
    * pointers to obj and ref_count over. Finally, since we created a new
    * reference, we need to increment ref_count. */
}
```

Getting just the approach, even without filling in the complicated C++ syntax, would count for a lot. Finishing out the code is now just a matter of filling the details.

```
template <class T> class SmartPointer {
 public:
   SmartPointer(T * ptr) {
      ref = ptr;
      ref_count = (unsigned*)malloc(sizeof(unsigned));
      *ref_count = 1;
   }

   SmartPointer(SmartPointer<T> & sptr) {
      ref = sptr.ref;
      ref_count = sptr.ref_count;
      ++(*ref_count);
   }

   /* Override the equal operator, so that when you set one smart pointer equal to
    * another the old smart pointer has its reference count decremented and the new
    * smart pointer has its reference count incrememented. */
   SmartPointer<T> & operator=(SmartPointer<T> & sptr) {
      if (this == &sptr) return *this;

      /* If already assigned to an object, remove one reference. */
      if (*ref_count > 0) {
         remove();
      }

      ref = sptr.ref;
      ref_count = sptr.ref_count;
      ++(*ref_count);
      return *this;
   }

   ~SmartPointer() {
      remove(); // Remove one reference to object.
   }

   T getValue() {
      return *ref;
   }

 protected:
   void remove() {
      --(*ref_count);
      if (*ref_count == 0) {
```

```
44          delete ref;
45          free(ref_count);
46          ref = NULL;
47          ref_count = NULL;
48        }
49    }
50
51    T * ref;
52    unsigned * ref_count;
53  };
```

The code for this problem is complicated, and you probably wouldn't be expected to complete it flawlessly.

12.10 Malloc: 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 164

SOLUTION

Typically, with `malloc`, we do not have control over where the memory is allocated within the heap. We just get a pointer to a block of memory which could start at any memory address within the heap.

We need to work with these constraints by requesting enough memory that we can return a memory address which is divisible by the desired value.

Suppose we are requesting a 100-byte chunk of memory, and we want it to start at a memory address that is a multiple of 16. How much extra memory would we need to allocate to ensure that we can do so? We would need to allocate an extra 15 bytes. With these 15 bytes, plus another 100 bytes right after that sequence, we know that we would have a memory address divisible by 16 with space for 100 bytes.

We could then do something like:

```
1  void* aligned_malloc(size_t required_bytes, size_t alignment) {
2    int offset = alignment - 1;
3    void* p = (void*) malloc(required_bytes + offset);
4    void* q = (void*) (((size_t)(p) + offset) & ~(alignment - 1));
5    return q;
6  }
```

Line 4 is a bit tricky, so let's discuss it. Suppose `alignment` is 16. We know that one of the first 16 memory address in the block at p must be divisible by 16. With `(p + 15) & 11...10000` we advance as need to this address. ANDing the last four bits of `p + 15` with `0000` guarantees that this new value will be divisible by 16 (either at the original p or in one of the following 15 addresses).

This solution is *almost* perfect, except for one big issue: how do we free the memory?

We've allocated an extra 15 bytes, in the above example, and we need to free them when we free the "real" memory.

We can do this by storing, in this "extra" memory, the address of where the full memory block begins. We will store this immediately before the aligned memory block. Of course, this means that we now need to allocate even *more* extra memory to ensure that we have enough space to store this pointer.

Therefore, to guarantee both an aligned address and space for this pointer, we will need to allocate an additional `alignment - 1 + sizeof(void*)` bytes.

The code below implements this approach.

```
1  void* aligned_malloc(size_t required_bytes, size_t alignment) {
2    void* p1; // initial block
3    void* p2; // aligned block inside initial block
4    int offset = alignment - 1 + sizeof(void*);
5    if ((p1 = (void*)malloc(required_bytes + offset)) == NULL) {
6      return NULL;
7    }
8    p2 = (void*)(((size_t)(p1) + offset) & ~(alignment - 1));
9    ((void **)p2)[-1] = p1;
10   return p2;
11 }
12
13 void aligned_free(void *p2) {
14   /* for consistency, we use the same names as aligned_malloc*/
15   void* p1 = ((void**)p2)[-1];
16   free(p1);
17 }
```

Let's look at the pointer arithmetic in lines 9 and 15. If we treat p2 as a `void**` (or an array of `void*`'s), we can just look at the index `-1` to retrieve p1.

In `aligned_free`, we take p2 as the same p2 returned from `aligned_malloc`. As before, we know that the value of p1 (which points to the beginning of the full memory block) was stored just before p2. By freeing p1, we deallocate the whole memory block.

12.11 2D Alloc: 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 164

SOLUTION

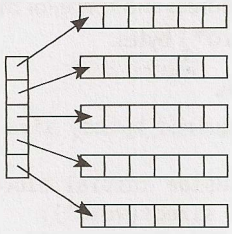
As you may know, a two-dimensional array is essentially an array of arrays. Since we use pointers with arrays, we can use double pointers to create a double array.

The basic idea is to create a one-dimensional array of pointers. Then, for each array index, we create a new one-dimensional array. This gives us a two-dimensional array that can be accessed via array indices.

The code below implements this.

```
1  int** my2DAlloc(int rows, int cols) {
2    int** rowptr;
3    int i;
4    rowptr = (int**) malloc(rows * sizeof(int*));
5    for (i = 0; i < rows; i++)  {
6      rowptr[i] = (int*) malloc(cols * sizeof(int));
7    }
8    return rowptr;
9  }
```

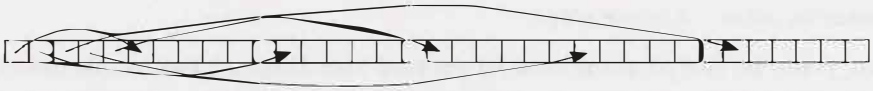
Observe how, in the above code, we've told `rowptr` where exactly each index should point. The following diagram represents how this memory is allocated.



To free this memory, we cannot simply call `free` on `rowptr`. We need to make sure to free not only the memory from the first `malloc` call, but also each subsequent call.

```
1  void my2DDealloc(int** rowptr, int rows) {
2    for (i = 0; i < rows; i++) {
3      free(rowptr[i]);
4    }
5    free(rowptr);
6  }
```

Rather than allocating the memory in many different blocks (one block for each row, plus one block to specify *where* each row is located), we can allocate this in a consecutive block of memory. Conceptually, for a two-dimensional array with five rows and six columns, this would look like the following.



If it seems strange to view the 2D array like this (and it probably does), remember that this is fundamentally no different than the first diagram. The only difference is that the memory is in a contiguous block, so our first five (in this example) elements point elsewhere in the same block of memory.

To implement this solution, we do the following.

