```
3   );
```

Second, we convert this into a list of populations using map.

```
1   Stream<Integer> populations = northAmerica.map(
2     c -> c.getPopulation()
3   );
```

Third and finally, we compute the sum using reduce.

```
1   int population = populations.reduce(0, (a, b) -> a + b);
```

This function puts it all together.

```
1   int getPopulation(List<Country> countries, String continent) {
2     /* Filter countries. */
3     Stream<Country> sublist = countries.stream().filter(
4       country -> { return country.getContinent().equals(continent);}
5     );
6
7     /* Convert to list of populations. */
8     Stream<Integer> populations = sublist.map(
9       c -> c.getPopulation()
10    );
11
12    /* Sum list. */
13    int population = populations.reduce(0, (a, b) -> a + b);
14    return population;
15  }
```

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.

```
1   int getPopulation(List<Country> countries, String continent) {
2     Stream<Integer> populations = countries.stream().map(
3       c -> c.getContinent().equals(continent) ? c.getPopulation() : 0);
4     return populations.reduce(0, (a, b) -> a + b);
5   }
```

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

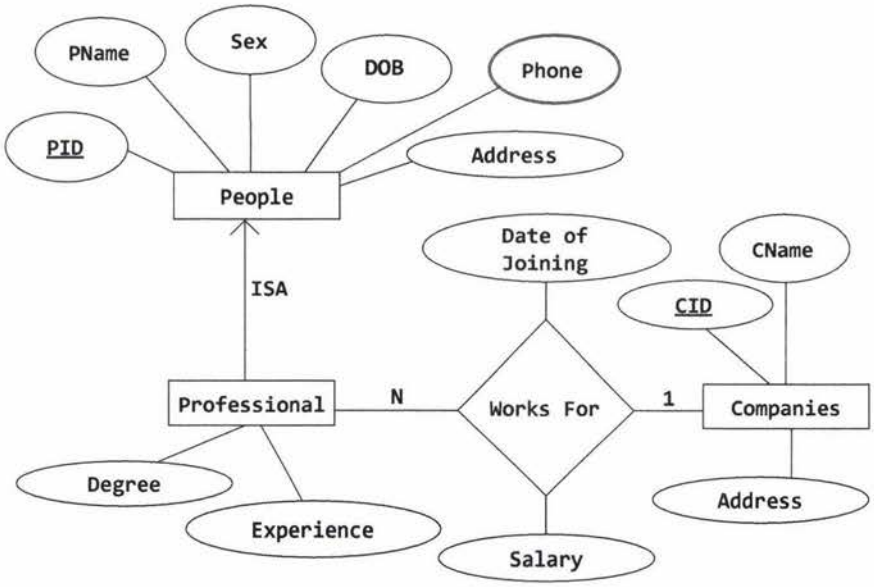
SOLUTION

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 time-stamp 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.

16

Solutions to Moderate

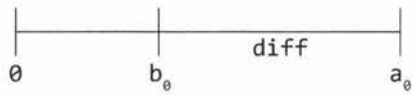
16.1 Number Swapper: Write a function to swap a number in place (that is, without temporary variables).

pg 181

SOLUTION

This is a classic interview problem, and it's a reasonably straightforward one. We'll walk through this using a_0 to indicate the original value of a and b_0 to indicate the original value of b. We'll also use `diff` to indicate the value of a_0 - b_0.

Let's picture these on a number line for the case where a > b.



First, we briefly set a to `diff`, which is the right side of the above number line. Then, when we add b and `diff` (and store that value in b), we get a_0. We now have b = a_0 and a = `diff`. All that's left to do is to set a equal to a_0 - `diff`, which is just b - a.

The code below implements this.

```
1  // Example for a = 9, b = 4
2  a = a - b; // a = 9 - 4 = 5
3  b = a + b; // b = 5 + 4 = 9
4  a = b - a; // a = 9 - 5
```

We can implement a similar solution with bit manipulation. The benefit of this solution is that it works for more data types than just integers.

```
1  // Example for a = 101 (in binary) and b = 110
2  a = a^b; // a = 101^110 = 011
3  b = a^b; // b = 011^110 = 101
4  a = a^b; // a = 011^101 = 110
```

This code works by using XORs. The easiest way to see how this works is by focusing on a specific bit. If we can correctly swap two bits, then we know the entire operation works correctly.

Let's take two bits, x and y, and walk through this line by line.

1. x = x ^ y

   This line essentially checks if x and y have different values. It will result in 1 if and only if x != y.

2. y = x ^ y

Or: `y = {0 if originally same, 1 if different} ^ {original y}`

Observe that XORing a bit with 1 always flips the bit, whereas XORing with 0 will never change it.

Therefore, if we do `y = 1 ^ {original y}` when `x != y`, then y will be flipped and therefore have x's original value.

Otherwise, if `x == y`, then we do `y = 0 ^ {original y}` and the value of y does not change.

Either way, y will be equal to the original value of x.

3. `x = x ^ y`

   Or: `x = {0 if originally same, 1 if different} ^ {original x}`

   At this point, y is equal to the original value of x. This line is essentially equivalent to the line above it, but for different variables.

   If we do `x = 1 ^ {original x}` when the values are different, x will be flipped.

   If we do `x = 0 ^ {original x}` when the values are the same, x will not be changed.

This operation happens for each bit. Since it correctly swaps each bit, it will correctly swap the entire number.

16.2 Word Frequencies: Design a method to find the frequency of occurrences of any given word in a book. What if we were running this algorithm multiple times?

pg 181

SOLUTION

Let's start with the simple case.

Solution: Single Query

In this case, we simply go through the book, word by word, and count the number of times that a word appears. This will take $O(n)$ time. We know we can't do better than that since we must look at every word in the book.

```
1  int getFrequency(String[] book, String word) {
2    word = word.trim().toLowerCase();
3    int count = 0;
4    for (String w : book) {
5      if (w.trim().toLowerCase().equals(word)) {
6        count++;
7      }
8    }
9    return count;
10 }
```

We have also converted the string to lowercase and trimmed it. You can discuss with your interviewer if this is necessary (or even desired).

Solution: Repetitive Queries

If we're doing the operation repeatedly, then we can probably afford to take some time and extra memory to do pre-processing on the book. We can create a hash table which maps from a word to its frequency. The frequency of any word can be easily looked up in $O(1)$ time. The code for this is below.

```
1  HashMap<String, Integer> setupDictionary(String[] book) {
2    HashMap<String, Integer> table =
```

```
3         new HashMap<String, Integer>();
4       for (String word : book) {
5         word = word.toLowerCase();
6         if (word.trim() != "") {
7           if (!table.containsKey(word)) {
8             table.put(word, 0);
9           }
10          table.put(word, table.get(word) + 1);
11        }
12      }
13      return table;
14  }
15
16  int getFrequency(HashMap<String, Integer> table, String word) {
17    if (table == null || word == null) return -1;
18    word = word.toLowerCase();
19    if (table.containsKey(word)) {
20      return table.get(word);
21    }
22    return 0;
23  }
```

Note that a problem like this is actually relatively easy. Thus, the interviewer is going to be looking heavily at how careful you are. Did you check for error conditions?

