# 5

# Solutions to Bit Manipulation

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

EXAMPLE

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

Output: N = 10001001100

*pg 115*

**SOLUTION**

This problem can be approached in three key steps:

1. Clear the bits j through i in N
2. Shift M so that it lines up with bits j through i
3. Merge M and N.

The trickiest part is Step 1. How do we clear the bits in N? We can do this with a mask. This mask will have all 1s, except for 0s in the bits j through i. We create this mask by creating the left half of the mask first, and then the right half.

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

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

In a problem like this (and many bit manipulation problems), you should make sure to thoroughly test your code. It's extremely easy to wind up with off-by-one errors.

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

*pg 116*

**SOLUTION**

NOTE: When otherwise ambiguous, we'll use the subscripts $x_2$ and $x_{10}$ to indicate whether x is in base 2 or base 10.

First, let's start off by asking ourselves what a non-integer number in binary looks like. By analogy to a decimal number, the binary number $0.101_2$ would look like:

$$0.101_2 = 1 * 1/2^1 + 0 * 1/2^2 + 1 * 1/2^3.$$

To print the decimal part, we can multiply by 2 and check if 2n is greater than or equal to 1. This is essentially "shifting" the fractional sum. That is:

$$\begin{aligned} r &= 2_{10} * n \\ &= 2_{10} * 0.101_2 \\ &= 1 * 1/2^0 + 0 * 1/2^1 + 1 * 1/2^2 \\ &= 1.01_2 \end{aligned}$$

If r >= 1, then we know that n had a 1 right after the decimal point. By doing this continuously, we can check every digit.

```
1   String printBinary(double num) {
2     if (num >= 1 || num <= 0) {
3       return "ERROR";
4     }
5
6     StringBuilder binary = new StringBuilder();
7     binary.append(".");
8     while (num > 0) {
9       /* Setting a limit on length: 32 characters */
10      if (binary.length() >= 32) {
11        return "ERROR";
12      }
13
14      double r = num * 2;
15      if (r >= 1) {
16        binary.append(1);
17        num = r - 1;
18      } else {
19        binary.append(0);
20        num = r;
21      }
22    }
23    return binary.toString();
24  }
```

Alternatively, rather than multiplying the number by two and comparing it to 1, we can compare the number to .5, then .25, and so on. The code below demonstrates this approach.

```
1  String printBinary2(double num) {
2    if (num >= 1 || num <= 0) {
3      return "ERROR";
4    }
5
6    StringBuilder binary = new StringBuilder();
7    double frac = 0.5;
8    binary.append(".");
9    while (num > 0) {
10     /* Setting a limit on length: 32 characters */
11     if (binary.length() > 32) {
12       return "ERROR";
13     }
14     if (num >= frac) {
15       binary.append(1);
16       num -= frac;
17     } else {
18       binary.append(0);
19     }
20     frac /= 2;
21   }
22   return binary.toString();
23 }
```

Both approaches are equally good; choose the one you feel most comfortable with.

Either way, you should make sure to prepare thorough test cases for this problem—and to actually run through them in your interview.

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

EXAMPLE

Input: 1775 (or: 11011101111)

Output: 8

*pg 116*

**SOLUTION**

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

**Brute Force**

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

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

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

```
2     if (n == -1) return Integer.BYTES * 8;
3     ArrayList<Integer> sequences = getAlternatingSequences(n);
4     return findLongestSequence(sequences);
5   }
6
7   /* Return a list of the sizes of the sequences. The sequence starts off with the
8      number of 0s (which might be 0) and then alternates with the counts of each
9      value.*/
10  ArrayList<Integer> getAlternatingSequences(int n) {
11    ArrayList<Integer> sequences = new ArrayList<Integer>();
12
13    int searchingFor = 0;
14    int counter = 0;
15
16    for (int i = 0; i < Integer.BYTES * 8; i++) {
17      if ((n & 1) != searchingFor) {
18        sequences.add(counter);
19        searchingFor = n & 1; // Flip 1 to 0 or 0 to 1
20        counter = 0;
21      }
22      counter++;
23      n >>>= 1;
24    }
25    sequences.add(counter);
26
27    return sequences;
28  }
29
30  /* Given the lengths of alternating sequences of 0s and 1s, find the longest one
31   * we can build. */
32  int findLongestSequence(ArrayList<Integer> seq) {
33    int maxSeq = 1;
34
35    for (int i = 0; i < seq.size(); i += 2) {
36      int zerosSeq = seq.get(i);
37      int onesSeqRight = i - 1 >= 0 ? seq.get(i - 1) : 0;
38      int onesSeqLeft = i + 1 < seq.size() ? seq.get(i + 1) : 0;
39
40      int thisSeq = 0;
41      if (zerosSeq == 1) { // Can merge
42        thisSeq = onesSeqLeft + 1 + onesSeqRight;
43      } if (zerosSeq > 1) { // Just add a zero to either side
44        thisSeq = 1 + Math.max(onesSeqRight, onesSeqLeft);
45      } else if (zerosSeq == 0) { // No zero, but take either side
46        thisSeq = Math.max(onesSeqRight, onesSeqLeft);
47      }
48      maxSeq = Math.max(thisSeq, maxSeq);
49    }
50
51    return maxSeq;
52  }
```

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

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

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

#### Optimal Algorithm

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

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

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

Update `maxLength` as we go.

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

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

**5.4** **Next Number:** Given a positive integer, print the next smallest and the next largest number that have the same number of 1 bits in their binary representation.

*pg 116*

#### SOLUTION

There are a number of ways to approach this problem, including using brute force, using bit manipulation, and using clever arithmetic. Note that the arithmetic approach builds on the bit manipulation approach. You'll want to understand the bit manipulation approach before going on to the arithmetic one.

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

### The Brute Force Approach

An easy approach is simply brute force: count the number of 1s in n, and then increment (or decrement) until you find a number with the same number of 1s. Easy—but not terribly interesting. Can we do something a bit more optimal? Yes!

Let's start with the code for `getNext`, and then move on to `getPrev`.

### Bit Manipulation Approach for Get Next Number

If we think about what the next number *should* be, we can observe the following. Given the number 13948, the binary representation looks like:

| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

We want to make this number bigger (but not *too* big). We also need to keep the same number of ones.

Observation: Given a number n and two bit locations `i` and `j`, suppose we flip bit `i` from a `1` to a `0`, and bit `j` from a `0` to a `1`. If `i > j`, then n will have decreased. If `i < j`, then n will have increased.

We know the following:

1. If we flip a zero to a one, we must flip a one to a zero.
2. When we do that, the number will be bigger if and only if the zero-to-one bit was to the left of the one-to-zero bit.
3. We want to make the number bigger, but not unnecessarily bigger. Therefore, we need to flip the rightmost zero which has ones on the right of it.

To put this in a different way, we are flipping the rightmost non-trailing zero. That is, using the above example, the trailing zeros are in the 0th and 1st spot. The rightmost non-trailing zero is at bit 7. Let's call this position p.

*Step 1: Flip rightmost non-trailing zero*

| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

With this change, we have increased the size of n. But, we also have one too many ones, and one too few zeros. We'll need to shrink the size of our number as much as possible while keeping that in mind.