```
1  int** my2DAlloc(int rows, int cols) {
2    int i;
3    int header = rows * sizeof(int*);
4    int data = rows * cols * sizeof(int);
5    int** rowptr = (int**)malloc(header + data);
6    if (rowptr == NULL) return NULL;
7
8    int* buf = (int*) (rowptr + rows);
9    for (i = 0; i < rows; i++) {
10     rowptr[i] = buf + i * cols;
11   }
12   return rowptr;
13 }
```

You should carefully observe what is happening on lines 11 through 13. If there are five rows of six columns each, `array[0]` will point to `array[5]`, `array[1]` will point to `array[11]`, and so on.

Then, when we actually call `array[1][3]`, the computer looks up `array[1]`, which is a pointer to another spot in memory—specifically, a pointer to `array[5]`. This element is treated as its own array, and we then get the third (zero-indexed) element from it.

Constructing the array in a single call to `malloc` has the added benefit of allowing disposal of the array with a single `free` call rather than using a special function to free the remaining data blocks.

13

Solutions to Java

13.1 Private Constructor: In terms of inheritance, what is the effect of keeping a constructor private?

pg 167

SOLUTION

Declaring a constructor `private` on class A means that you can only access the (private) constructor if you could also access A's private methods. Who, other than A, can access A's private methods and constructor? A's inner classes can. Additionally, if A is an inner class of Q, then Q's other inner classes can.

This has direct implications for inheritance, since a subclass calls its parent's constructor. The class A can be inherited, but only by its own or its parent's inner classes.

13.2 Return from Finally: In Java, does the finally block get executed if we insert a return statement inside the try block of a try-catch-finally?

pg 167

SOLUTION

Yes, it will get executed. The `finally` block gets executed when the `try` block exits. Even when we attempt to exit within the `try` block (via a `return` statement, a `continue` statement, a `break` statement or any exception), the `finally` block will still be executed.

Note that there are some cases in which the `finally` block will not get executed, such as the following:

- If the virtual machine exits during `try/catch` block execution.
- If the thread which is executing during the `try/catch` block gets killed.

13.3 Final, etc.: What is the difference between `final`, `finally`, and `finalize`?

pg 167

SOLUTIONS

Despite their similar sounding names, `final`, `finally` and `finalize` have very different purposes. To speak in very general terms, `final` is used to control whether a variable, method, or class is "changeable." The `finally` keyword is used in a `try/ catch` block to ensure that a segment of code is always executed. The `finalize()` method is called by the garbage collector once it determines that no more references exist.

Further detail on these keywords and methods is provided below.

final

The final statement has a different meaning depending on its context.

- When applied to a variable (primitive): The value of the variable cannot change.
- When applied to a variable (reference): The reference variable cannot point to any other object on the heap.
- When applied to a method: The method cannot be overridden.
- When applied to a class: The class cannot be subclassed.

finally keyword

There is an optional `finally` block after the `try` block or after the `catch` block. Statements in the `finally` block will always be executed, even if an exception is thrown (except if Java Virtual Machine exits from the `try` block). The `finally` block is often used to write the clean-up code. It 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()

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   }
```

13.4 Generics vs. Templates: Explain the difference between templates in C++ and generics in Java.

pg 167

SOLUTION

Many programmers consider templates and generics to be essentially equivalent because both allow you to do something like `List<String>`. But, *how* each language does this, and *why*, varies significantly.

The implementation of Java generics is rooted in an idea of "type erasure." This technique eliminates the parameterized types when source code is translated to the Java Virtual Machine (JVM) byte code.

For example, suppose you have the Java code below:

```
1   Vector<String>  vector = new Vector<String>();
2   vector.add(new String("hello"));
3   String  str = vector.get(0);
```

During compilation, this code is re-written into:

```
1   Vector vector = new Vector();
2   vector.add(new String("hello"));
3   String  str = (String) vector.get(0);
```

The use of Java generics didn't really change much about our capabilities; it just made things a bit prettier. For this reason, Java generics are sometimes called "syntactic sugar."

This is quite different from C++. In C++, templates are essentially a glorified macro set, with the compiler creating a new copy of the template code for each type. Proof of this is in the fact that an instance of `MyClass<Foo>` will not share a static variable with `MyClass<Bar>`. Two instances of `MyClass<Foo>`, however, will share a static variable.

To illustrate this, consider the code below:

```
1   /*** MyClass.h ***/
2   template<class T> class MyClass {
3    public:
4     static int val;
5     MyClass(int v) { val = v; }
6   };
7
```

```
/*** MyClass.cpp ***/
template<typename T>
int MyClass<T>::bar;

template class MyClass<Foo>;
template class MyClass<Bar>;

/*** main.cpp ***/
MyClass<Foo> * foo1 = new MyClass<Foo>(10);
MyClass<Foo> * foo2 = new MyClass<Foo>(15);
MyClass<Bar> * bar1 = new MyClass<Bar>(20);
MyClass<Bar> * bar2 = new MyClass<Bar>(35);

int f1 = foo1->val; // will equal 15
int f2 = foo2->val; // will equal 15
int b1 = bar1->val; // will equal 35
int b2 = bar2->val; // will equal 35
```

In Java, static variables are shared across instances of `MyClass`, regardless of the different type parameters.

Java generics and C++ templates have a number of other differences. These include:

- C++ templates can use primitive types, like `int`. Java cannot and must instead use `Integer`.
- In Java, you can restrict the template's type parameters to be of a certain type. For instance, you might use generics to implement a `CardDeck` and specify that the type parameter must extend from `CardGame`.
- In C++, the type parameter can be instantiated, whereas Java does not support this.
- In Java, the type parameter (i.e., the `Foo` in `MyClass<Foo>`) cannot be used for static methods and variables, since these would be shared between `MyClass<Foo>` and `MyClass<Bar>`. In C++, these classes are different, so the type parameter can be used for static methods and variables.
- In Java, all instances of `MyClass`, regardless of their type parameters, are the same type. The type parameters are erased at runtime. In C++, instances with different type parameters are different types.

Remember: Although Java generics and C++ templates look the same in many ways, they are very different.

13.5 TreeMap, HashMap, LinkedHashMap: Explain the differences between `TreeMap`, `HashMap`, and `LinkedHashMap`. Provide an example of when each one would be best.

pg 167

SOLUTION

All offer a key->value map and a way to iterate through the keys. The most important distinction between these classes is the time guarantees and the ordering of the keys.

- `HashMap` offers `O(1)` lookup and insertion. If you iterate through the keys, though, the ordering of the keys is essentially arbitrary. It is implemented by an array of linked lists.
- `TreeMap` offers `O(log N)` lookup and insertion. Keys are ordered, so if you need to iterate through the keys in sorted order, you can. This means that keys must implement the `Comparable` interface. `TreeMap` is implemented by a Red-Black Tree.
- `LinkedHashMap` offers `O(1)` lookup and insertion. Keys are ordered by their insertion order. It is implemented by doubly-linked buckets.

Imagine you passed an empty `TreeMap`, `HashMap`, and `LinkedHashMap` into the following function:

```
void insertAndPrint(AbstractMap<Integer, String> map) {
  int[] array = {1, -1, 0};
  for (int x : array) {
    map.put(x, Integer.toString(x));
  }

  for (int k : map.keySet()) {
    System.out.print(k + ", ");
  }
}
```

The output for each will look like the results below.

HashMap	LinkedHashMap	TreeMap
(any ordering)	{1, -1, 0}	{-1, 0, 1}

Very important: The output of `LinkedHashMap` and `TreeMap` must look like the above. For `HashMap`, the output was, in my own tests, `{0, 1, -1}`, but it could be any ordering. There is no *guarantee* on the ordering.

When might you need ordering in real life?

- Suppose you were creating a mapping of names to `Person` objects. You might want to periodically output the people in alphabetical order by name. A `TreeMap` lets you do this.
- A `TreeMap` also offers a way to, given a name, output the next 10 people. This could be useful for a "More" function in many applications.
- A `LinkedHashMap` is useful whenever you need the ordering of keys to match the ordering of insertion. This might be useful in a caching situation, when you want to delete the oldest item.

Generally, unless there is a reason not to, you would use `HashMap`. That is, if you need to get the keys back in insertion order, then use `LinkedHashMap`. If you need to get the keys back in their true/natural order, then use `TreeMap`. Otherwise, `HashMap` is probably best. It is typically faster and requires less overhead.