16.3 Intersection: Given two straight line segments (represented as a start point and an end point), compute the point of intersection, if any.

pg 181

SOLUTION

We first need to think about what it means for two line segments to intersect.

For two infinite lines to intersect, they only have to have different slopes. If they have the same slope, then they must be the exact same line (same y-intercept). That is:

```
slope 1 != slope 2
OR
slope 1 == slope 2 AND intersect 1 == intersect 2
```

For two straight lines to intersect, the condition above must be true, *plus* the point of intersection must be within the ranges of each line segment.

```
extended infinite segments intersect
AND
intersection is within line segment 1 (x and y coordinates)
AND
intersection is within line segment 2 (x and y coordinates)
```

What if the two segments represent the same infinite line? In this case, we have to ensure that some portion of their segments overlap. If we order the line segments by their x locations (start is before end, point 1 is before point 2), then an intersection occurs only if:

```
Assume:
    start1.x < start2.x && start1.x < end1.x && start2.x < end2.x
Then intersection occurs if:
    start2 is between start1 and end1
```

We can now go ahead and implement this algorithm.

```
Point intersection(Point start1, Point end1, Point start2, Point end2) {
   /* Rearranging these so that, in order of x values: start is before end and
    * point 1 is before point 2. This will make some of the later logic simpler. */
   if (start1.x > end1.x) swap(start1, end1);
   if (start2.x > end2.x) swap(start2, end2);
   if (start1.x > start2.x) {
      swap(start1, start2);
      swap(end1, end2);
   }

   /* Compute lines (including slope and y-intercept). */
   Line line1 = new Line(start1, end1);
   Line line2 = new Line(start2, end2);

   /* If the lines are parallel, they intercept only if they have the same y
    * intercept and start 2 is on line 1. */
   if (line1.slope == line2.slope) {
      if (line1.yintercept == line2.yintercept &&
          isBetween(start1, start2, end1)) {
         return start2;
      }
      return null;
   }

   /* Get intersection coordinate. */
   double x = (line2.yintercept - line1.yintercept) / (line1.slope - line2.slope);
   double y = x * line1.slope + line1.yintercept;
   Point intersection = new Point(x, y);

   /* Check if within line segment range. */
   if (isBetween(start1, intersection, end1) &&
       isBetween(start2, intersection, end2)) {
      return intersection;
   }
   return null;
}

/* Checks if middle is between start and end. */
boolean isBetween(double start, double middle, double end) {
   if (start > end) {
      return end <= middle && middle <= start;
   } else {
      return start <= middle && middle <= end;
   }
}

/* Checks if middle is between start and end. */
boolean isBetween(Point start, Point middle, Point end) {
   return isBetween(start.x, middle.x, end.x) &&
          isBetween(start.y, middle.y, end.y);
}

/* Swap coordinates of point one and two. */
void swap(Point one, Point two) {
   double x = one.x;
   double y = one.y;
```

```
57    one.setLocation(two.x, two.y);
58    two.setLocation(x, y);
59  }
60
61  public class Line {
62    public double slope, yintercept;
63
64    public Line(Point start, Point end) {
65      double deltaY = end.y - start.y;
66      double deltaX = end.x - start.x;
67      slope = deltaY / deltaX; // Will be Infinity (not exception) when deltaX = 0
68      yintercept = end.y - slope * end.x;
69    }
70
71  public class Point {
72    public double x, y;
73    public Point(double x, double y) {
74      this.x = x;
75      this.y = y;
76    }
77
78    public void setLocation(double x, double y) {
79      this.x = x;
80      this.y = y;
81    }
82  }
```

For simplicity and compactness (it really makes the code easier to read), we've chosen to make the variables within `Point` and `Line` `public`. You can discuss with your interviewer the advantages and disadvantages of this choice.

16.4 Tic Tac Win: Design an algorithm to figure out if someone has won a game of tic-tac-toe.

pg 181

SOLUTION

At first glance, this problem seems really straightforward. We're just checking a tic-tac-toe board; how hard could it be? It turns out that the problem is a bit more complex, and there is no single "perfect" answer. The optimal solution depends on your preferences.

There are a few major design decisions to consider:

1. Will `hasWon` be called just once or many times (for instance, as part of a tic-tac-toe website)? If the latter is the case, we may want to add pre-processing time to optimize the runtime of `hasWon`.
2. Do we know the last move that was made?
3. Tic-tac-toe is usually on a 3x3 board. Do we want to design for just that, or do we want to implement it as an `NxN` solution?
4. In general, how much do we prioritize compactness of code versus speed of execution vs. clarity of code? Remember: The most efficient code may not always be the best. Your ability to understand and maintain the code matters, too.

Solution #1: If hasWon is called many times

There are only 3^9, or about 20,000, tic-tac-toe boards (assuming a 3x3 board). Therefore, we can represent our tic-tac-toe board as an `int`, with each digit representing a piece (0 means Empty, 1 means Red, 2 means Blue). We set up a hash table or array in advance with all possible boards as keys and the value indicating who has won. Our function then is simply this:

```
1  Piece hasWon(int board) {
2    return winnerHashtable[board];
3  }
```

To convert a board (represented by a char array) to an `int`, we can use what is essentially a "base 3" representation. Each board is represented as $3^0v_0 + 3^1v_1 + 3^2v_2 + \ldots + 3^8v_8$, where v_i is a 0 if the space is empty, a 1 if it's a "blue spot" and a 2 if it's a "red spot."

```
1  enum Piece { Empty, Red, Blue };
2
3  int convertBoardToInt(Piece[][] board) {
4    int sum = 0;
5    for (int i = 0; i < board.length; i++) {
6      for (int j = 0; j < board[i].length; j++) {
7        /* Each value in enum has an integer associated with it. We
8         * can just use that. */
9        int value = board[i][j].ordinal();
10       sum = sum * 3 + value;
11     }
12   }
13   return sum;
14 }
```

Now looking up the winner of a board is just a matter of looking it up in a hash table.

Of course, if we need to convert a board into this format every time we want to check for a winner, we haven't saved ourselves any time compared with the other solutions. But, if we can store the board this way from the very beginning, then the lookup process will be very efficient.