We can shrink the number by rearranging all the bits to the right of bit p such that the 0s are on the left and the 1s are on the right. As we do this, we want to replace one of the 1s with a 0.

A relatively easy way of doing this is to count how many ones are to the right of p, clear all the bits from 0 until p, and then add back in `c1-1` ones. Let `c1` be the number of ones to the right of p and `c0` be the number of zeros to the right of p.

Let's walk through this with an example.

*Step 2: Clear bits to the right of p. From before,* `c0 = 2. c1 = 5. p = 7.`

| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

To clear these bits, we need to create a mask that is a sequence of ones, followed by p zeros. We can do this as follows:

```
a = 1 << p; // all zeros except for a 1 at position p.
b = a - 1;    // all zeros, followed by p ones.
mask = ~b;    // all ones, followed by p zeros.
n = n & mask; // clears rightmost p bits.
```

Or, more concisely, we do:

```
n &= ~((1 << p) - 1).
```

*Step 3: Add in* `c1 - 1` *ones.*

| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

To insert `c1 - 1` ones on the right, we do the following:

```
a = 1 << (c1 - 1); // 0s with a 1 at position c1 - 1
b = a - 1;         // 0s with 1s at positions 0 through c1 - 1
n = n | b;         // inserts 1s at positions 0 through c1 - 1
```

Or, more concisely:

```
n |= (1 << (c1 - 1)) - 1;
```

We have now arrived at the smallest number bigger than n with the same number of ones.

The code for `getNext` is below.

```
1   int getNext(int n) {
2     /* Compute c0 and c1 */
3     int c = n;
4     int c0 = 0;
5     int c1 = 0;
6     while (((c & 1) == 0) && (c != 0)) {
7       c0++;
8       c >>= 1;
9     }
10
11    while ((c & 1) == 1) {
12      c1++;
13      c >>= 1;
14    }
15
16    /* Error: if n == 11..1100...00, then there is no bigger number with the same
17     * number of 1s. */
18    if (c0 + c1 == 31 || c0 + c1 == 0) {
19      return -1;
20    }
21
22    int p = c0 + c1; // position of rightmost non-trailing zero
23
24    n |= (1 << p); // Flip rightmost non-trailing zero
25    n &= ~((1 << p) - 1); // Clear all bits to the right of p
26    n |= (1 << (c1 - 1)) - 1; // Insert (c1-1) ones on the right.
```

```
27    return n;
28  }
```

**Bit Manipulation Approach for Get Previous Number**

To implement `getPrev`, we follow a very similar approach.

1. Compute `c0` and `c1`. Note that `c1` is the number of trailing ones, and `c0` is the size of the block of zeros immediately to the left of the trailing ones.
2. Flip the rightmost non-trailing one to a zero. This will be at position `p = c1 + c0`.
3. Clear all bits to the right of bit p.
4. Insert `c1 + 1` ones immediately to the right of position p.

Note that Step 2 sets bit p to a zero and Step 3 sets bits `0` through `p-1` to a zero. We can merge these steps.

Let's walk through this with an example.

*Step 1: Initial Number.* `p = 7. c1 = 2. c0 = 5.`

| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

*Steps 2 & 3: Clear bits 0 through p.*

| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

We can do this as follows:

```
int a = ~0;             // Sequence of 1s
int b = a << (p + 1);   // Sequence of 1s followed by p + 1 zeros.
n &= b;                 // Clears bits 0 through p.
```

*Steps 4: Insert `c1 + 1` ones immediately to the right of position* p.

| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Note that since `p = c1 + c0`, the `(c1 + 1)` ones will be followed by `(c0 - 1)` zeros.

We can do this as follows:

```
int a = 1 << (c1 + 1); // 0s with 1 at position (c1 + 1)
int b = a - 1;         // 0s followed by c1 + 1 ones
int c = b << (c0 - 1); // c1+1 ones followed by c0-1 zeros.
n |= c;
```

The code to implement this is below.

```
1   int getPrev(int n) {
2     int temp = n;
3     int c0 = 0;
4     int c1 = 0;
5     while (temp & 1 == 1) {
6       c1++;
7       temp >>= 1;
8     }
9
```

```
10    if (temp == 0)  return -1;
11
12    while (((temp & 1) == 0) && (temp != 0)) {
13      c0++;
14      temp >>= 1;
15    }
16
17    int p = c0 + c1; // position of rightmost non-trailing one
18    n &= ((~0) << (p + 1)); // clears from bit p onwards
19
20    int mask = (1 << (c1 + 1)) - 1; // Sequence of (c1+1) ones
21    n |= mask << (c0 - 1);
22
23    return n;
24  }
```

#### Arithmetic Approach to Get Next Number

If `c0` is the number of trailing zeros, `c1` is the size of the one block immediately following, and `p = c0 + c1`, we can word our solution from earlier as follows:

1. Set the pth bit to 1.
2. Set all bits following p to 0.
3. Set bits 0 through `c1 - 2` to 1. This will be `c1 - 1` total bits.

A quick and dirty way to perform steps 1 and 2 is to set the trailing zeros to 1 (giving us p trailing ones), and then add 1. Adding one will flip all trailing ones, so we wind up with a 1 at bit p followed by p zeros. We can perform this arithmetically.

```
n += 2^c0 - 1;    // Sets trailing 0s to 1, giving us p trailing 1s
n += 1;           // Flips first p 1s to 0s, and puts a 1 at bit p.
```

Now, to perform Step 3 arithmetically, we just do:

```
n += 2^(c1 - 1) - 1; // Sets trailing c1 - 1 zeros to ones.
```

This math reduces to:

```
next = n + (2^c0 - 1) + 1 + (2^(c1 - 1) - 1)
     = n + 2^c0 + 2^(c1 - 1) - 1
```

The best part is that, using a little bit manipulation, it's simple to code.

```
1  int getNextArith(int n) {
2    /* ... same calculation for c0 and c1 as before ... */
3    return n + (1 << c0) + (1 << (c1 - 1)) - 1;
4  }
```

#### Arithmetic Approach to Get Previous Number

If $c_1$ is the number of trailing ones, $c_0$ is the size of the zero block immediately following, and $p = c_0 + c_1$, we can word the initial `getPrev` solution as follows:

1. Set the pth bit to 0
2. Set all bits following p to 1
3. Set bits 0 through $c_0 - 1$ to 0.

We can implement this arithmetically as follows. For clarity in the example, we will assume `n = 10000011`. This makes $c_1 = 2$ and $c_0 = 5$.

```
n -= 2^c1 - 1;          // Removes trailing 1s. n is now 10000000.
n -= 1;                 // Flips trailing 0s. n is now 01111111.
n -= 2^(c0 - 1) - 1;    // Flips last (c0-1) 0s. n is now 01110000.
```

This reduces mathematically to:

```
next    = n - (2^c1 - 1) - 1 - (2^(c0 - 1) - 1).
        = n - 2^c1 - 2^(c0 - 1) + 1
```

Again, this is very easy to implement.

```
1  int getPrevArith(int n) {
2    /* ... same calculation for c0 and c1 as before ... */
3    return n - (1 << c1) - (1 << (c0 - 1)) + 1;
4  }
```

Whew! Don't worry, you wouldn't be expected to get all this in an interview—at least not without a lot of help from the interviewer.