13.6 Object Reflection: Explain what object reflection is in Java and why it is useful.

pg 168

SOLUTION

Object Reflection is a feature in Java that provides a way to get reflective information about Java classes and objects, and perform operations such as:

1. Getting information about the methods and fields present inside the class at runtime.
2. Creating a new instance of a class.
3. Getting and setting the object fields directly by getting field reference, regardless of what the access modifier is.

The code below offers an example of object reflection.

```
/* Parameters */
Object[] doubleArgs = new Object[] { 4.2, 3.9 };

/* Get class */
Class rectangleDefinition = Class.forName("MyProj.Rectangle");

```

```
/* Equivalent: Rectangle rectangle = new Rectangle(4.2, 3.9); */
Class[] doubleArgsClass = new Class[] {double.class, double.class};
Constructor doubleArgsConstructor =
   rectangleDefinition.getConstructor(doubleArgsClass);
Rectangle rectangle = (Rectangle) doubleArgsConstructor.newInstance(doubleArgs);

/* Equivalent: Double area = rectangle.area(); */
Method m = rectangleDefinition.getDeclaredMethod("area");
Double area = (Double) m.invoke(rectangle);
```

This code does the equivalent of:

```
Rectangle rectangle = new Rectangle(4.2, 3.9);
Double area = rectangle.area();
```

Why Is Object Reflection Useful?

Of course, it doesn't seem very useful in the above example, but reflection can be very useful in some cases. Three main reasons are:

1. It can help you observe or manipulate the runtime behavior of applications.
2. It can help you debug or test programs, as you have direct access to methods, constructors, and fields.
3. You can call methods by name when you don't know the method in advance. For example, we may let the user pass in a class name, parameters for the constructor, and a method name. We can then use this information to create an object and call a method. Doing these operations without reflection would require a complex series of if-statements, if it's possible at all.

13.7 Lambda Expressions: There is a class `Country` that has methods `getContinent()` and `getPopulation()`. Write a function `int getPopulation(List<Country> countries, String continent)` that computes the total population of a given continent, given a list of all countries and the name of a continent.

pg 168

SOLUTION

This question really comes in two parts. First, we need to generate a list of the countries in North America. Then, we need to compute their total population.

Without lambda expressions, this is fairly straightforward to do.

```
int getPopulation(List<Country> countries, String continent) {
   int sum = 0;
   for (Country c : countries) {
      if (c.getContinent().equals(continent)) {
         sum += c.getPopulation();
      }
   }
   return sum;
}
```

To implement this with lambda expressions, let's break this up into multiple parts.

First, we use `filter` to get a list of the countries in the specified continent.

```
Stream<Country> northAmerica = countries.stream().filter(
   country -> { return country.getContinent().equals(continent);}
```

```
);
```

Second, we convert this into a list of populations using `map`.

```
Stream<Integer> populations = northAmerica.map(
  c -> c.getPopulation()
);
```

Third and finally, we compute the sum using `reduce`.

```
int population = populations.reduce(0, (a, b) -> a + b);
```

This function puts it all together.

```
int getPopulation(List<Country> countries, String continent) {
  /* Filter countries. */
  Stream<Country> sublist = countries.stream().filter(
    country -> { return country.getContinent().equals(continent);}
  );

  /* Convert to list of populations. */
  Stream<Integer> populations = sublist.map(
    c -> c.getPopulation()
  );

  /* Sum list. */
  int population = populations.reduce(0, (a, b) -> a + b);
  return population;
}
```

Alternatively, because of the nature of this specific problem, we can actually remove the `filter` entirely. The `reduce` operation can have logic that maps the population of countries not in the right continent to zero. The sum will effectively disregard countries not within `continent`.

```
int getPopulation(List<Country> countries, String continent) {
  Stream<Integer> populations = countries.stream().map(
    c -> c.getContinent().equals(continent) ? c.getPopulation() : 0);
  return populations.reduce(0, (a, b) -> a + b);
}
```

Lambda functions were new to Java 8, so if you don't recognize them, that's probably why. Now is a great time to learn about them, though!

13.8 Lambda Random: Using Lambda expressions, write a function `List<Integer> getRandomSubset(List<Integer> list)` that returns a random subset of arbitrary size. All subsets (including the empty set) should be equally likely to be chosen.

pg 439

SOLUTION

It's tempting to approach this problem by picking a subset size from `0` to `N` and then generating a random subset of that size.

That creates two issues:

1. We'd have to weight those probabilities. If `N > 1`, there are more subsets of size `N/2` than there are of subsets of size `N` (of which there is always only one).
2. It's actually more difficult to generate a subset of a restricted size (e.g., specifically 10) than it is to generate a subset of any size.

Instead, rather than generating a subset based on sizes, let's think about it based on elements. (The fact that we're told to use lambda expressions is also a hint that we should think about some sort of iteration or processing through the elements.)

Imagine we were iterating through `{1, 2, 3}` to generate a subset. Should 1 be in this subset?

We've got two choices: yes or no. We need to weight the probability of "yes" vs. "no" based on the percent of subsets that contain 1. So, what percent of elements contain 1?

For any specific element, there are as many subsets that contain the element as do not contain it. Consider the following:

```
{}          {1}
{2}         {1, 2}
{3}         {1, 3}
{2, 3}      {1, 2, 3}
```

Note how the difference between the subsets on the left and the subsets on the right is the existence of 1. The left and right sides must have the same number of subsets because we can convert from one to the other by just adding an element.

This means that we can generate a random subset by iterating through the list and flipping a coin (i.e., deciding on a 50/50 chance) to pick whether or not each element will be in it.

Without lambda expressions, we can write something like this:

```
1  List<Integer> getRandomSubset(List<Integer> list) {
2    List<Integer> subset = new ArrayList<Integer>();
3    Random random = new Random();
4    for (int item : list) {
5      /* Flip coin. */
6      if (random.nextBoolean()) {
7        subset.add(item);
8      }
9    }
10   return subset;
11 }
```

To implement this approach using lambda expressions, we can do the following:

```
1  List<Integer> getRandomSubset(List<Integer> list) {
2    Random random = new Random();
3    List<Integer> subset = list.stream().filter(
4      k -> { return random.nextBoolean(); /* Flip coin. */
5    }).collect(Collectors.toList());
6    return subset;
7  }
```

Or, we can use a predicate (defined within the class or within the function):

```
1  Random random = new Random();
2  Predicate<Object> flipCoin = o -> {
3    return random.nextBoolean();
4  };
5
6  List<Integer> getRandomSubset(List<Integer> list) {
7    List<Integer> subset = list.stream().filter(flipCoin).
8      collect(Collectors.toList());
9    return subset;
10 }
```

The nice thing about this implementation is that now we can apply the `flipCoin` predicate in other places.

14

Solutions to Databases

Questions 1 through 3 refer to the following database schema:

Apartments	
AptID	int
UnitNumber	varchar(10)
BuildingID	int

Buildings	
BuildingID	int
ComplexID	int
BuildingName	varchar(100)
Address	varchar(500)

Requests	
RequestID	int
Status	varchar(100)
AptID	int
Description	varchar(500)

Complexes	
ComplexID	int
ComplexName	varchar(100)

AptTenants	
TenantID	int
AptID	int

Tenants	
TenantID	int
TenantName	varchar(100)

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.