Solution #2: If we know the last move

If we know the very last move that was made (and we've been checking for a winner up until now), then we only need to check the row, column, and diagonal that overlaps with this position.

```
1  Piece hasWon(Piece[][] board, int row, int column) {
2    if (board.length != board[0].length) return Piece.Empty;
3
4    Piece piece = board[row][column];
5
6    if (piece == Piece.Empty) return Piece.Empty;
7
8    if (hasWonRow(board, row) || hasWonColumn(board, column)) {
9      return piece;
10   }
11
12   if (row == column && hasWonDiagonal(board, 1)) {
13     return piece;
14   }
15
16   if (row == (board.length - column - 1) && hasWonDiagonal(board, -1)) {
17     return piece;
18   }
```

```
19
20    return Piece.Empty;
21  }
22
23  boolean hasWonRow(Piece[][] board, int row) {
24    for (int c = 1; c < board[row].length; c++) {
25      if (board[row][c] != board[row][0]) {
26        return false;
27      }
28    }
29    return true;
30  }
31
32  boolean hasWonColumn(Piece[][] board, int column) {
33    for (int r = 1; r < board.length; r++) {
34      if (board[r][column] != board[0][column]) {
35        return false;
36      }
37    }
38    return true;
39  }
40
41  boolean hasWonDiagonal(Piece[][] board, int direction) {
42    int row = 0;
43    int column = direction == 1 ? 0 : board.length - 1;
44    Piece first = board[0][column];
45    for (int i = 0; i < board.length; i++) {
46      if (board[row][column] != first) {
47        return false;
48      }
49      row += 1;
50      column += direction;
51    }
52    return true;
53  }
```

There is actually a way to clean up this code to remove some of the duplicated code. We'll see this approach in a later function.

Solution #3: Designing for just a 3x3 board

If we really only want to implement a solution for a 3x3 board, the code is relatively short and simple. The only complex part is trying to be clean and organized, without writing too much duplicated code.

The code below checks each row, column, and diagonal to see if there is a winner.

```
1   Piece hasWon(Piece[][] board) {
2     for (int i = 0; i < board.length; i++) {
3       /* Check Rows */
4       if (hasWinner(board[i][0], board[i][1], board[i][2])) {
5         return board[i][0];
6       }
7
8       /* Check Columns */
9       if (hasWinner(board[0][i], board[1][i], board[2][i])) {
10        return board[0][i];
11      }
```

```
    }

    /* Check Diagonal */
    if (hasWinner(board[0][0], board[1][1], board[2][2])) {
      return board[0][0];
    }

    if (hasWinner(board[0][2], board[1][1], board[2][0])) {
      return board[0][2];
    }

    return Piece.Empty;
  }

  boolean hasWinner(Piece p1, Piece p2, Piece p3) {
    if (p1 == Piece.Empty) {
      return false;
    }
    return p1 == p2 && p2 == p3;
  }
```

This is an okay solution in that it's relatively easy to understand what is going on. The problem is that the values are hard coded. It's easy to accidentally type the wrong indices.

Additionally, it won't be easy to scale this to an NxN board.

Solution #4: Designing for an NxN board

There are a number of ways to implement this on an NxN board.

Nested For-Loops

The most obvious way is through a series of nested for-loops.

```
Piece hasWon(Piece[][] board) {
  int size = board.length;
  if (board[0].length != size) return Piece.Empty;
  Piece first;

  /* Check rows. */
  for (int i = 0; i < size; i++) {
    first = board[i][0];
    if (first == Piece.Empty) continue;
    for (int j = 1; j < size; j++) {
      if (board[i][j] != first) {
        break;
      } else if (j == size - 1) { // Last element
        return first;
      }
    }
  }

  /* Check columns. */
  for (int i = 0; i < size; i++) {
    first = board[0][i];
    if (first == Piece.Empty) continue;
    for (int j = 1; j < size; j++) {
      if (board[j][i] != first) {
```

```
            break;
         } else if (j == size - 1) { // Last element
            return first;
         }
      }
   }

   /* Check diagonals. */
   first = board[0][0];
   if (first != Piece.Empty) {
      for (int i = 1; i < size; i++) {
         if (board[i][i] != first) {
            break;
         } else if (i == size - 1) { // Last element
            return first;
         }
      }
   }

   first = board[0][size - 1];
   if (first != Piece.Empty) {
      for (int i = 1; i < size; i++) {
         if (board[i][size - i - 1] != first) {
            break;
         } else if (i == size - 1) { // Last element
            return first;
         }
      }
   }

   return Piece.Empty;
}
```

This is, to the say the least, pretty ugly. We're doing nearly the same work each time. We should look for a way of reusing the code.

Increment and Decrement Function

One way that we can reuse the code better is to just pass in the values to another function that increments/decrements the rows and columns. The `hasWon` function now just needs the starting position and the amount to increment the row and column by.

```
class Check {
   public int row, column;
   private int rowIncrement, columnIncrement;
   public Check(int row, int column, int rowI, int colI) {
      this.row = row;
      this.column = column;
      this.rowIncrement = rowI;
      this.columnIncrement = colI;
   }

   public void increment() {
      row += rowIncrement;
      column += columnIncrement;
   }

   public boolean inBounds(int size) {
```

```
17        return row >= 0 && column >= 0 && row < size && column < size;
18     }
19  }
20
21  Piece hasWon(Piece[][] board) {
22     if (board.length != board[0].length) return Piece.Empty;
23     int size = board.length;
24
25     /* Create list of things to check. */
26     ArrayList<Check> instructions = new ArrayList<Check>();
27     for (int i = 0; i < board.length; i++) {
28        instructions.add(new Check(0, i, 1, 0));
29        instructions.add(new Check(i, 0, 0, 1));
30     }
31     instructions.add(new Check(0, 0, 1, 1));
32     instructions.add(new Check(0, size - 1, 1, -1));
33
34     /* Check them. */
35     for (Check instr : instructions) {
36        Piece winner = hasWon(board, instr);
37        if (winner != Piece.Empty) {
38              return winner;
39        }
40     }
41     return Piece.Empty;
42  }
43
44  Piece hasWon(Piece[][] board, Check instr) {
45     Piece first = board[instr.row][instr.column];
46     while (instr.inBounds(board.length)) {
47        if (board[instr.row][instr.column] != first) {
48           return Piece.Empty;
49        }
50        instr.increment();
51     }
52     return first;
53  }
```

The Check function is essentially operating as an iterator.