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

*pg 116*

**SOLUTION**

We can work backwards to solve this question.

**What does it mean if A & B == 0?**

It means that A and B never have a 1 bit in the same place. So if `n & (n-1) == 0`, then n and `n-1` never share a 1.

**What does n-1 look like (as compared with n)?**

Try doing subtraction by hand (in base 2 or 10). What happens?

```
  1101011000 [base 2]          593100 [base 10]
-          1               -       1
= 1101010111 [base 2]        = 593099 [base 10]
```

When you subtract 1 from a number, you look at the least significant bit. If it's a 1 you change it to 0, and you are done. If it's a zero, you must "borrow" from a larger bit. So, you go to increasingly larger bits, changing each bit from a 0 to a 1, until you find a 1. You flip that 1 to a 0 and you are done.

Thus, `n-1` will look like n, except that n's initial 0s will be 1s in `n-1`, and n's least significant 1 will be a 0 in `n-1`. That is:

```
if     n = abcde1000
then n-1 = abcde0111
```

**So what does n & (n-1) == 0 indicate?**

n and `n-1` must have no 1s in common. Given that they look like this:

```
if     n = abcde1000
then n-1 = abcde0111
```

abcde must be all 0s, which means that n must look like this: `00001000`. The value n is therefore a power of two.

So, we have our answer: `((n & (n-1)) == 0)` checks if n is a power of 2 (or if n is 0).

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

EXAMPLE

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

Output: 2

*pg 116*

**SOLUTION**

This seemingly complex problem is actually rather straightforward. To approach this, ask yourself how you would figure out which bits in two numbers are different. Simple: with an XOR.

Each 1 in the XOR represents a bit that is different between A and B. Therefore, to check the number of bits that are different between A and B, we simply need to count the number of bits in A^B that are 1.

```
1  int bitSwapRequired(int a, int b) {
2    int count = 0;
3    for (int c = a ^ b; c != 0; c = c >> 1) {
4      count += c & 1;
5    }
6    return count;
7  }
```

This code is good, but we can make it a bit better. Rather than simply shifting c repeatedly while checking the least significant bit, we can continuously flip the least significant bit and count how long it takes c to reach 0. The operation `c = c & (c - 1)` will clear the least significant bit in c.

The code below utilizes this approach.

```
1  int bitSwapRequired(int a, int b) {
2    int count = 0;
3    for (int c = a ^ b; c != 0; c = c & (c-1)) {
4      count++;
5    }
6    return count;
7  }
```

The above code is one of those bit manipulation problems that comes up sometimes in interviews. Though it'd be hard to come up with it on the spot if you've never seen it before, it is useful to remember the trick for your interviews.

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

*pg 116*

**SOLUTION**

Like many of the previous problems, it's useful to think about this problem in a different way. Operating on individual pairs of bits would be difficult, and probably not that efficient either. So how else can we think about this problem?

We can approach this as operating on the odds bits first, and then the even bits. Can we take a number n and move the odd bits over by 1? Sure. We can mask all odd bits with `10101010` in binary (which is `0xAA`),

then shift them right by 1 to put them in the even spots. For the even bits, we do an equivalent operation. Finally, we merge these two values.

This takes a total of five instructions. The code below implements this approach.

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

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

We've implemented the code above for 32-bit integers in Java. If you were working with 64-bit integers, you would need to change the mask. The logic, however, would remain the same.

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

The method signature should look something like:

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

*pg 116*

**SOLUTION**

A naive solution to the problem is straightforward: iterate in a for loop from x1 to x2, setting each pixel along the way. But that's hardly any fun, is it? (Nor is it very efficient.)

A better solution is to recognize that if x1 and x2 are far away from each other, several full bytes will be contained between them. These full bytes can be set one at a time by doing `screen[byte_pos] = 0xFF`. The residual start and end of the line can be set using masks.

```
1  void drawLine(byte[] screen, int width, int x1, int x2, int y) {
2    int start_offset = x1 % 8;
3    int first_full_byte = x1 / 8;
4    if (start_offset != 0) {
5      first_full_byte++;
6    }
7
8    int end_offset = x2 % 8;
9    int last_full_byte = x2 / 8;
10   if (end_offset != 7) {
11     last_full_byte--;
12   }
13
14   // Set full bytes
15   for (int b = first_full_byte; b <= last_full_byte; b++) {
16     screen[(width / 8) * y + b] = (byte) 0xFF;
17   }
18
19   // Create masks for start and end of line
20   byte start_mask = (byte) (0xFF >> start_offset);
21   byte end_mask = (byte) ~(0xFF >> (end_offset + 1));
22
23   // Set start and end of line
24   if ((x1 / 8) == (x2 / 8)) { // x1 and x2 are in the same byte
```

```
25        byte mask = (byte) (start_mask & end_mask);
26        screen[(width / 8) * y + (x1 / 8)] |= mask;
27      } else {
28        if (start_offset != 0) {
29          int byte_number = (width / 8) * y + first_full_byte - 1;
30          screen[byte_number] |= start_mask;
31        }
32        if (end_offset != 7) {
33          int byte_number = (width / 8) * y + last_full_byte + 1;
34          screen[byte_number] |= end_mask;
35        }
36      }
37    }
```

Be careful on this problem; there are a lot of "gotchas" and special cases. For example, you need to consider the case where x1 and x2 are in the same byte. Only the most careful candidates can implement this code bug-free.

# 6

# Solutions to Math and Logic Puzzles

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

*pg 122*

**SOLUTION**

Sometimes, tricky constraints can be a clue. This is the case with the constraint that we can only use the scale once.

Because we can only use the scale once, we know something interesting: we must weigh multiple pills at the same time. In fact, we know we must weigh pills from at least 19 bottles at the same time. Otherwise, if we skipped two or more bottles entirely, how could we distinguish between those missed bottles? Remember that we only have *one* chance to use the scale.

So how can we weigh pills from more than one bottle and discover which bottle has the heavy pills? Let's suppose there were just two bottles, one of which had heavier pills. If we took one pill from each bottle, we would get a weight of 2.1 grams, but we wouldn't know which bottle contributed the extra 0.1 grams. We know we must treat the bottles differently somehow.

If we took one pill from Bottle #1 and two pills from Bottle #2, what would the scale show? It depends. If Bottle #1 were the heavy bottle, we would get 3.1 grams. If Bottle #2 were the heavy bottle, we would get 3.2 grams. And that is the trick to this problem.

We know the "expected" weight of a bunch of pills. The difference between the expected weight and the actual weight will indicate which bottle contributed the heavier pills, *provided* we select a different number of pills from each bottle.

We can generalize this to the full solution: take one pill from Bottle #1, two pills from Bottle #2, three pills from Bottle #3, and so on. Weigh this mix of pills. If all pills were one gram each, the scale would read 210 grams (`1 + 2 + ... + 20 = 20 * 21 / 2 = 210`). Any "overage" must come from the extra 0.1 gram pills.

This formula will tell you the bottle number:

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

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

**6.2** **Basketball:** You have a basketball hoop and someone says that you can play one of two games.

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

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

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

*pg 123*

**SOLUTION**

To solve this problem, we can apply straightforward probability laws by comparing the probabilities of winning each game.