14.1 Multiple Apartments: Write a SQL query to get a list of tenants who are renting more than one apartment.

pg 172

SOLUTION

To implement this, we can use the `HAVING` and `GROUP BY` clauses and then perform an `INNER JOIN` with `Tenants`.

```
1  SELECT TenantName
2  FROM Tenants
3  INNER JOIN
4     (SELECT TenantID FROM AptTenants GROUP BY TenantID HAVING count(*) > 1) C
5  ON Tenants.TenantID = C.TenantID
```

Whenever you write a `GROUP BY` clause in an interview (or in real life), make sure that anything in the `SELECT` clause is either an aggregate function or contained within the `GROUP BY` clause.

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

pg 173

SOLUTION

This problem uses a straightforward join of Requests and Apartments to get a list of building IDs and the number of open requests. Once we have this list, we join it again with the Buildings table.

```
1  SELECT BuildingName, ISNULL(Count, 0) as 'Count'
2  FROM Buildings
3  LEFT JOIN
4    (SELECT Apartments.BuildingID, count(*) as 'Count'
5     FROM Requests INNER JOIN Apartments
6     ON Requests.AptID = Apartments.AptID
7     WHERE Requests.Status = 'Open'
8     GROUP BY Apartments.BuildingID) ReqCounts
9  ON ReqCounts.BuildingID = Buildings.BuildingID
```

Queries like this that utilize sub-queries should be thoroughly tested, even when coding by hand. It may be useful to test the inner part of the query first, and then test the outer part.

14.3 Close All Requests: Building #11 is undergoing a major renovation. Implement a query to close all requests from apartments in this building.

pg 173

SOLUTION

UPDATE queries, like SELECT queries, can have WHERE clauses. To implement this query, we get a list of all apartment IDs within building #11 and the list of update requests from those apartments.

```
1  UPDATE Requests
2  SET Status = 'Closed'
3  WHERE AptID IN (SELECT AptID FROM Apartments WHERE BuildingID = 11)
```

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

pg 173

SOLUTION

JOIN is used to combine the results of two tables. To perform a JOIN, each of the tables must have at least one field that will be used to find matching records from the other table. The join type defines which records will go into the result set.

Let's take for example two tables: one table lists the "regular" beverages, and another lists the calorie-free beverages. Each table has two fields: the beverage name and its product code. The "code" field will be used to perform the record matching.

Regular Beverages:

Name	Code
Budweiser	BUDWEISER
Coca-Cola	COCACOLA

Name	Code
Pepsi	PEPSI

Calorie-Free Beverages:

Name	Code
Diet Coca-Cola	COCACOLA
Fresca	FRESCA
Diet Pepsi	PEPSI
Pepsi Light	PEPSI
Purified Water	Water

If we wanted to join `Beverage` with `Calorie-Free Beverages`, we would have many options. These are discussed below.

- `INNER JOIN`: The result set would contain only the data where the criteria match. In our example, we would get three records: one with a COCACOLA code and two with `PEPSI` codes.
- `OUTER JOIN`: An `OUTER JOIN` will always contain the results of `INNER JOIN`, but it may also contain some records that have no matching record in the other table. `OUTER JOIN`s are divided into the following subtypes:
  - `LEFT OUTER JOIN`, or simply `LEFT JOIN`: The result will contain all records from the left table. If no matching records were found in the right table, then its fields will contain the `NULL` values. In our example, we would get four records. In addition to `INNER JOIN` results, `BUDWEISER` would be listed, because it was in the left table.
  - `RIGHT OUTER JOIN`, or simply `RIGHT JOIN`: This type of join is the opposite of `LEFT JOIN`. It will contain every record from the right table; the missing fields from the left table will be `NULL`. Note that if we have two tables, A and B, then we can say that the statement A `LEFT JOIN` B is equivalent to the statement B `RIGHT JOIN` A. In our example above, we will get five records. In addition to `INNER JOIN` results, `FRESCA` and `WATER` records will be listed.
  - `FULL OUTER JOIN`: This type of join combines the results of the `LEFT` and `RIGHT JOINS`. All records from both tables will be included in the result set, regardless of whether or not a matching record exists in the other table. If no matching record was found, then the corresponding result fields will have a `NULL` value. In our example, we will get six records.

14.5 Denormalization: What is denormalization? Explain the pros and cons.

pg 173

SOLUTION

Denormalization is a database optimization technique in which we add redundant data to one or more tables. This can help us avoid costly joins in a relational database.

By contrast, in a traditional normalized database, we store data in separate logical tables and attempt to minimize redundant data. We may strive to have only one copy of each piece of data in the database.

For example, in a normalized database, we might have a `Courses` table and a `Teachers` table. Each entry in `Courses` would store the `teacherID` for a `Course` but not the `teacherName`. When we need to retrieve a list of all `Courses` with the `Teacher` name, we would do a join between these two tables.

In some ways, this is great; if a teacher changes his or her name, we only have to update the name in one place.

The drawback, however, is that if the tables are large, we may spend an unnecessarily long time doing joins on tables.

Denormalization, then, strikes a different compromise. Under denormalization, we decide that we're okay with some redundancy and some extra effort to update the database in order to get the efficiency advantages of fewer joins.

Cons of Denormalization	Pros of Denormalization
Updates and inserts are more expensive.	Retrieving data is faster since we do fewer joins.
Denormalization can make update and insert code harder to write.	Queries to retrieve can be simpler (and therefore less likely to have bugs), since we need to look at fewer tables.
Data may be inconsistent. Which is the "correct" value for a piece of data?	
Data redundancy necessitates more storage.	

In a system that demands scalability, like that of any major tech companies, we almost always use elements of both normalized and denormalized databases.

14.6 Entity-Relationship Diagram: Draw an entity-relationship diagram for a database with companies, people, and professionals (people who work for companies).

pg 173

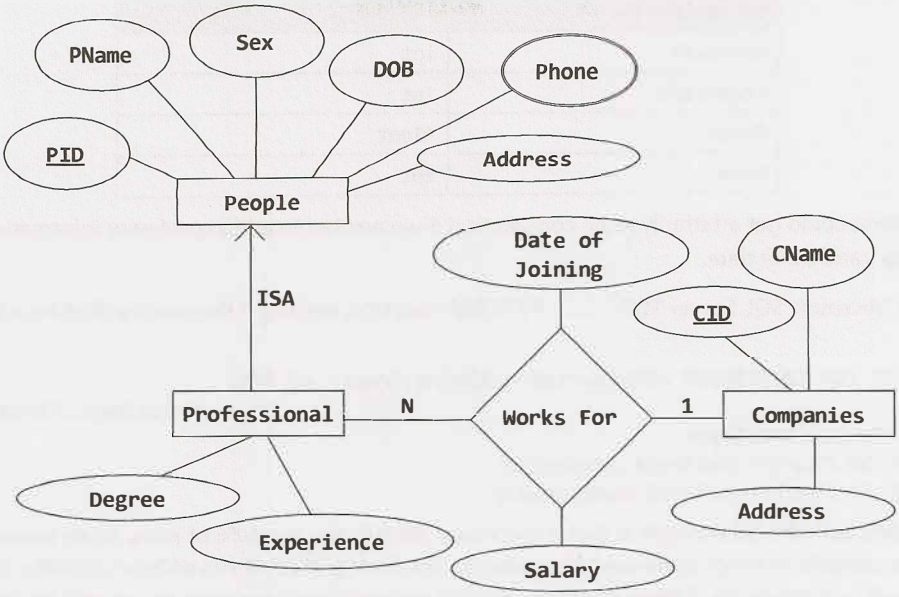
SOLUTION

People who work for `Companies` are `Professionals`. So, there is an `ISA` ("is a") relationship between `People` and `Professionals` (or we could say that a `Professional` is derived from `People`).