Iterator

Another way of doing it is, of course, to actually build an iterator.

```
1   Piece hasWon(Piece[][] board) {
2      if (board.length != board[0].length) return Piece.Empty;
3      int size = board.length;
4
5      ArrayList<PositionIterator> instructions = new ArrayList<PositionIterator>();
6      for (int i = 0; i < board.length; i++) {
7         instructions.add(new PositionIterator(new Position(0, i), 1, 0, size));
8         instructions.add(new PositionIterator(new Position(i, 0), 0, 1, size));
9      }
10     instructions.add(new PositionIterator(new Position(0, 0), 1, 1, size));
11     instructions.add(new PositionIterator(new Position(0, size - 1), 1, -1, size));
12
13     for (PositionIterator iterator : instructions) {
14        Piece winner = hasWon(board, iterator);
```

```
      if (winner != Piece.Empty) {
        return winner;
      }
    }
    return Piece.Empty;
  }

  Piece hasWon(Piece[][] board, PositionIterator iterator) {
    Position firstPosition = iterator.next();
    Piece first = board[firstPosition.row][firstPosition.column];
    while (iterator.hasNext()) {
      Position position = iterator.next();
      if (board[position.row][position.column] != first) {
        return Piece.Empty;
      }
    }
    return first;
  }

  class PositionIterator implements Iterator<Position> {
    private int rowIncrement, colIncrement, size;
    private Position current;

    public PositionIterator(Position p, int rowIncrement,
                            int colIncrement, int size) {
      this.rowIncrement = rowIncrement;
      this.colIncrement = colIncrement;
      this.size = size;
      current = new Position(p.row - rowIncrement, p.column - colIncrement);
    }

    @Override
    public boolean hasNext() {
      return current.row + rowIncrement < size &&
             current.column + colIncrement < size;
    }

    @Override
    public Position next() {
      current = new Position(current.row + rowIncrement,
                             current.column + colIncrement);
      return current;
    }
  }

  public class Position {
    public int row, column;
    public Position(int row, int column) {
      this.row = row;
      this.column = column;
    }
  }
```

All of this is potentially overkill, but it's worth discussing the options with your interviewer. The point of this problem is to assess your understanding of how to code in a clean and maintainable way.

16.5 Factorial Zeros: Write an algorithm which computes the number of trailing zeros in n factorial.

pg 181

SOLUTION

A simple approach is to compute the factorial, and then count the number of trailing zeros by continuously dividing by ten. The problem with this though is that the bounds of an `int` would be exceeded very quickly. To avoid this issue, we can look at this problem mathematically.

Consider a factorial like 19!:

```
19! = 1*2*3*4*5*6*7*8*9*10*11*12*13*14*15*16*17*18*19
```

A trailing zero is created with multiples of 10, and multiples of 10 are created with pairs of 5-multiples and 2-multiples.

For example, in 19!, the following terms create the trailing zeros:

```
19! = 2 * ... * 5 * ... * 10 * ... * 15 * 16 * ...
```

Therefore, to count the number of zeros, we only need to count the pairs of multiples of 5 and 2. There will always be more multiples of 2 than 5, though, so simply counting the number of multiples of 5 is sufficient.

One "gotcha" here is 15 contributes a multiple of 5 (and therefore one trailing zero), while 25 contributes two (because `25 = 5 * 5`).

There are two different ways to write this code.

The first way is to iterate through all the numbers from 2 through n, counting the number of times that 5 goes into each number.

```
1  /* If the number is a 5 of five, return which power of 5. For example: 5 -> 1,
2   * 25-> 2, etc. */
3  int factorsOf5(int i) {
4     int count = 0;
5     while (i % 5 == 0) {
6        count++;
7        i /= 5;
8     }
9     return count;
10 }
11
12 int countFactZeros(int num) {
13    int count = 0;
14    for (int i = 2; i <= num; i++) {
15       count += factorsOf5(i);
16    }
17    return count;
18 }
```

This isn't bad, but we can make it a little more efficient by directly counting the factors of 5. Using this approach, we would first count the number of multiples of 5 between 1 and n (which is $n/5$), then the number of multiples of 25 ($n/25$), then 125, and so on.

To count how many multiples of `m` are in n, we can just divide n by `m`.

```
1  int countFactZeros(int num) {
2     int count = 0;
3     if (num < 0) {
4        return -1;
5     }
```

```
6      for (int i = 5; num / i > 0; i *= 5) {
7         count += num / i;
8      }
9      return count;
10  }
```

This problem is a bit of a brainteaser, but it can be approached logically (as shown above). By thinking through what exactly will contribute a zero, you can come up with a solution. You should be very clear in your rules upfront so that you can implement it correctly.

16.6 Smallest Difference: Given two arrays of integers, compute the pair of values (one value in each array) with the smallest (non-negative) difference. Return the difference.

EXAMPLE

Input: {1, 3, 15, 11, 2}, {23, 127, 235, 19, 8}

Output: 3. That is, the pair (11, 8).

pg 181

SOLUTION

Let's start first with a brute force solution.

Brute Force

The simple brute force way is to just iterate through all pairs, compute the difference, and compare it to the current minimum difference.

```
1   int findSmallestDifference(int[] array1, int[] array2) {
2      if (array1.length == 0 || array2.length == 0) return -1;
3
4      int min = Integer.MAX_VALUE;
5      for (int i = 0; i < array1.length; i++) {
6         for (int j = 0; j < array2.length; j++) {
7            if (Math.abs(array1[i] - array2[j]) < min) {
8               min = Math.abs(array1[i] - array2[j]);
9            }
10        }
11     }
12     return min;
13  }
```

One minor optimization we could perform from here is to return immediately if we find a difference of zero, since this is the smallest difference possible. However, depending on the input, this might actually be slower.

This will only be faster if there's a pair with difference zero early in the list of pairs. But to add this optimization, we need to execute an additional line of code each time. There's a tradeoff here; it's faster for some inputs and slower for others. Given that it adds complexity in reading the code, it may be best to leave it out.

With or without this "optimization," the algorithm will take $O(AB)$ time.

Optimal

A more optimal approach is to sort the arrays. Once the arrays are sorted, we can find the minimum difference by iterating through the array.

Consider the following two arrays:

```
A: {1, 2, 11, 15}
B: {4, 12, 19, 23, 127, 235}
```

Try the following approach:

1. Suppose a pointer a points to the beginning of A and a pointer b points to the beginning of B. The current difference between a and b is 3. Store this as the `min`.
2. How can we (potentially) make this difference smaller? Well, the value at b is bigger than the value at a, so moving b will only make the difference larger. Therefore, we want to move a.
3. Now a points to 2 and b (still) points to 4. This difference is 2, so we should update `min`. Move a, since it is smaller.
4. Now a points to 11 and b points to 4. Move b.
5. Now a points to 11 and b points to 12. Update `min` to 1. Move b.

And so on.

```
1  int findSmallestDifference(int[] array1, int[] array2) {
2    Arrays.sort(array1);
3    Arrays.sort(array2);
4    int a = 0;
5    int b = 0;
6    int difference = Integer.MAX_VALUE;
7    while (a < array1.length && b < array2.length) {
8      if (Math.abs(array1[a] - array2[b]) < difference) {
9        difference = Math.abs(array1[a] - array2[b]);
10     }
11
12     /* Move smaller value. */
13     if (array1[a] < array2[b]) {
14       a++;
15     } else {
16       b++;
17     }
18   }
19   return difference;
20 }
```

This algorithm takes `O(A log A + B log B)` time to sort and `O(A + B)` time to find the minimum difference. Therefore, the overall runtime is `O(A log A + B log B)`.