**Probability of winning Game 1:**

The probability of winning Game 1 is p, by definition.

**Probability of winning Game 2:**

Let $s(k,n)$ be the probability of making exactly k shots out of n. The probability of winning Game 2 is the probability of making exactly two shots out of three OR making all three shots. In other words:

$$P(\text{winning}) = s(2,3) + s(3,3)$$

The probability of making all three shots is:

$$s(3,3) = p^3$$

The probability of making exactly two shots is:

$$\begin{aligned} &P(\text{making 1 and 2, and missing 3}) \\ &\quad + P(\text{making 1 and 3, and missing 2}) \\ &\quad + P(\text{missing 1, and making 2 and 3}) \\ &= p * p * (1 - p) + p * (1 - p) * p + (1 - p) * p * p \\ &= 3 (1 - p) p^2 \end{aligned}$$

Adding these together, we get:

$$\begin{aligned} &= p^3 + 3 (1 - p) p^2 \\ &= p^3 + 3p^2 - 3p^3 \\ &= 3p^2 - 2p^3 \end{aligned}$$

**Which game should you play?**

You should play Game 1 if $P(\text{Game 1}) > P(\text{Game 2})$:

$$\begin{aligned} &p > 3p^2 - 2p^3. \\ &1 > 3p - 2p^2 \\ &2p^2 - 3p + 1 > 0 \\ &(2p - 1)(p - 1) > 0 \end{aligned}$$

Both terms must be positive, or both must be negative. But we know $p < 1$, so $p - 1 < 0$. This means both terms must be negative.

$$\begin{aligned} &2p - 1 < 0 \\ &2p < 1 \\ &p < .5 \end{aligned}$$

So, we should play Game 1 if $0 < p < .5$ and Game 2 if $.5 < p < 1$.

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

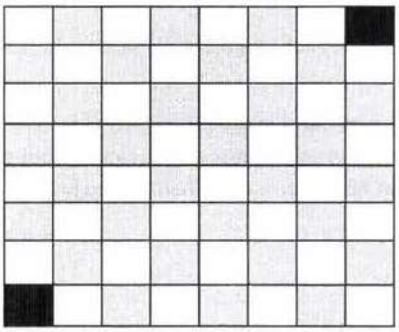
**6.3** **Dominos:** There is an 8x8 chessboard in which two diagonally opposite corners have been cut off. You are given 31 dominos, and a single domino can cover exactly two squares. Can you use the 31 dominos to cover the entire board? Prove your answer (by providing an example or showing why it's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

*pg 123*

**SOLUTION**

At first, it seems like this should be possible. It's an 8 x 8 board, which has 64 squares, but two have been cut off, so we're down to 62 squares. A set of 31 dominoes should be able to fit there, right?

When we try to lay down dominoes on row 1, which only has 7 squares, we may notice that one domino must stretch into the row 2. Then, when we try to lay down dominoes onto row 2, again we need to stretch a domino into row 3.



For each row we place, we'll always have one domino that needs to poke into the next row. No matter how many times and ways we try to solve this issue, we won't be able to successfully lay down all the dominoes.

There's a cleaner, more solid proof for why it won't work. The chessboard initially has 32 black and 32 white squares. By removing opposite corners (which must be the same color), we're left with 30 of one color and 32 of the other color. Let's say, for the sake of argument, that we have 30 black and 32 white squares.

Each domino we set on the board will always take up one white and one black square. Therefore, 31 dominos will take up 31 white squares and 31 black squares exactly. On this board, however, we must have 30 black squares and 32 white squares. Hence, it is impossible.

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

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

*pg 123*

**SOLUTION**

The ants will collide if any of them are moving towards each other. So, the only way that they won't collide is if they are all moving in the same direction (clockwise or counterclockwise). We can compute this probability and work backwards from there.

Since each ant can move in two directions, and there are three ants, the probability is:

$$P(\text{clockwise}) = (1/2)^3$$
$$P(\text{counter clockwise}) = (1/2)^3$$
$$P(\text{same direction}) = (1/2)^3 + (1/2)^3 = 1/4$$

The probability of collision is therefore the probability of the ants *not* moving in the same direction:

$$P(\text{collision}) = 1 - P(\text{same direction}) = 1 - 1/4 = 3/4$$

To generalize this to an n-vertex polygon: there are still only two ways in which the ants can move to avoid a collision, but there are $2^n$ ways they can move in total. Therefore, in general, probability of collision is:

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

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

*pg 123*

**SOLUTION**

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

| 5 Quart | 3 Quart | Action |
|---|---|---|
| 5 | 0 | Filled 5-quart jug. |
| 2 | 3 | Filled 3-quart with 5-quart's contents. |
| 2 | 0 | Dumped 3-quart. |
| 0 | 2 | Fill 3-quart with 5-quart's contents. |
| 5 | 2 | Filled 5-quart. |
| 4 | 3 | Fill remainder of 3-quart with 5-quart. |
| 4 | | Done! We have 4 quarts. |

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

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

*pg 123*

**SOLUTION**

Let's apply the Base Case and Build approach. Assume that there are n people on the island and c of them have blue eyes. We are explicitly told that `c > 0`.

**Case c = 1: Exactly one person has blue eyes.**

Assuming all the people are intelligent, the blue-eyed person should look around and realize that no one else has blue eyes. Since he knows that at least one person has blue eyes, he must conclude that it is he who has blue eyes. Therefore, he would take the flight that evening.

**Case c = 2: Exactly two people have blue eyes.**

The two blue-eyed people see each other, but are unsure whether c is 1 or 2. They know, from the previous case, that if c = 1, the blue-eyed person would leave on the first night. Therefore, if the other blue-eyed person is still there, he must deduce that c = 2, which means that he himself has blue eyes. Both men would then leave on the second night.

**Case c > 2: The General Case.**

As we increase c, we can see that this logic continues to apply. If c = 3, then those three people will immediately know that there are either 2 or 3 people with blue eyes. If there were two people, then those two people would have left on the second night. So, when the others are still around after that night, each person would conclude that c = 3 and that they, therefore, have blue eyes too. They would leave that night.

This same pattern extends up through any value of c. Therefore, if c men have blue eyes, it will take c nights for the blue-eyed men to leave. All will leave on the same night.

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

*pg 123*

**SOLUTION**

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

We can solve this problem multiple ways.

### Mathematically

We can work out the probability for each gender sequence.

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

And so on.

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

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

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

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

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

$$1/4 = 32/128 \qquad 4/32 = 16/128$$
$$2/8 = 32/128 \qquad 5/64 = 10/128$$
$$3/16 = 24/128 \qquad 6/128 = 6/128$$

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

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

### Logically

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

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

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

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

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

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

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

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

#### Simulation

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

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

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

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

*pg 124*

**SOLUTION**

We may observe that, regardless of how we drop Egg 1, Egg 2 must do a linear search (from lowest to highest) between the "breaking floor" and the next highest non-breaking floor. For example, if Egg 1 is dropped from floors 5 and 10 without breaking, but it breaks when it's dropped from floor 15, then Egg 2 must be dropped, in the worst case, from floors 11, 12, 13, and 14.