Each `Professional` has additional information such as degree and work experiences in addition to the properties derived from `People`.

A `Professional` works for one company at a time (probably—you might want to validate this assumption), but `Companies` can hire many `Professionals`. So, there is a many-to-one relationship between `Professionals` and `Companies`. This "Works For" relationship can store attributes such as an employee's start date and salary. These attributes are defined only when we relate a `Professional` with a `Company`.

A `Person` can have multiple phone numbers, which is why `Phone` is a multi-valued attribute.



14.7 Design Grade Database: 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 173

SOLUTION

In a simplistic database, we'll have at least three objects: `Students`, `Courses`, and `CourseEnrollment`. `Students` will have at least a student name and ID and will likely have other personal information. `Courses` will contain the course name and ID and will likely contain the course description, professor, and other information. `CourseEnrollment` will pair `Students` and `Courses` and will also contain a field for `CourseGrade`.

Students	
StudentID	int
StudentName	varchar(100)
Address	varchar(500)

Courses	
CourseID	int
CourseName	varchar(100)
ProfessorID	int

CourseEnrollment	
CourseID	int
StudentID	int
Grade	float
Term	int

This database could get arbitrarily more complicated if we wanted to add in professor information, billing information, and other data.

Using the Microsoft SQL Server TOP ... PERCENT function, we might (incorrectly) first try a query like this:

```
1  SELECT TOP 10 PERCENT AVG(CourseEnrollment.Grade) AS GPA,
2                                                  CourseEnrollment.StudentID
3  FROM CourseEnrollment
4  GROUP BY CourseEnrollment.StudentID
5  ORDER BY AVG(CourseEnrollment.Grade)
```

The problem with the above code is that it will return literally the top 10% of rows, when sorted by GPA. Imagine a scenario in which there are 100 students, and the top 15 students all have 4.0 GPAs. The above function will only return 10 of those students, which is not really what we want. In case of a tie, we want to include the students who tied for the top 10% -- even if this means that our honor roll includes more than 10% of the class.

To correct this issue, we can build something similar to this query, but instead first get the GPA cut off.

```
1  DECLARE @GPACutOff float;
2  SET @GPACutOff = (SELECT min(GPA) as 'GPAMin' FROM (
3        SELECT TOP 10 PERCENT AVG(CourseEnrollment.Grade) AS GPA
4        FROM CourseEnrollment
5        GROUP BY CourseEnrollment.StudentID
6        ORDER BY GPA desc) Grades);
```

Then, once we have @GPACutOff defined, selecting the students with at least this GPA is reasonably straightforward.

```
1  SELECT StudentName, GPA
2  FROM (SELECT AVG(CourseEnrollment.Grade) AS GPA, CourseEnrollment.StudentID
3        FROM CourseEnrollment
4        GROUP BY CourseEnrollment.StudentID
5        HAVING AVG(CourseEnrollment.Grade) >= @GPACutOff) Honors
6  INNER JOIN Students ON Honors.StudentID = Student.StudentID
```

Be very careful about what implicit assumptions you make. If you look at the above database description, what potentially incorrect assumption do you see? One is that each course can only be taught by one professor. At some schools, courses may be taught by multiple professors.

However, you *will* need to make some assumptions, or you'd drive yourself crazy. Which assumptions you make is less important than just recognizing *that* you made assumptions. Incorrect assumptions, both in the real world and in an interview, can be dealt with *as long as they are acknowledged.*

Remember, additionally, that there's a trade-off between flexibility and complexity. Creating a system in which a course can have multiple professors does increase the database's flexibility, but it also increases its complexity. If we tried to make our database flexible to every possible situation, we'd wind up with something hopelessly complex.

Make your design reasonably flexible, and state any other assumptions or constraints. This goes for not just database design, but object-oriented design and programming in general.

15

Solutions to Threads and Locks

15.1 Thread vs. Process: What's the difference between a thread and a process?

pg 179

SOLUTION

Processes and threads are related to each other but are fundamentally different.

A process can be thought of as an instance of a program in execution. A process is an independent entity to which system resources (e.g., CPU time and memory) are allocated. Each process is executed in a separate address space, and one process cannot access the variables and data structures of another process. If a process wishes to access another process' resources, inter-process communications have to be used. These include pipes, files, sockets, and other forms.

A thread exists within a process and shares the process' resources (including its heap space). Multiple threads within the same process will share the same heap space. This is very different from processes, which cannot directly access the memory of another process. Each thread still has its own registers and its own stack, but other threads can read and write the heap memory.

A thread is a particular execution path of a process. When one thread modifies a process resource, the change is immediately visible to sibling threads.

15.2 Context Switch: How would you measure the time spent in a context switch?

pg 179

SOLUTION

This is a tricky question, but let's start with a possible solution.

A context switch is the time spent switching between two processes (i.e., bringing a waiting process into execution and sending an executing process into waiting/terminated state). This happens in multitasking. The operating system must bring the state information of waiting processes into memory and save the state information of the currently running process.

In order to solve this problem, we would like to record the timestamps of the last and first instruction of the swapping processes. The context switch time is the difference in the timestamps between the two processes.

Let's take an easy example: Assume there are only two processes, P_1 and P_2.

P_1 is executing and P_2 is waiting for execution. At some point, the operating system must swap P_1 and P_2—let's assume it happens at the Nth instruction of P_1. If $t_{x,k}$ indicates the timestamp in microseconds of the kth instruction of process x, then the context switch would take $t_{2,1} - t_{1,n}$ microseconds.

The tricky part is this: how do we know when this swapping occurs? We cannot, of course, record the timestamp of every instruction in the process.

Another issue is that swapping is governed by the scheduling algorithm of the operating system and there may be many kernel level threads which are also doing context switches. Other processes could be contending for the CPU or the kernel handling interrupts. The user does not have any control over these extraneous context switches. For instance, if at time $t_{1,n}$ the kernel decides to handle an interrupt, then the context switch time would be overstated.

In order to overcome these obstacles, we must first construct an environment such that after P_1 executes, the task scheduler immediately selects P_2 to run. This may be accomplished by constructing a data channel, such as a pipe, between P_1 and P_2 and having the two processes play a game of ping-pong with a data token.

That is, let's allow P_1 to be the initial sender and P_2 to be the receiver. Initially, P_2 is blocked (sleeping) as it awaits the data token. When P_1 executes, it delivers the token over the data channel to P_2 and immediately attempts to read a response token. However, since P_2 has not yet had a chance to run, no such token is available for P_1 and the process is blocked. This relinquishes the CPU.

A context switch results and the task scheduler must select another process to run. Since P_2 is now in a ready-to-run state, it is a desirable candidate to be selected by the task scheduler for execution. When P_2 runs, the roles of P_1 and P_2 are swapped. P_2 is now acting as the sender and P_1 as the blocked receiver. The game ends when P_2 returns the token to P_1.

To summarize, an iteration of the game is played with the following steps:

1. P_2 blocks awaiting data from P_1.
2. P_1 marks the start time.
3. P_1 sends token to P_2.
4. P_1 attempts to read a response token from P_2. This induces a context switch.
5. P_2 is scheduled and receives the token.
6. P_2 sends a response token to P_1.
7. P_2 attempts read a response token from P_1. This induces a context switch.
8. P_1 is scheduled and receives the token.
9. P_1 marks the end time.