16.7 Number Max: Write a method that finds the maximum of two numbers. You should not use if-else or any other comparison operator.

pg 181

SOLUTION

A common way of implementing a `max` function is to look at the sign of `a - b`. In this case, we can't use a comparison operator on this sign, but we *can* use multiplication.

Let k equal the sign of `a - b` such that if `a - b >= 0`, then k is 1. Else, `k = 0`. Let q be the inverse of k.

We can then implement the code as follows:

```
1  /* Flips a 1 to a 0 and a 0 to a 1 */
2  int flip(int bit) {
```

```
3     return 1^bit;
4   }
5
6   /* Returns 1 if a is positive, and 0 if a is negative */
7   int sign(int a) {
8     return flip((a >> 31) & 0x1);
9   }
10
11  int getMaxNaive(int a, int b) {
12    int k = sign(a - b);
13    int q = flip(k);
14    return a * k + b * q;
15  }
```

This code almost works. It fails, unfortunately, when a - b overflows. Suppose, for example, that a is INT_MAX - 2 and b is -15. In this case, a - b will be greater than INT_MAX and will overflow, resulting in a negative value.

We can implement a solution to this problem by using the same approach. Our goal is to maintain the condition where k is 1 when a > b. We will need to use more complex logic to accomplish this.

When does a - b overflow? It will overflow only when a is positive and b is negative, or the other way around. It may be difficult to specially detect the overflow condition, but we *can* detect when a and b have different signs. Note that if a and b have different signs, then we want k to equal `sign(a)`.

The logic looks like:

```
1   if a and b have different signs:
2     // if a > 0, then b < 0, and k = 1.
3     // if a < 0, then b > 0, and k = 0.
4     // so either way, k = sign(a)
5     let k = sign(a)
6   else
7     let k = sign(a - b) // overflow is impossible
```

The code below implements this, using multiplication instead of if-statements.

```
1   int getMax(int a, int b) {
2     int c = a - b;
3
4     int sa = sign(a); // if a >= 0, then 1 else 0
5     int sb = sign(b); // if b >= 0, then 1 else 0
6     int sc = sign(c); // depends on whether or not a - b overflows
7
8     /* Goal: define a value k which is 1 if a > b and 0 if a < b.
9      * (if a = b, it doesn't matter what value k is) */
10
11    // If a and b have different signs, then k = sign(a)
12    int use_sign_of_a = sa ^ sb;
13
14    // If a and b have the same sign, then k = sign(a - b)
15    int use_sign_of_c = flip(sa ^ sb);
16
17    int k = use_sign_of_a * sa + use_sign_of_c * sc;
18    int q = flip(k); // opposite of k
19
20    return a * k + b * q;
21  }
```

Note that for clarity, we split up the code into many different methods and variables. This is certainly not the most compact or efficient way to write it, but it does make what we're doing much cleaner.

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

pg 182

SOLUTION

This is not an especially challenging problem, but it is a somewhat tedious one. The key is to be organized in how you approach the problem—and to make sure you have good test cases.

We can think about converting a number like 19,323,984 as converting each of three 3-digit segments of the number, and inserting "thousands" and "millions" in between as appropriate. That is,

```
convert(19,323,984) = convert(19) + " million " + convert(323) + " thousand " +
                      convert(984)
```

The code below implements this algorithm.

```
1  String[] smalls = {"Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven",
2    "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen",
3    "Sixteen", "Seventeen", "Eighteen", "Nineteen"};
4  String[] tens = {"", "", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy",
5    "Eighty", "Ninety"};
6  String[] bigs = {"", "Thousand", "Million", "Billion"};
7  String hundred = "Hundred";
8  String negative = "Negative";
9
10 String convert(int num) {
11   if (num == 0) {
12     return smalls[0];
13   } else if (num < 0) {
14     return negative + " " + convert(-1 * num);
15   }
16
17   LinkedList<String> parts = new LinkedList<String>();
18   int chunkCount = 0;
19
20   while (num > 0) {
21     if (num % 1000 != 0) {
22       String chunk = convertChunk(num % 1000) + " " + bigs[chunkCount];
23       parts.addFirst(chunk);
24     }
25     num /= 1000; // shift chunk
26     chunkCount++;
27   }
28
29   return listToString(parts);
30 }
31
32 String convertChunk(int number) {
33   LinkedList<String> parts = new LinkedList<String>();
34
35   /* Convert hundreds place */
36   if (number >= 100) {
37     parts.addLast(smalls[number / 100]);
```

```
38        parts.addLast(hundred);
39        number %= 100;
40     }
41
42     /* Convert tens place */
43     if (number >= 10 && number <= 19) {
44        parts.addLast(smalls[number]);
45     } else if (number >= 20) {
46        parts.addLast(tens[number / 10]);
47        number %= 10;
48     }
49
50     /* Convert ones place */
51     if (number >= 1 && number <= 9) {
52        parts.addLast(smalls[number]);
53     }
54
55     return listToString(parts);
56  }
57  /* Convert a linked list of strings to a string, dividing it up with spaces. */
58  String listToString(LinkedList<String> parts) {
59     StringBuilder sb = new StringBuilder();
60     while (parts.size() > 1) {
61        sb.append(parts.pop());
62        sb.append(" ");
63     }
64     sb.append(parts.pop());
65     return sb.toString();
66  }
```

The key in a problem like this is to make sure you consider all the special cases. There are a lot of them.

16.9 Operations: Write methods to implement the multiply, subtract, and divide operations for integers. The results of all of these are integers. Use only the add operator.

pg 182

SOLUTION

The only operation we have to work with is the add operator. In each of these problems, it's useful to think in depth about what these operations really do or how to phrase them in terms of other operations (either add or operations we've already completed).

Subtraction

How can we phrase subtraction in terms of addition? This one is pretty straightforward. The operation `a - b` is the same thing as `a + (-1) * b`. However, because we are not allowed to use the * (multiply) operator, we must implement a `negate` function.

```
1  /* Flip a positive sign to negative or negative sign to pos. */
2  int negate(int a) {
3     int neg = 0;
4     int newSign = a < 0 ? 1 : -1;
5     while (a != 0) {
6        neg += newSign;
7        a += newSign;
8     }
```

```
9     return neg;
10 }
11
12 /* Subtract two numbers by negating b and adding them */
13 int minus(int a, int b) {
14    return a + negate(b);
15 }
```

The negation of the value k is implemented by adding -1 k times. Observe that this will take O(k) time.

If optimizing is something we value here, we can try to get a to zero faster. (For this explanation, we'll assume that a is positive.) To do this, we can first reduce a by 1, then 2, then 4, then 8, and so on. We'll call this value `delta`. We want a to reach exactly zero. When reducing a by the next `delta` would change the sign of a, we reset `delta` back to 1 and repeat the process.