### The Approach

As a first try, suppose we drop an egg from the 10th floor, then the 20th, ...

- If Egg 1 breaks on the first drop (floor 10), then we have at most 10 drops total.
- If Egg 1 breaks on the last drop (floor 100), then we have at most 19 drops total (floors 10, 20, ...,90, 100, then 91 through 99).

That's pretty good, but all we've considered is the absolute worst case. We should do some "load balancing" to make those two cases more even.

Our goal is to create a system for dropping Egg 1 such that the number of drops is as consistent as possible, whether Egg 1 breaks on the first drop or the last drop.

1. A perfectly load-balanced system would be one in which `Drops(Egg 1) + Drops(Egg 2)` is always the same, regardless of where Egg 1 breaks.
2. For that to be the case, since each drop of Egg 1 takes one more step, Egg 2 is allowed one fewer step.
3. We must, therefore, reduce the number of steps potentially required by Egg 2 by one drop each time. For example, if Egg 1 is dropped on floor 20 and then floor 30, Egg 2 is potentially required to take 9 steps. When we drop Egg 1 again, we must reduce potential Egg 2 steps to only 8. That is, we must drop Egg 1 at floor 39.
4. Therefore, Egg 1 must start at floor X, then go up by X-1 floors, then X-2, ..., until it gets to 100.
5. Solve for X.

$$X + (X - 1) + (X - 2) + ... + 1 = 100$$

$$X(X+1)/2 = 100$$

$$X \approx 13.65$$

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

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

"round up" case).

Therefore, we should round X up to 14. That is, we go to floor 14, then 27, then 39, .... This takes 14 steps in the worse case.

As in many other maximizing / minimizing problems, the key in this problem is "worst case balancing."

The following code simulates this approach.

```
1  int breakingPoint = ...;
2  int countDrops = 0;
3
4  boolean drop(int floor) {
5    countDrops++;
6    return floor >= breakingPoint;
7  }
8
9  int findBreakingPoint(int floors) {
10   int interval = 14;
11   int previousFloor = 0;
12   int egg1 = interval;
13
14   /* Drop egg1 at decreasing intervals. */
15   while (!drop(egg1) && egg1 <= floors) {
16     interval -= 1;
17     previousFloor = egg1;
18     egg1 += interval;
19   }
20
21   /* Drop egg2 at 1 unit increments. */
22   int egg2 = previousFloor + 1;
23   while (egg2 < egg1 && egg2 <= floors && !drop(egg2)) {
24     egg2 += 1;
25   }
26
27   /* If it didn't break, return -1. */
28   return egg2 > floors ? -1 : egg2;
29 }
```

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

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

This will involve the quadratic formula.

**6.9** **100 Lockers:** There are 100 closed lockers in a hallway. A man begins by opening all 100 lockers. Next, he closes every second locker. Then, on his third pass, he toggles every third locker (closes it if it is open or opens it if it is closed). This process continues for 100 passes, such that on each pass `i`, the man toggles every `i`th locker. After his 100th pass in the hallway, in which he toggles only locker #100, how many lockers are open?

*pg 124*

**SOLUTION**

We can tackle this problem by thinking through what it means for a door to be toggled. This will help us deduce which doors at the very end will be left opened.

**Question: For which rounds is a door toggled (open or closed)?**

A door n is toggled once for each factor of n, including itself and 1. That is, door 15 is toggled on rounds 1, 3, 5, and 15.

**Question: When would a door be left open?**

A door is left open if the number of factors (which we will call x) is odd. You can think about this by pairing factors off as an open and a close. If there's one remaining, the door will be open.

**Question: When would x be odd?**

The value x is odd if n is a perfect square. Here's why: pair n's factors by their complements. For example, if n is 36, the factors are (1, 36), (2, 18), (3, 12), (4, 9), (6, 6). Note that (6, 6) only contributes one factor, thus giving n an odd number of factors.

**Question: How many perfect squares are there?**

There are 10 perfect squares. You could count them (1, 4, 9, 16, 25, 36, 49, 64, 81, 100), or you could simply realize that you can take the numbers 1 through 10 and square them:

```
1*1, 2*2, 3*3, ..., 10*10
```

Therefore, there are 10 lockers open at the end of this process.

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

Follow up: Write code to simulate your approach.

*pg 124*

**SOLUTION**

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

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

**Naive Approach (28 days)**

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

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

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

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

```
1  class Bottle {
2    private boolean poisoned = false;
3    private int id;
4
5    public Bottle(int id) { this.id = id; }
6    public int getId() { return id; }
7    public void setAsPoisoned() { poisoned = true; }
8    public boolean isPoisoned() { return poisoned; }
9  }
10
11 class TestStrip {
12   public static int DAYS_FOR_RESULT = 7;
13   private ArrayList<ArrayList<Bottle>> dropsByDay =
14     new ArrayList<ArrayList<Bottle>>();
15   private int id;
16
17   public TestStrip(int id) { this.id = id; }
18   public int getId() { return id; }
19
20   /* Resize list of days/drops to be large enough. */
21   private void sizeDropsForDay(int day) {
22     while (dropsByDay.size() <= day) {
23       dropsByDay.add(new ArrayList<Bottle>());
24     }
25   }
26
27   /* Add drop from bottle on specific day. */
28   public void addDropOnDay(int day, Bottle bottle) {
29     sizeDropsForDay(day);
30     ArrayList<Bottle> drops = dropsByDay.get(day);
31     drops.add(bottle);
32   }
33
34   /* Checks if any of the bottles in the set are poisoned. */
35   private boolean hasPoison(ArrayList<Bottle> bottles) {
36     for (Bottle b : bottles) {
37       if (b.isPoisoned()) {
38         return true;
39       }
40     }
41     return false;
42   }
43
44   /* Gets bottles used in the test DAYS_FOR_RESULT days ago. */
45   public ArrayList<Bottle> getLastWeeksBottles(int day) {
46     if (day < DAYS_FOR_RESULT) {
47       return null;
48     }
49     return dropsByDay.get(day - DAYS_FOR_RESULT);
50   }
51
52   /* Checks for poisoned bottles since before DAYS_FOR_RESULT */
53   public boolean isPositiveOnDay(int day) {
```

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

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

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

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

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

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

**Optimized Approach (10 days)**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Implementing this requires some careful work to prevent bugs.