The key is that the delivery of a data token induces a context switch. Let T_d and T_r be the time it takes to deliver and receive a data token, respectively, and let T_c be the amount of time spent in a context switch. At step 2, P_1 records the timestamp of the delivery of the token, and at step 9, it records the timestamp of the response. The amount of time elapsed, T, between these events may be expressed by:

$$T = 2 * (T_d + T_c + T_r)$$

This formula arises because of the following events: P_1 sends a token (3), the CPU context switches (4), P_2 receives it (5). P_2 then sends the response token (6), the CPU context switches (7), and finally P_1 receives it (8).

P_1 will be able to easily compute T, since this is just the time between events 3 and 8. So, to solve for T_c, we must first determine the value of $T_d + T_r$.

How can we do this? We can do this by measuring the length of time it takes P_1 to send and receive a token to itself. This will not induce a context switch since P_1 is running on the CPU at the time it sent the token and will not block to receive it.

The game is played a number of iterations to average out any variability in the elapsed time between steps 2 and 9 that may result from unexpected kernel interrupts and additional kernel threads contending for the CPU. We select the smallest observed context switch time as our final answer.

However, all we can ultimately say that this is an approximation which depends on the underlying system. For example, we make the assumption that P_2 is selected to run once a data token becomes available. However, this is dependent on the implementation of the task scheduler and we cannot make any guarantees.

That's okay; it's important in an interview to recognize when your solution might not be perfect.

15.3 Dining Philosophers: 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 180

SOLUTION

First, let's implement a simple simulation of the dining philosophers problem in which we don't concern ourselves with deadlocks. We can implement this solution by having `Philosopher` extend `Thread`, and `Chopstick` call `lock.lock()` when it is picked up and `lock.unlock()` when it is put down.

```
1  class Chopstick {
2    private Lock lock;
3
4    public Chopstick() {
5      lock = new ReentrantLock();
6    }
7
8    public void pickUp() {
9      void lock.lock();
10   }
11
12   public void putDown() {
13     lock.unlock();
14   }
15 }
16
17 class Philosopher extends Thread {
18   private int bites = 10;
19   private Chopstick left, right;
20
21   public Philosopher(Chopstick left, Chopstick right) {
22     this.left = left;
23     this.right = right;
24   }
```

```
25
26    public void eat() {
27       pickUp();
28       chew();
29       putDown();
30    }
31
32    public void pickUp() {
33       left.pickUp();
34       right.pickUp();
35    }
36
37    public void chew() { }
38
39    public void putDown() {
40       right.putDown();
41       left.putDown();
42    }
43
44    public void run() {
45       for (int i = 0; i < bites; i++) {
46          eat();
47       }
48    }
49 }
```

Running the above code may lead to a deadlock if all the philosophers have a left chopstick and are waiting for the right one.

Solution #1: All or Nothing

To prevent deadlocks, we can implement a strategy where a philosopher will put down his left chopstick if he is unable to obtain the right one.

```
1  public class Chopstick {
2     /* same as before */
3
4     public boolean pickUp() {
5        return lock.tryLock();
6     }
7  }
8
9  public class Philosopher extends Thread {
10    /* same as before */
11
12    public void eat() {
13       if (pickUp()) {
14          chew();
15          putDown();
16       }
17    }
18
19    public boolean pickUp() {
20       /* attempt to pick up */
21       if (!left.pickUp()) {
22          return false;
23       }
24       if (!right.pickUp()) {
```

```
25          left.putDown();
26          return false;
27        }
28        return true;
29      }
30    }
```

In the above code, we need to be sure to release the left chopstick if we can't pick up the right one—and to not call `putDown()` on the chopsticks if we never had them in the first place.

One issue with this is that if all the philosophers were perfectly synchronized, they could simultaneously pick up their left chopstick, be unable to pick up the right one, and then put back down the left one—only to have the process repeated again.

Solution #2: Prioritized Chopsticks

Alternatively, we can label the chopsticks with a number from `0` to `N - 1`. Each philosopher attempts to pick up the lower numbered chopstick first. This essentially means that each philosopher goes for the left chopstick before right one (assuming that's the way you labeled it), except for the last philosopher who does this in reverse. This will break the cycle.

```
1   public class Philosopher extends Thread {
2     private int bites = 10;
3     private Chopstick lower, higher;
4     private int index;
5     public Philosopher(int i, Chopstick left, Chopstick right) {
6       index = i;
7       if (left.getNumber() < right.getNumber()) {
8         this.lower = left;
9         this.higher = right;
10      } else {
11        this.lower = right;
12        this.higher = left;
13      }
14    }
15
16    public void eat() {
17      pickUp();
18      chew();
19      putDown();
20    }
21
22    public void pickUp() {
23      lower.pickUp();
24      higher.pickUp();
25    }
26
27    public void chew() { ... }
28
29    public void putDown() {
30      higher.putDown();
31      lower.putDown();
32    }
33
34    public void run() {
35      for (int i = 0; i < bites; i++) {
36        eat();
```

```
37        }
38     }
39  }
40
41  public class Chopstick {
42     private Lock lock;
43     private int number;
44
45     public Chopstick(int n) {
46        lock = new ReentrantLock();
47        this.number = n;
48     }
49
50     public void pickUp() {
51        lock.lock();
52     }
53
54     public void putDown() {
55        lock.unlock();
56     }
57
58     public int getNumber() {
59        return number;
60     }
61  }
```

With this solution, a philosopher can never hold the larger chopstick without holding the smaller one. This prevents the ability to have a cycle, since a cycle means that a higher chopstick would "point" to a lower one.

15.4 Deadlock-Free Class: Design a class which provides a lock only if there are no possible deadlocks.

pg 180

SOLUTION

There are several common ways to prevent deadlocks. One of the popular ways is to require a process to declare upfront what locks it will need. We can then verify if a deadlock would be created by issuing these locks, and we can fail if so.

With these constraints in mind, let's investigate how we can detect deadlocks. Suppose this was the order of locks requested:

```
A = {1, 2, 3, 4}
B = {1, 3, 5}
C = {7, 5, 9, 2}
```

This may create a deadlock because we could have the following scenario:

```
A locks 2, waits on 3
B locks 3, waits on 5
C locks 5, waits on 2
```

We can think about this as a graph, where 2 is connected to 3, 3 is connected to 5, and 5 is connected to 2. A deadlock is represented by a cycle. An edge `(w, v)` exists in the graph if a process declares that it will request lock `v` immediately after lock `w`. For the earlier example, the following edges would exist in the graph: `(1, 2)`, `(2, 3)`, `(3, 4)`, `(1, 3)`, `(3, 5)`, `(7, 5)`, `(5, 9)`, `(9, 2)`. The "owner" of the edge does not matter.

This class will need a `declare` method, which threads and processes will use to declare what order they will request resources in. This declare method will iterate through the declare order, adding each contiguous pair of elements (`v`, `w`) to the graph. Afterwards, it will check to see if any cycles have been created. If any cycles have been created, it will backtrack, removing these edges from the graph, and then exit.

We have one final component to discuss: how do we detect a cycle? We can detect a cycle by doing a depth-first search through each connected component (i.e., each connected part of the graph). Complex algorithms exist to find all the connected components of a graph, but our work in this problem does not require this degree of complexity.

We know that if a cycle was created, one of our new edges must be to blame. Thus, as long as our depth-first search touches all of these edges at some point, then we know that we have fully searched for a cycle.