For example:

```
a:     29   28   26   22   14   13   11    7    6    4    0
delta:  -1   -2   -4   -8   -1   -2   -4   -1   -2   -4
```

The code below implements this algorithm.

```
1  int negate(int a) {
2     int neg = 0;
3     int newSign = a < 0 ? 1 : -1;
4     int delta = newSign;
5     while (a != 0) {
6        boolean differentSigns = (a + delta > 0) != (a > 0);
7        if (a + delta != 0 && differentSigns) { // If delta is too big, reset it.
8           delta = newSign;
9        }
10       neg += delta;
11       a += delta;
12       delta += delta; // Double the delta
13    }
14    return neg;
15 }
```

Figuring out the runtime here takes a bit of calculation.

Observe that reducing a by half takes O(log a) work. Why? For each round of "reduce a by half", the absolute values of a and `delta` always add up to the same number. The values of `delta` and a will converge at $^{a}/_{2}$. Since `delta` is being doubled each time, it will take O(log a) steps to reach half of a.

We do O(log a) rounds.

1. Reducing a to $^{a}/_{2}$ takes O(log a) time.
2. Reducing $^{a}/_{2}$ to $^{a}/_{4}$ takes O(log $^{a}/_{2}$) time.
3. Reducing $^{a}/_{4}$ to $^{a}/_{8}$ takes O(log $^{a}/_{4}$) time.

... As so on, for O(log a) rounds.

The runtime therefore is O(log a + log($^{a}/_{2}$) + log($^{a}/_{4}$) + ...), with O(log a) terms in the expression.

Recall two rules of logs:

- log(xy) = log x + log y
- log($^{x}/_{y}$) = log x - log y.

If we apply this to the above expression, we get:

1. $O(\log a + \log(a/2) + \log(a/4) + \ldots)$
2. $O(\log a + (\log a - \log 2) + (\log a - \log 4) + (\log a - \log 8) + \ldots$
3. $O((\log a)*(\log a) - (\log 2 + \log 4 + \log 8 + \ldots + \log a))$ // $O(\log a)$ terms
4. $O((\log a)*(\log a) - (1 + 2 + 3 + \ldots + \log a))$ // computing the values of logs
5. $O((\log a)*(\log a) - \frac{(\log a)(1+\log a)}{2})$ // apply equation for sum of 1 through k
6. $O((\log a)^2)$ // drop second term from step 5

Therefore, the runtime is $O((\log a)^2)$.

This math is considerably more complicated than most people would be able to do (or expected to do) in an interview. You could make a simplification: You do $O(\log a)$ rounds and the longest round takes $O(\log a)$ work. Therefore, as an upper bound, `negate` takes $O((\log a)^2)$ time. In this case, the upper bound happens to be the true time.

There are some faster solutions too. For example, rather than resetting `delta` to 1 at each round, we could change `delta` to its previous value. This would have the effect of `delta` "counting up" by multiples of two, and then "counting down" by multiples of two. The runtime of this approach would be $O(\log a)$. However, this implementation would require a stack, division, or bit shifting—any of which might violate the spirit of the problem. You could certainly discuss those implementations with your interviewer though.

Multiplication

The connection between addition and multiplication is equally straightforward. To multiply a by b, we just add a to itself b times.

```
1  /* Multiply a by b by adding a to itself b times */
2  int multiply(int a, int b) {
3    if (a < b) {
4      return multiply(b, a); // algorithm is faster if b < a
5    }
6    int sum = 0;
7    for (int i = abs(b); i > 0; i = minus(i, 1)) {
8      sum += a;
9    }
10   if (b < 0) {
11     sum = negate(sum);
12   }
13   return sum;
14 }
15
16 /* Return absolute value */
17 int abs(int a) {
18   if (a < 0) {
19     return negate(a);
20   } else {
21     return a;
22   }
23 }
```

The one thing we need to be careful of in the above code is to properly handle multiplication of negative numbers. If b is negative, we need to flip the value of `sum`. So, what this code really does is:

`multiply(a, b) <-- abs(b) * a * (-1 if b < 0).`

We also implemented a simple `abs` function to help.

Division

Of the three operations, division is certainly the hardest. The good thing is that we can use the `multiply`, `subtract`, and `negate` methods now to implement `divide`.

We are trying to compute x where $x = a/b$. Or, to put this another way, find x where $a = bx$. We've now changed the problem into one that can be stated with something we know how to do: multiplication.

We could implement this by multiplying b by progressively higher values, until we reach a. That would be fairly inefficient, particularly given that our implementation of `multiply` involves a lot of adding.

Alternatively, we can look at the equation $a = xb$ to see that we can compute x by adding b to itself repeatedly until we reach a. The number of times we need to do that will equal x.

Of course, a might not be evenly divisible by b, and that's okay. Integer division, which is what we've been asked to implement, is supposed to truncate the result.

The code below implements this algorithm.

```
1  int divide(int a, int b) throws java.lang.ArithmeticException {
2    if (b == 0) {
3      throw new java.lang.ArithmeticException("ERROR");
4    }
5    int absa = abs(a);
6    int absb = abs(b);
7
8    int product = 0;
9    int x = 0;
10   while (product + absb <= absa) { /* don't go past a */
11     product += absb;
12     x++;
13   }
14
15   if ((a < 0 && b < 0) || (a > 0 && b > 0)) {
16     return x;
17   } else {
18     return negate(x);
19   }
20 }
```

In tackling this problem, you should be aware of the following:

- A logical approach of going back to what exactly multiplication and division do comes in handy. Remember that. All (good) interview problems can be approached in a logical, methodical way!
- The interviewer is looking for this sort of logical work-your-way-through-it approach.
- This is a great problem to demonstrate your ability to write clean code—specifically, to show your ability to reuse code. For example, if you were writing this solution and didn't put `negate` in its own method, you should move it into its own method once you see that you'll use it multiple times.
- Be careful about making assumptions while coding. Don't assume that the numbers are all positive or that a is bigger than b.

16.10 Living People: Given a list of people with their birth and death years, implement a method to compute the year with the most number of people alive. You may assume that all people were born between 1900 and 2000 (inclusive). If a person was alive during any portion of that year, they should be included in that year's count. For example, Person (birth = 1908, death = 1909) is included in the counts for both 1908 and 1909.

pg 182

SOLUTION

The first thing we should do is outline what this solution will look like. The interview question hasn't specified the exact form of input. In a real interview, we could ask the interviewer how the input is structured. Alternatively, you can explicitly state your (reasonable) assumptions.

Here, we'll need to make our own assumptions. We will assume that we have an array of simple `Person` objects:

```
1  public class Person {
2    public int birth;
3    public int death;
4    public Person(int birthYear, int deathYear) {
5      birth = birthYear;
6      death = deathYear;
7    }
8  }
```

We could have also given `Person` a `getBirthYear()` and `getDeathYear()` objects. Some would argue that's better style, but for compactness and clarity, we'll just keep the variables public.