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

```
29      if (digits[3] == -1) { /* Day 3 didn't give new result */
30        /* Digit 2 equals digit 0 or digit 1. But, digit 2, when incremented also
31         * matches digit 0 or digit 1. This means that digit 0 incremented matches
32         * digit 1, or the other way around. */
33        digits[2] = ((digits[0] + 1) % nTestStrips) == digits[1] ?
34                    digits[0] : digits[1];
35      } else {
36          digits[2] = (digits[3] - 1 + nTestStrips) % nTestStrips;
37      }
38    }
39
40    return digits[0] * 100 + digits[1] * 10 + digits[2];
41  }
42
43  /* Run set of tests for this day. */
44  void runTestSet(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips, int day) {
45    if (day > 3) return; // only works for 3 days (digits) + one extra
46
47    for (Bottle bottle : bottles) {
48      int index = getTestStripIndexForDay(bottle, day, strips.size());
49      TestStrip testStrip = strips.get(index);
50      testStrip.addDropOnDay(day, bottle);
51    }
52  }
53
54  /* Get strip that should be used on this bottle on this day. */
55  int getTestStripIndexForDay(Bottle bottle, int day, int nTestStrips) {
56    int id = bottle.getId();
57    switch (day) {
58      case 0: return id /100;
59      case 1: return (id % 100) / 10;
60      case 2: return id % 10;
61      case 3: return (id % 10 + 1) % nTestStrips;
62      default: return -1;
63    }
64  }
65
66  /* Get results that are positive for a particular day, excluding prior results. */
67  int getPositiveOnDay(ArrayList<TestStrip> testStrips, int day,
68                       HashSet<Integer> previousResults) {
69    for (TestStrip testStrip : testStrips) {
70      int id = testStrip.getId();
71      if (testStrip.isPositiveOnDay(day) && !previousResults.contains(id)) {
72        return testStrip.getId();
73      }
74    }
75    return -1;
76  }
```

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

### Optimal Approach (7 days)

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

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

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

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

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

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

# 7

# Solutions to Object-Oriented Design

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

*pg 127*

**SOLUTION**

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

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

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

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

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

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

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

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

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

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

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

*pg 127*

**SOLUTION**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

*pg 127*

**SOLUTION**

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

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

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

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

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

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

A user also can be introduced:

- Adding
- Deleting
- Credit information

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

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

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

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

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

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

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

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

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

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

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

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

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

*pg 127*

**SOLUTION**

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

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

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

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

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

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

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

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

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

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

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

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

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

*pg 127*

**SOLUTION**

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

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

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

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

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

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

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

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

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

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

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

```
14 }
15
16 public class User {
17   private int userId;
18   private String details;
19   private int accountType;
20
21   public void renewMembership() {  }
22
23   public User(int id, String details, int accountType) {
24     userId = id;
25     this.details = details;
26     this.accountType = accountType;
27   }
28
29   /* Getters and setters */
30   public int getID() { return userId; }
31   public void setID(int id) { userId = id; }
32   public String getDetails() {
33     return details;
34   }
35
36   public void setDetails(String details) {
37     this.details = details;
38   }
39   public int getAccountType() { return accountType; }
40   public void setAccountType(int t) { accountType = t; }
41 }
```

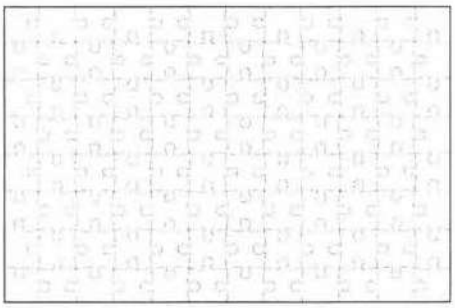
The decision to tear off user management, library, and display into their own classes, when this functionality could have been in the general `OnlineReaderSystem` class, is an interesting one. On a very small system, making this decision could make the system overly complex. However, as the system grows,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

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

*pg 128*

**SOLUTION**

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



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

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

For our solution, we will use the absolute position.

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

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

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

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

A potential object-oriented design looks like the following:

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

```
23        }
24     }
25  }
26
27  public class Puzzle {
28     private LinkedList<Piece> pieces; /* Remaining pieces to put away. */
29     private Piece[][] solution;
30     private int size;
31
32     public Puzzle(int size, LinkedList<Piece> pieces) { ... }
33
34
35     /* Put piece into the solution, turn it appropriately, and remove from list. */
36     private void setEdgeInSolution(LinkedList<Piece> pieces, Edge edge, int row,
37                                    int column, Orientation orientation) {
38        Piece piece = edge.getParentPiece();
39        piece.setEdgeAsOrientation(edge, orientation);
40        pieces.remove(piece);
41        solution[row][column] = piece;
42     }
43
44     /* Find the matching piece in piecesToSearch and insert it at row, column. */
45     private boolean fitNextEdge(LinkedList<Piece> piecesToSearch, int row, int col);
46
47     /* Solve puzzle. */
48     public boolean solve() { ... }
49  }
50
51  public class Piece {
52     private HashMap<Orientation, Edge> edges = new HashMap<Orientation, Edge>();
53
54     public Piece(Edge[] edgeList) { ... }
55
56     /* Rotate edges by "numberRotations". */
57     public void rotateEdgesBy(int numberRotations) { ... }
58
59     public boolean isCorner() { ... }
60     public boolean isBorder() { ... }
61  }
62
63  public class Edge {
64     private Shape shape;
65     private Piece parentPiece;
66     public Edge(Shape shape) { ... }
67     public boolean fitsWith(Edge edge) { ... }
68  }
```

### Algorithm to Solve the Puzzle

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

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

The code below outlines this algorithm.

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

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

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

*pg 128*

**SOLUTION**

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

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

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

**What specific actions does it need to support?**

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

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

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

**What can we learn about these requirements?**

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

**What are the core components of the system?**

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

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

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

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

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

**What are the key objects and methods?**

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

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

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

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

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

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

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

```
    private ArrayList<GroupChat> groupChats;

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

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

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

    private String accountName;
    private String fullName;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

*Q2: How do we deal with conflicting information?*

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

*Q3: How do we make our server scale?*

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

*Q4: How we do prevent denial of service attacks?*

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

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

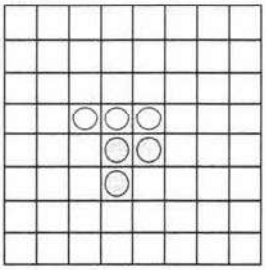
*pg 128*

**SOLUTION**

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

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

This sequence of moves leads to the board below.



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

**Should BlackPiece and WhitePiece be classes?**

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

**Do we need separate Board and Game classes?**

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

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

**Who keeps score?**

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

**Should Game be a Singleton class?**

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

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

One possible design for Othello is below.

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

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

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

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

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

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

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

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

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

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

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

*pg 128*

## SOLUTION

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

### Implementing the CircularArray class

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

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

The code below implements this approach.

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

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

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

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

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

#### Implementing the Iterator Interface

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

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

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

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

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

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

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

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

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

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

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

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

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

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | | | |
| | 1 | * | 1 | | | |
| | 2 | 2 | 2 | | | |
| | 1 | * | 1 | | | |
| | 1 | 1 | 1 | | | |
| | | | 1 | 1 | 1 | |
| | | | 1 | * | 1 | |

The player initially sees a board with nothing exposed.

| | | | | | | |
|---|---|---|---|---|---|---|
| ? | ? | ? | ? | ? | ? | ? |
| ? | ? | ? | ? | ? | ? | ? |
| ? | ? | ? | ? | ? | ? | ? |
| ? | ? | ? | ? | ? | ? | ? |
| ? | ? | ? | ? | ? | ? | ? |
| ? | ? | ? | ? | ? | ? | ? |
| ? | ? | ? | ? | ? | ? | ? |

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

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | ? | ? | ? | ? | ? |
| | 1 | ? | ? | ? | ? | ? |
| | 2 | ? | ? | ? | ? | ? |
| | 1 | ? | ? | ? | ? | ? |
| | 1 | 1 | 1 | ? | ? | ? |
| | | | 1 | ? | ? | ? |
| | | | 1 | ? | ? | ? |

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

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | | | |
| | 1 | ? | 1 | | | |
| | 2 | 2 | 2 | | | |
| | 1 | ? | 1 | | | |
| | 1 | 1 | 1 | | | |
| | | | 1 | 1 | 1 | |
| | | | 1 | ? | 1 | |

*pg 129*

**SOLUTION**

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

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

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

### Design: Cell

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

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

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

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

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

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

### Design: Board

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

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

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

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

- Expanding blank areas.

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

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

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

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

**Design: Game**

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

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

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

### Algorithms

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

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

*Placing the Bombs*

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

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

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

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

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

*Setting the Numbered Cells*

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

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

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

*Expanding a Blank Region*

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

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

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