The pseudocode for this special case cycle detection looks like this:

```
boolean checkForCycle(locks[] locks) {
   touchedNodes = hash table(lock -> boolean)
   initialize touchedNodes to false for each lock in locks
   for each (lock x in process.locks) {
      if (touchedNodes[x] == false) {
         if (hasCycle(x, touchedNodes)) {
            return true;
         }
      }
   }
   return false;
}

boolean hasCycle(node x, touchedNodes) {
   touchedNodes[r] = true;
   if (x.state == VISITING) {
      return true;
   } else if (x.state == FRESH) {
      ... (see full code below)
   }
}
```

In the above code, note that we may do several depth-first searches, but `touchedNodes` is only initialized once. We iterate until all the values in `touchedNodes` are false.

The code below provides further details. For simplicity, we assume that all locks and processes (owners) are ordered sequentially.

```
class LockFactory {
   private static LockFactory instance;

   private int numberOfLocks = 5; /* default */
   private LockNode[] locks;

   /* Maps from a process or owner to the order that the owner claimed it would
    * call the locks in */
   private HashMap<Integer, LinkedList<LockNode>> lockOrder;

   private LockFactory(int count) { ... }
   public static LockFactory getInstance() { return instance; }

   public static synchronized LockFactory initialize(int count) {
      if (instance == null) instance = new LockFactory(count);
```

```
16      return instance;
17    }
18
19    public boolean hasCycle(HashMap<Integer, Boolean> touchedNodes,
20                            int[] resourcesInOrder) {
21      / *check for a cycle */
22      for (int resource : resourcesInOrder) {
23        if (touchedNodes.get(resource) == false) {
24          LockNode n = locks[resource];
25          if (n.hasCycle(touchedNodes)) {
26            return true;
27          }
28        }
29      }
30      return false;
31    }
32
33    / *To prevent deadlocks, force the processes to declare upfront what order they
34     * will need the locks in. Verify that this order does not create a deadlock (a
35     * cycle in a directed graph) */
36    public boolean declare(int ownerId, int[] resourcesInOrder) {
37      HashMap<Integer, Boolean> touchedNodes = new HashMap<Integer, Boolean>();
38
39      / *add nodes to graph */
40      int index = 1;
41      touchedNodes.put(resourcesInOrder[0], false);
42      for (index = 1; index < resourcesInOrder.length; index++) {
43        LockNode prev = locks[resourcesInOrder[index - 1]];
44        LockNode curr = locks[resourcesInOrder[index]];
45        prev.joinTo(curr);
46        touchedNodes.put(resourcesInOrder[index], false);
47      }
48
49      / *if we created a cycle, destroy this resource list and return false */
50      if (hasCycle(touchedNodes, resourcesInOrder)) {
51        for (int j = 1; j < resourcesInOrder.length; j++) {
52          LockNode p = locks[resourcesInOrder[j - 1]];
53          LockNode c = locks[resourcesInOrder[j]];
54          p.remove(c);
55        }
56        return false;
57      }
58
59      / *No cycles detected. Save the order that was declared, so that we can
60       * verify that the process is really calling the locks in the order it said
61       * it would. */
62      LinkedList<LockNode> list = new LinkedList<LockNode>();
63      for (int i = 0; i < resourcesInOrder.length; i++) {
64        LockNode resource = locks[resourcesInOrder[i]];
65        list.add(resource);
66      }
67      lockOrder.put(ownerId, list);
68
69      return true;
70    }
71
```

```
72    /* Get the lock, verifying first that the process is really calling the locks in
73     * the order it said it would. */
74    public Lock getLock(int ownerId, int resourceID) {
75       LinkedList<LockNode> list = lockOrder.get(ownerId);
76       if (list == null) return null;
77
78       LockNode head = list.getFirst();
79       if (head.getId() == resourceID) {
80          list.removeFirst();
81          return head.getLock();
82       }
83       return null;
84    }
85 }
86
87 public class LockNode {
88    public enum VisitState { FRESH, VISITING, VISITED };
89
90    private ArrayList<LockNode> children;
91    private int lockId;
92    private Lock lock;
93    private int maxLocks;
94
95    public LockNode(int id, int max) { ... }
96
97    /* Join "this" to "node", checking that it doesn't create a cycle */
98    public void joinTo(LockNode node) { children.add(node); }
99    public void remove(LockNode node) { children.remove(node); }
100
101   /* Check for a cycle by doing a depth-first-search. */
102   public boolean hasCycle(HashMap<Integer, Boolean> touchedNodes) {
103      VisitState[] visited = new VisitState[maxLocks];
104      for (int i = 0; i < maxLocks; i++) {
105         visited[i] = VisitState.FRESH;
106      }
107      return hasCycle(visited, touchedNodes);
108   }
109
110   private boolean hasCycle(VisitState[] visited,
111                            HashMap<Integer, Boolean> touchedNodes) {
112      if (touchedNodes.containsKey(lockId)) {
113         touchedNodes.put(lockId, true);
114      }
115
116      if (visited[lockId] == VisitState.VISITING) {
117         /* We looped back to this node while still visiting it, so we know there's
118          * a cycle. */
119         return true;
120      } else if (visited[lockId] == VisitState.FRESH) {
121         visited[lockId] = VisitState.VISITING;
122         for (LockNode n : children) {
123            if (n.hasCycle(visited, touchedNodes)) {
124               return true;
125            }
126         }
127         visited[lockId] = VisitState.VISITED;
```

```
128       }
129       return false;
130    }
131
132    public Lock getLock() {
133       if (lock == null) lock = new ReentrantLock();
134       return lock;
135    }
136
137    public int getId() { return lockId; }
138 }
```

As always, when you see code this complicated and lengthy, you wouldn't be expected to write all of it. More likely, you would be asked to sketch out pseudocode and possibly implement one of these methods.

15.5 Call In Order: 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 180

SOLUTION

The general logic is to check if `first()` has completed before executing `second()`, and if `second()` has completed before calling `third()`. Because we need to be very careful about thread safety, simple `boolean` flags won't do the job.

What about using a `lock` to do something like the below code?

```
1  public class FooBad {
2     public int pauseTime = 1000;
3     public ReentrantLock lock1, lock2;
4
5     public FooBad() {
6        try {
7           lock1 = new ReentrantLock();
8           lock2 = new ReentrantLock();
9
10          lock1.lock();
11          lock2.lock();
12       } catch (...) { ... }
13    }
14
15    public void first() {
16       try {
17          ...
18          lock1.unlock(); // mark finished with first()
19       } catch (...) { ... }
20    }
```

```
21
22    public void second() {
23       try {
24          lock1.lock(); // wait until finished with first()
25          lock1.unlock();
26          ...
27
28          lock2.unlock(); // mark finished with second()
29       } catch (...) { ... }
30    }
31
32    public void third() {
33       try {
34          lock2.lock(); // wait until finished with third()
35          lock2.unlock();
36          ...
37       } catch (...) { ... }
38    }
39 }
```

This code won't actually quite work due to the concept of *lock ownership*. One thread is actually performing the lock (in the FooBad constructor), but different threads attempt to unlock the locks. This is not allowed, and your code will raise an exception. A lock in Java is owned by the same thread which locked it.

