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

1. O(log a + log($a/2$) + log($a/4$) + ...)
2. O(log a + (log a - log 2) + (log a - log 4) + (log a - log 8) + ...
3. O((log a)*(log a) - (log 2 + log 4 + log 8 + ... + log a)) // O(log a) terms
4. O((log a)*(log a) - (1 + 2 + 3 + ... + log a)) // computing the values of logs
5. O((log a)*(log a) - $(\log a)(1+\log a)/2$) // 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.

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

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

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

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

Becomes:

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

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

pg 182

SOLUTION

Since we know the element will be passed in as an `Element` and `Attribute`, our code is reasonably simple. We can implement this by applying a tree-like approach.

We repeatedly call `encode()` on parts of the XML structure, handling the code in slightly different ways depending on the type of the XML element.

```
1  void encode(Element root, StringBuilder sb) {
2    encode(root.getNameCode(), sb);
3    for (Attribute a : root.attributes) {
4      encode(a, sb);
5    }
6    encode("0", sb);
7    if (root.value != null && root.value != "") {
8      encode(root.value, sb);
9    } else {
10     for (Element e : root.children) {
11       encode(e, sb);
12     }
13   }
14   encode("0", sb);
15 }
16
17 void encode(String v, StringBuilder sb) {
18   sb.append(v);
19   sb.append(" ");
20 }
21
22 void encode(Attribute attr, StringBuilder sb) {
23   encode(attr.getTagCode(), sb);
24   encode(attr.value, sb);
25 }
26
```