```
22              }
23            }
24          }
25        }
26    }
```

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

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

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

*pg 129*

**SOLUTION**

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

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

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

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

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

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

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

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

*pg 129*

**SOLUTION**

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

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

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

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

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

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

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

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

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

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

The code below uses this implementation.

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

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

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

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

# 8

# Solutions to Recursion and Dynamic Programming

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

*pg 134*

**SOLUTION**

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

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

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

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

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

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

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

**Brute Force Solution**

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

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

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

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

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

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

A simple implementation of this code is below.

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

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

### Memoization Solution

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

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

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

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

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

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

**8.2** **Robot in a Grid:** Imagine a robot sitting on the upper left corner of grid with r rows and c columns. The robot can only move in two directions, right and down, but certain cells are "off limits" such that the robot cannot step on them. Design an algorithm to find a path for the robot from the top left to the bottom right.

*pg 135*

**SOLUTION**

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

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

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

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

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

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

We should look for a faster way.

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

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

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

This is what the dynamic programming algorithm below does.

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

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

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

FOLLOW UP

What if the values are not distinct?

*pg 135*

**SOLUTION**

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

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

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

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

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

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

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

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

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

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

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

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

**Follow Up: What if the elements are not distinct?**

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

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

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

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

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

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

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

The code below implements this algorithm.

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

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

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

*pg 135*

**SOLUTION**

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

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

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

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

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

**Solution #1: Recursion**

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

*Base Case:* $n = 0$.

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

*Case:* $n = 1$.

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

*Case:* $n = 2$.

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

*Case:* $n = 3$.

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

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

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

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

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

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

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

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

*Case:* `n > 0`

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

The following code implements this algorithm:

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

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

**Solution #2: Combinatorics**

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

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

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

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

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

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

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

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

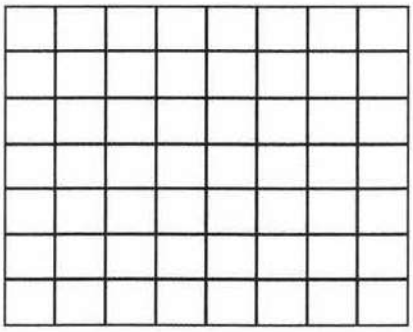
*pg 135*

**SOLUTION**

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

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

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



**Solution #1**

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

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

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

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

```
3     int smaller = a < b ? a : b;
4     return minProductHelper(smaller, bigger);
5   }
6
7   int minProductHelper(int smaller, int bigger) {
8     if (smaller == 0) { // 0 x bigger = 0
9       return 0;
10    } else if (smaller == 1) { // 1 x bigger = bigger
11      return bigger;
12    }
13
14    /* Compute half. If uneven, compute other half. If even, double it. */
15    int s = smaller >> 1; // Divide by 2
16    int side1 = minProduct(s, bigger);
17    int side2 = side1;
18    if (smaller % 2 == 1) {
19      side2 = minProductHelper(smaller - s, bigger);
20    }
21
22    return side1 + side2;
23  }
```

Can we do better? Yes.

### Solution #2

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

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

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

```
1   int minProduct(int a, int b) {
2     int bigger = a < b ? b : a;
3     int smaller = a < b ? a : b;
4
5     int memo[] = new int[smaller + 1];
6     return minProduct(smaller, bigger, memo);
7   }
8
9   int minProduct(int smaller, int bigger, int[] memo) {
10    if (smaller == 0) {
11      return 0;
12    } else if (smaller == 1) {
13      return bigger;
14    } else if (memo[smaller] > 0) {
15      return memo[smaller];
16    }
17
18    /* Compute half. If uneven, compute other half. If even, double it. */
```

```
19    int s = smaller >> 1; // Divide by 2
20    int side1 = minProduct(s, bigger, memo); // Compute half
21    int side2 = side1;
22    if (smaller % 2 == 1) {
23       side2 = minProduct(smaller - s, bigger, memo);
24    }
25
26    /* Sum and cache.*/
27    memo[smaller] = side1 + side2;
28    return memo[smaller];
29 }
```

We can still make this a bit faster.

#### Solution #3

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

This is unnecessary. Instead, we can do:

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

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

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

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

```
1  int minProduct(int a, int b) {
2     int bigger = a < b ? b : a;
3     int smaller = a < b ? a : b;
4     return minProductHelper(smaller, bigger);
5  }
6
7  int minProductHelper(int smaller, int bigger) {
8     if (smaller == 0) return 0;
9     else if (smaller == 1) return bigger;
10
11    int s = smaller >> 1; // Divide by 2
12    int halfProd = minProductHelper(s, bigger);
13
14    if (smaller % 2 == 0) {
15       return halfProd + halfProd;
16    } else {
17       return halfProd + halfProd + bigger;
18    }
19 }
```

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

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

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

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

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

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

*pg 135*

**SOLUTION**

This problem sounds like a good candidate for the Base Case and Build approach.

Let's start with the smallest possible example: n = 1.

Case n = 1. Can we move Disk 1 from Tower 1 to Tower 3? Yes.

1. We simply move Disk 1 from Tower 1 to Tower 3.

Case n = 2. Can we move Disk 1 and Disk 2 from Tower 1 to Tower 3? Yes.

1. Move Disk 1 from Tower 1 to Tower 2
2. Move Disk 2 from Tower 1 to Tower 3
3. Move Disk 1 from Tower 2 to Tower 3

Note how in the above steps, Tower 2 acts as a buffer, holding a disk while we move other disks to Tower 3.

Case n = 3. Can we move Disk 1, 2, and 3 from Tower 1 to Tower 3? Yes.

1. We know we can move the top two disks from one tower to another (as shown earlier), so let's assume we've already done that. But instead, let's move them to Tower 2.
2. Move Disk 3 to Tower 3.
3. Move Disk 1 and Disk 2 to Tower 3. We already know how to do this—just repeat what we did in Step 1.

Case n = 4. Can we move Disk 1, 2, 3 and 4 from Tower 1 to Tower 3? Yes.

1. Move Disks 1, 2, and 3 to Tower 2. We know how to do that from the earlier examples.
2. Move Disk 4 to Tower 3.
3. Move Disks 1, 2 and 3 back to Tower 3.

Remember that the labels of Tower 2 and Tower 3 aren't important. They're equivalent towers. So, moving disks to Tower 3 with Tower 2 serving as a buffer is equivalent to moving disks to Tower 2 with Tower 3 serving as a buffer.

This approach leads to a natural recursive algorithm. In each part, we are doing the following steps, outlined below with pseudocode:

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

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

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

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

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

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

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

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

  public int index() {
    return index;
  }

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

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

```

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

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

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

*pg 135*

**SOLUTION**

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

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

*Base Case: permutations of first character substring*

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

$P(a_1) = a_1$

*Case: permutations of* $a_1a_2$

$P(a_1a_2) = a_1a_2$ and $a_2a_1$

*Case: permutations of* $a_1a_2a_3$

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

*Case: permutations of* $a_1a_2a_3a_4$

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

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

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

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

We can now implement this algorithm recursively.