Instead, we can replicate this behavior with semaphores. The logic is identical.

```
1  public class Foo {
2     public Semaphore sem1, sem2;
3
4     public Foo() {
5        try {
6           sem1 = new Semaphore(1);
7           sem2 = new Semaphore(1);
8
9           sem1.acquire();
10          sem2.acquire();
11       } catch (...) { ... }
12    }
13
14    public void first() {
15       try {
16          ...
17          sem1.release();
18       } catch (...) { ... }
19    }
20
21    public void second() {
22       try {
23          sem1.acquire();
24          sem1.release();
25          ...
26          sem2.release();
27       } catch (...) { ... }
28    }
29
30    public void third() {
31       try {
32          sem2.acquire();
```

```
33          sem2.release();
34          ...
35        } catch (...) { ... }
36    }
37  }
```

15.6 Synchronized Methods: 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 180

SOLUTION

By applying the word `synchronized` to a method, we ensure that two threads cannot execute synchronized methods *on the same object instance* at the same time.

So, the answer to the first part really depends. If the two threads have the same instance of the object, then no, they cannot simultaneously execute method A. However, if they have different instances of the object, then they can.

Conceptually, you can see this by considering locks. A synchronized method applies a "lock" on *all* synchronized methods in that instance of the object. This blocks other threads from executing synchronized methods within that instance.

In the second part, we're asked if `thread1` can execute synchronized method A while `thread2` is executing non-synchronized method B. Since B is not synchronized, there is nothing to block `thread1` from executing A while `thread2` is executing B. This is true regardless of whether `thread1` and `thread2` have the same instance of the object.

Ultimately, the key concept to remember is that only one synchronized method can be in execution per instance of that object. Other threads can execute non-synchronized methods on that instance, or they can execute any method on a different instance of the object.

15.7 FizzBuzz: In the classic problem FizzBuzz, you are told to print the numbers from 1 to n. However, when the number is divisible by 3, print "Fizz". When it is divisible by 5, print "Buzz". When it is divisible by 3 and 5, print "FizzBuzz". In this problem, you are asked to do this in a multithreaded way. Implement a multithreaded version of FizzBuzz with four threads. One thread checks for divisibility of 3 and prints "Fizz". Another thread is responsible for divisibility of 5 and prints "Buzz". A third thread is responsible for divisibility of 3 and 5 and prints "FizzBuzz". A fourth thread does the numbers.

pg 180

SOLUTION

Let's start off with implementing a single threaded version of FizzBuzz.

Single Threaded

Although this problem (in the single threaded version) shouldn't be hard, a lot of candidates overcomplicate it. They look for something "beautiful" that reuses the fact that the divisible by 3 and 5 case ("FizzBuzz") seems to resemble the individual cases ("Fizz" and "Buzz").

In actuality, the best way to do it, considering readability and efficiency, is just the straightforward way.

```
1  void fizzbuzz(int n) {
```

```
2     for (int i = 1; i <= n; i++) {
3       if (i % 3 == 0 && i % 5 == 0) {
4         System.out.println("FizzBuzz");
5       } else if (i % 3 == 0) {
6         System.out.println("Fizz");
7       } else if (i % 5 == 0) {
8         System.out.println("Buzz");
9       } else {
10        System.out.println(i);
11      }
12    }
13  }
```

The primary thing to be careful of here is the order of the statements. If you put the check for divisibility by 3 before the check for divisibility by 3 and 5, it won't print the right thing.

Multithreaded

To do this multithreaded, we want a structure that looks something like this:

FizzBuzz Thread	Fizz Thread

if i div by 3 && 5
  print FizzBuzz
  increment i
repeat until i > n
	if i div by only 3
	  print Fizz
	  increment i
	repeat until i > n

Buzz Thread	Number Thread

if i div by only 5
  print Buzz
  increment i
repeat until i > n
	if i not div by 3 or 5
	  print i
	  increment i
	repeat until i > n

The code for this will look something like:

```
1   while (true) {
2     if (current > max) {
3       return;
4     }
5     if (/* divisibility test */) {
6       System.out.println(/* print something */);
7       current++;
8     }
9   }
```

We'll need to add some synchronization in the loop. Otherwise, the value of `current` could change between lines 2 - 4 and lines 5 - 8, and we can inadvertently exceed the intended bounds of the loop. Additionally, incrementing is not thread-safe.

To actually implement this concept, there are many possibilities. One possibility is to have four entirely separate thread classes that share a reference to the current variable (which can be wrapped in an object).

The loop for each thread is substantially similar. They just have different target values for the divisibility checks, and different print values.

	FizzBuzz	Fizz	Buzz	Number
current % 3 == 0	true	true	false	false
current % 5 == 0	true	false	true	false
to print	FizzBuzz	Fizz	Buzz	current

For the most part, this can be handled by taking in "target" parameters and the value to print. The output for the `Number` thread needs to be overwritten, though, as it's not a simple, fixed string.

We can implement a `FizzBuzzThread` class which handles most of this. A `NumberThread` class can extend `FizzBuzzThread` and override the `print` method.

```
1  Thread[] threads = {new FizzBuzzThread(true, true, n, "FizzBuzz"),
2                      new FizzBuzzThread(true, false, n, "Fizz"),
3                      new FizzBuzzThread(false, true, n, "Buzz"),
4                      new NumberThread(false, false, n)};
5  for (Thread thread : threads) {
6    thread.start();
7  }
8
9  public class FizzBuzzThread extends Thread {
10   private static Object lock = new Object();
11   protected static int current = 1;
12   private int max;
13   private boolean div3, div5;
14   private String toPrint;
15
16   public FizzBuzzThread(boolean div3, boolean div5, int max, String toPrint) {
17     this.div3 = div3;
18     this.div5 = div5;
19     this.max = max;
20     this.toPrint = toPrint;
21   }
22
23   public void print() {
24     System.out.println(toPrint);
25   }
26
27   public void run() {
28     while (true) {
29       synchronized (lock) {
30         if (current > max) {
31           return;
32         }
33
34         if ((current % 3 == 0) == div3 &&
35             (current % 5 == 0) == div5) {
36           print();
37           current++;
38         }
39       }
40     }
41   }
42 }
43
44 public class NumberThread extends FizzBuzzThread {
```

```
45    public NumberThread(boolean div3, boolean div5, int max) {
46       super(div3, div5, max, null);
47    }
48
49    public void print() {
50       System.out.println(current);
51    }
52 }
```

Observe that we need to put the comparison of `current` and `max` before the if statement, to ensure the value will only get printed when `current` is less than or equal to `max`.

Alternatively, if we're working in a language which supports this (Java 8 and many other languages do), we can pass in a `validate` method and a `print` method as parameters.

```
1  int n = 100;
2  Thread[] threads = {
3     new FBThread(i -> i % 3 == 0 && i % 5 == 0, i -> "FizzBuzz", n),
4     new FBThread(i -> i % 3 == 0 && i % 5 != 0, i -> "Fizz", n),
5     new FBThread(i -> i % 3 != 0 && i % 5 == 0, i -> "Buzz", n),
6     new FBThread(i -> i % 3 != 0 && i % 5 != 0, i -> Integer.toString(i), n)};
7  for (Thread thread : threads) {
8     thread.start();
9  }
10
11 public class FBThread extends Thread {
12    private static Object lock = new Object();
13    protected static int current = 1;
14    private int max;
15    private Predicate<Integer> validate;
16    private Function<Integer, String> printer;
17    int x = 1;
18
19    public FBThread(Predicate<Integer> validate,
20                    Function<Integer, String> printer, int max) {
21       this.validate = validate;
22       this.printer = printer;
23       this.max = max;
24    }
25
26    public void run() {
27       while (true) {
28          synchronized (lock) {
29             if (current > max) {
30                return;
31             }
32             if (validate.test(current)) {
33                System.out.println(printer.apply(current));
34                current++;
35             }
36          }
37       }
38    }
39 }
```

There are of course many other ways of implementing this as well.