The important thing here is to actually use a `Person` object. This shows better style than, say, having an integer array for birth years and an integer array for death years (with an implicit association of `births[i]` and `deaths[i]` being associated with the same person). You don't get a lot of chances to demonstrate great coding style, so it's valuable to take the ones you get.

With that in mind, let's start with a brute force algorithm.

Brute Force

The brute force algorithm falls directly out from the wording of the problem. We need to find the year with the most number of people alive. Therefore, we go through each year and check how many people are alive in that year.

```
1  int maxAliveYear(Person[] people, int min, int max) {
2    int maxAlive = 0;
3    int maxAliveYear = min;
4
5    for (int year = min; year <= max; year++) {
6      int alive = 0;
7      for (Person person : people) {
8        if (person.birth <= year && year <= person.death) {
9          alive++;
10       }
11     }
12     if (alive > maxAlive) {
13       maxAlive = alive;
14       maxAliveYear = year;
15     }
16   }
```

```
17
18    return maxAliveYear;
19  }
```

Note that we have passed in the values for the min year (1900) and max year (2000). We shouldn't hard code these values.

The runtime of this is O(RP), where R is the range of years (100 in this case) and P is the number of people.

Slightly Better Brute Force

A slightly better way of doing this is to create an array where we track the number of people born in each year. Then, we iterate through the list of people and increment the array for each year they are alive.

```
1   int maxAliveYear(Person[] people, int min, int max) {
2     int[] years = createYearMap(people, min, max);
3     int best = getMaxIndex(years);
4     return best + min;
5   }
6
7   /* Add each person's years to a year map. */
8   int[] createYearMap(Person[] people, int min, int max) {
9     int[] years = new int[max - min + 1];
10    for (Person person : people) {
11      incrementRange(years, person.birth - min, person.death - min);
12    }
13    return years;
14  }
15
16  /* Increment array for each value between left and right. */
17  void incrementRange(int[] values, int left, int right) {
18    for (int i = left; i <= right; i++) {
19      values[i]++;
20    }
21  }
22
23  /* Get index of largest element in array. */
24  int getMaxIndex(int[] values) {
25    int max = 0;
26    for (int i = 1; i < values.length; i++) {
27      if (values[i] > values[max]) {
28        max = i;
29      }
30    }
31    return max;
32  }
```

Be careful on the size of the array in line 9. If the range of years is 1900 to 2000 inclusive, then that's 101 years, not 100. That is why the array has size `max - min + 1`.

Let's think about the runtime by breaking this into parts.

- We create an R-sized array, where R is the min and max years.
- Then, for P people, we iterate through the years (Y) that the person is alive.
- Then, we iterate through the R-sized array again.

The total runtime is O(PY + R). In the worst case, Y is R and we have done no better than we did in the first algorithm.

More Optimal

Let's create an example. (In fact, an example is really helpful in almost all problems. Ideally, you've already done this.) Each column below is matched, so that the items correspond to the same person. For compactness, we'll just write the last two digits of the year.

```
birth: 12  20  10  01  10  23  13  90  83  75
death: 15  90  98  72  98  82  98  98  99  94
```

It's worth noting that it doesn't really matter whether these years are matched up. Every birth adds a person and every death removes a person.

Since we don't actually need to match up the births and deaths, let's sort both. A sorted version of the years might help us solve the problem.

```
birth: 01  10  10  12  13  20  23  75  83  90
death: 15  72  82  90  94  98  98  98  98  99
```

We can try walking through the years.

- At year 0, no one is alive.
- At year 1, we see one birth.
- At years 2 through 9, nothing happens.
- Let's skip ahead until year 10, when we have two births. We now have three people alive.
- At year 15, one person dies. We are now down to two people alive.
- And so on.

If we walk through the two arrays like this, we can track the number of people alive at each point.

```
1  int maxAliveYear(Person[] people, int min, int max) {
2    int[] births = getSortedYears(people, true);
3    int[] deaths = getSortedYears(people, false);
4
5    int birthIndex = 0;
6    int deathIndex = 0;
7    int currentlyAlive = 0;
8    int maxAlive = 0;
9    int maxAliveYear = min;
10
11   /* Walk through arrays. */
12   while (birthIndex < births.length) {
13     if (births[birthIndex] <= deaths[deathIndex]) {
14       currentlyAlive++; // include birth
15       if (currentlyAlive > maxAlive) {
16         maxAlive = currentlyAlive;
17         maxAliveYear = births[birthIndex];
18       }
19       birthIndex++; // move birth index
20     } else if (births[birthIndex] > deaths[deathIndex]) {
21       currentlyAlive--; // include death
22       deathIndex++; // move death index
23     }
24   }
25
26   return maxAliveYear;
27 }
28
29 /* Copy birth years or death years (depending on the value of copyBirthYear into
```

```
30   * integer array, then sort array. */
31  int[] getSortedYears(Person[] people, boolean copyBirthYear) {
32    int[] years = new int[people.length];
33    for (int i = 0; i < people.length; i++) {
34      years[i] = copyBirthYear ? people[i].birth : people[i].death;
35    }
36    Arrays.sort(years);
37    return years;
38  }
```

There are some very easy things to mess up here.

On line 13, we need to think carefully about whether this should be a less than (<) or a less than or equals (<=). The scenario we need to worry about is that you see a birth and death in the same year. (It doesn't matter whether the birth and death is from the same person.)

When we see a birth and death from the same year, we want to include the birth *before* we include the death, so that we count this person as alive for that year. That is why we use a <= on line 13.

We also need to be careful about where we put the updating of `maxAlive` and `maxAliveYear`. It needs to be after the `currentAlive++`, so that it takes into account the updated total. But it needs to be before `birthIndex++`, or we won't have the right year.

This algorithm will take `O(P log P)` time, where P is the number of people.

More Optimal (Maybe)

Can we optimize this further? To optimize this, we'd need to get rid of the sorting step. We're back to dealing with unsorted values:

```
birth: 12  20  10  01  10  23  13  90  83  75
death: 15  90  98  72  98  82  98  98  99  94
```

Earlier, we had logic that said that a birth is just adding a person and a death is just subtracting a person. Therefore, let's represent the data using the logic:

```
01: +1      10: +1      10: +1      12: +1      13: +1
15: -1      20: +1      23: +1      72: -1      75: +1
82: -1      83: +1      90: +1      90: -1      94: -1
98: -1      98: -1      98: -1      98: -1      99: -1
```

We can create an array of the years, where the value at `array[year]` indicates how the population changed in that year. To create this array, we walk through the list of people and increment when they're born and decrement when they die.

Once we have this array, we can walk through each of the years, tracking the current population as we go (adding the value at `array[year]` each time).

This logic is reasonably good, but we should think about it more. Does it really work?

One edge case we should consider is when a person dies the same year that they're born. The increment and decrement operations will cancel out to give 0 population change. According to the wording of the problem, this person should be counted as living in that year.

In fact, the "bug" in our algorithm is broader than that. This same issue applies to all people. People who die in 1908 shouldn't be removed from the population count until 1909.

There's a simple fix: instead of decrementing `array[deathYear]`, we should decrement `array[deathYear + 1]`.

```
1   int maxAliveYear(Person[] people, int min, int max) {
```

```
    /* Build population delta array. */
    int[] populationDeltas = getPopulationDeltas(people, min, max);
    int maxAliveYear = getMaxAliveYear(populationDeltas);
    return maxAliveYear + min;
}

/* Add birth and death years to deltas array. */
int[] getPopulationDeltas(Person[] people, int min, int max) {
    int[] populationDeltas = new int[max - min + 2];
    for (Person person : people) {
        int birth = person.birth - min;
        populationDeltas[birth]++;

        int death = person.death - min;
        populationDeltas[death + 1]--;
    }
    return populationDeltas;
}

/* Compute running sums and return index with max. */
int getMaxAliveYear(int[] deltas) {
    int maxAliveYear = 0;
    int maxAlive = 0;
    int currentlyAlive = 0;
    for (int year = 0; year < deltas.length; year++) {
        currentlyAlive += deltas[year];
        if (currentlyAlive > maxAlive) {
            maxAliveYear = year;
            maxAlive = currentlyAlive;
        }
    }

    return maxAliveYear;
}
```

This algorithm takes O(R + P) time, where R is the range of years and P is the number of people. Although O(R + P) might be faster than O(P log P) for many expected inputs, you cannot directly compare the speeds to say that one is faster than the other.

16.11 Diving Board: You are building a diving board by placing a bunch of planks of wood end-to-end. There are two types of planks, one of length `shorter` and one of length `longer`. You must use exactly K planks of wood. Write a method to generate all possible lengths for the diving board.

pg 182

SOLUTION

One way to approach this is to think about the choices we make as we're building a diving board. This leads us to a recursive algorithm.

Recursive Solution

For a recursive solution, we can imagine ourselves building a diving board. We make K decisions, each time choosing which plank we will put on next. Once we've put on K planks, we have a complete diving board and we can add this to the list (assuming we haven't seen this length before).

We can follow this logic to write recursive code. Note that we don't need to track the sequence of planks. All we need to know is the current length and the number of planks remaining.

```
1  HashSet<Integer> allLengths(int k, int shorter, int longer) {
2    HashSet<Integer> lengths = new HashSet<Integer>();
3    getAllLengths(k, 0, shorter, longer, lengths);
4    return lengths;
5  }
6
7  void getAllLengths(int k, int total, int shorter, int longer,
8                     HashSet<Integer> lengths) {
9    if (k == 0) {
10     lengths.add(total);
11     return;
12   }
13   getAllLengths(k - 1, total + shorter, shorter, longer, lengths);
14   getAllLengths(k - 1, total + longer, shorter, longer, lengths);
15 }
```

We've added each length to a hash set. This will automatically prevent adding duplicates.

This algorithm takes $O(2^K)$ time, since there are two choices at each recursive call and we recurse to a depth of K.

Memoization Solution

As in many recursive algorithms (especially those with exponential runtimes), we can optimize this through memorization (a form of dynamic programming).

Observe that some of the recursive calls will be essentially equivalent. For example, picking plank 1 and then plank 2 is equivalent to picking plank 2 and then plank 1.

Therefore, if we've seen this `(total, plank count)` pair before then we stop this recursive path. We can do this using a `HashSet` with a key of `(total, plank count)`.

> Many candidates will make a mistake here. Rather than stopping only when they've seen `(total, plank count)`, they'll stop whenever they've seen just `total` before. This is incorrect. Seeing two planks of length 1 is not the same thing as one plank of length 2, because there are different numbers of planks remaining. In memoization problems, be very careful about what you choose for your key.

The code for this approach is very similar to the earlier approach.

```
1  HashSet<Integer> allLengths(int k, int shorter, int longer) {
2    HashSet<Integer> lengths = new HashSet<Integer>();
3    HashSet<String> visited = new HashSet<String>();
4    getAllLengths(k, 0, shorter, longer, lengths, visited);
5    return lengths;
6  }
7
8  void getAllLengths(int k, int total, int shorter, int longer,
9                     HashSet<Integer> lengths, HashSet<String> visited) {
10   if (k == 0) {
11     lengths.add(total);
12     return;
13   }
14   String key = k + " " + total;
```

```
15    if (visited.contains(key)) {
16       return;
17    }
18    getAllLengths(k - 1, total + shorter, shorter, longer, lengths, visited);
19    getAllLengths(k - 1, total + longer, shorter, longer, lengths, visited);
20    visited.add(key);
21 }
```

For simplicity, we've set the key to be a string representation of `total` and the current plank count. Some people may argue it's better to use a data structure to represent this pair. There are benefits to this, but there are drawbacks as well. It's worth discussing this tradeoff with your interviewer.

The runtime of this algorithm is a bit tricky to figure out.

One way we can think about the runtime is by understanding that we're basically filling in a table of SUMS x PLANK COUNTS. The biggest possible sum is K * LONGER and the biggest possible plank count is K. Therefore, the runtime will be no worse than $O(K^2 * LONGER)$.

Of course, a bunch of those sums will never actually be reached. How many unique sums can we get? Observe that any path with the same number of each type of planks will have the same sum. Since we can have at most K planks of each type, there are only K different sums we can make. Therefore, the table is really KxK, and the runtime is $O(K^2)$.

Optimal Solution

If you re-read the prior paragraph, you might notice something interesting. There are only K distinct sums we can get. Isn't that the whole point of the problem—to find all possible sums?

We don't actually need to go through all arrangements of planks. We just need to go through all unique sets of K planks (sets, not orders!). There are only K ways of picking K planks if we only have two possible types: {0 of type A, K of type B}, {1 of type A, K-1 of type B}, {2 of type A, K-2 of type B}, ...

This can be done in just a simple for loop. At each "sequence", we just compute the sum.

```
1  HashSet<Integer> allLengths(int k, int shorter, int longer) {
2     HashSet<Integer> lengths = new HashSet<Integer>();
3     for (int nShorter = 0; nShorter <= k; nShorter++) {
4        int nLonger = k - nShorter;
5        int length = nShorter * shorter + nLonger * longer;
6        lengths.add(length);
7     }
8     return lengths;
9  }
```

We've used a `HashSet` here for consistency with the prior solutions. This isn't really necessary though, since we shouldn't get any duplicates. We could instead use an `ArrayList`. If we do this, though, we just need to handle an edge case where the two types of planks are the same length. In this case, we would just return an `ArrayList` of size 1.