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

```
7        return permutations;
8     }
9
10    char first = str.charAt(0); // get the first char
11    String remainder = str.substring(1); // remove the first char
12    ArrayList<String> words = getPerms(remainder);
13    for (String word : words) {
14       for (int j = 0; j <= word.length(); j++) {
15          String s = insertCharAt(word, first, j);
16          permutations.add(s);
17       }
18    }
19    return permutations;
20 }
21
22 /* Insert char c at index i in word. */
23 String insertCharAt(String word, char c, int i) {
24    String start = word.substring(0, i);
25    String end = word.substring(i);
26    return start + c + end;
27 }
```

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

*Base Case: single-character strings*

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

$P(a_1) = a_1$

*Case: two-character strings*

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

*Case: three-character strings*

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

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

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

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

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

This is now a fairly straightforward algorithm to implement.

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

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

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

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

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

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

*pg 135*

**SOLUTION**

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

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

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

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

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

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

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

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

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

The code below implements this algorithm.

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

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

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

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

EXAMPLE

Input: 3

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

*pg 136*

**SOLUTION**

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

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

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

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

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

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

So, we have the following:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

*pg 136*

**SOLUTION**

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

We can implement this algorithm recursively:

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

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

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

We could alternatively implement this using breadth-first search.

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

*pg 136*

**SOLUTION**

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

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

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

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

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

We can apply the same logic to `makeChange(100 using 2 quarters)`, `makeChange(100 using 3 quarters)` and `makeChange(100 using 4 quarters)`. We have thus reduced the above statement to the following.

```
makeChange(100) = makeChange(100 using 0 quarters) +
                  makeChange(75 using 0 quarters) +
                  makeChange(50 using 0 quarters) +
                  makeChange(25 using 0 quarters) +
                  1
```

Note that the final statement from above, `makeChange(100 using 4 quarters)`, equals 1. We call this "fully reduced."

Now what? We've used up all our quarters, so now we can start applying our next biggest denomination: dimes.

Our approach for quarters applies to dimes as well, but we apply this for *each* of the four of five parts of the above statement. So, for the first part, we get the following statements:

```
makeChange(100 using 0 quarters) = makeChange(100 using 0 quarters, 0 dimes) +
                                   makeChange(100 using 0 quarters, 1 dime)  +
                                   makeChange(100 using 0 quarters, 2 dimes) +
                                   ...
                                   makeChange(100 using 0 quarters, 10 dimes)

makeChange(75 using 0 quarters) = makeChange(75 using 0 quarters, 0 dimes) +
                                  makeChange(75 using 0 quarters, 1 dime)  +
                                  makeChange(75 using 0 quarters, 2 dimes) +
                                  ...
                                  makeChange(75 using 0 quarters, 7 dimes)

makeChange(50 using 0 quarters) = makeChange(50 using 0 quarters, 0 dimes) +
                                  makeChange(50 using 0 quarters, 1 dime)  +
                                  makeChange(50 using 0 quarters, 2 dimes) +
```

```
                                    ...
                                    makeChange(50 using 0 quarters, 5 dimes)

makeChange(25 using 0 quarters) = makeChange(25 using 0 quarters, 0 dimes) +
                                  makeChange(25 using 0 quarters, 1 dime)  +
                                  makeChange(25 using 0 quarters, 2 dimes)
```

Each one of these, in turn, expands out once we start applying nickels. We end up with a tree-like recursive structure where each call expands out to four or more calls.

The base case of our recursion is the fully reduced statement. For example, `makeChange(50 using 0 quarters, 5 dimes)` is fully reduced to 1, since 5 dimes equals 50 cents.

This leads to a recursive algorithm that looks like this:

```
1  int makeChange(int amount, int[] denoms, int index) {
2    if (index >= denoms.length - 1) return 1; // last denom
3    int denomAmount = denoms[index];
4    int ways = 0;
5    for (int i = 0; i * denomAmount <= amount; i++) {
6      int amountRemaining = amount - i * denomAmount;
7      ways += makeChange(amountRemaining, denoms, index + 1);
8    }
9    return ways;
10 }
11
12 int makeChange(int n) {
13   int[] denoms = {25, 10, 5, 1};
14   return makeChange(n, denoms, 0);
15 }
```

This works, but it's not as optimal as it could be. The issue is that we will be recursively calling `makeChange` several times for the same values of `amount` and `index`.

We can resolve this issue by storing the previously computed values. We'll need to store a mapping from each pair `(amount, index)` to the precomputed result.

```
1  int makeChange(int n) {
2    int[] denoms = {25, 10, 5, 1};
3    int[][] map = new int[n + 1][denoms.length]; // precomputed vals
4    return makeChange(n, denoms, 0, map);
5  }
6
7  int makeChange(int amount, int[] denoms, int index, int[][] map) {
8    if (map[amount][index] > 0) { // retrieve value
9      return map[amount][index];
10   }
11   if (index >= denoms.length - 1) return 1; // one denom remaining
12   int denomAmount = denoms[index];
13   int ways = 0;
14   for (int i = 0; i * denomAmount <= amount; i++) {
15     // go to next denom, assuming i coins of denomAmount
16     int amountRemaining = amount - i * denomAmount;
17     ways += makeChange(amountRemaining, denoms, index + 1, map);
18   }
19   map[amount][index] = ways;
20   return ways;
21 }
```

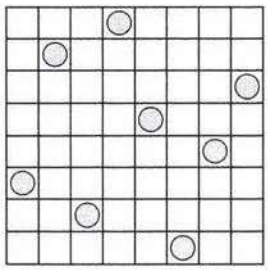
Note that we've used a two-dimensional array of integers to store the previously computed values. This is simpler, but takes up a little extra space. Alternatively, we could use an actual hash table that maps from `amount` to a new hash table, which then maps from `denom` to the precomputed value. There are other alternative data structures as well.

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

*pg 136*

**SOLUTION**

We have eight queens which must be lined up on an 8x8 chess board such that none share the same row, column or diagonal. So, we know that each row and column (and diagonal) must be used exactly once.



**A "Solved" Board with 8 Queens**

Picture the queen that is placed last, which we'll assume is on row 8. (This is an okay assumption to make since the ordering of placing the queens is irrelevant.) On which cell in row 8 is this queen? There are eight possibilities, one for each column.

So if we want to know all the valid ways of arranging 8 queens on an 8x8 chess board, it would be:

```
ways to arrange 8 queens on an 8x8 board =
   ways to arrange 8 queens on an 8x8 board with queen at (7, 0) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 1) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 2) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 3) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 4) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 5) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 6) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 7)
```

We can compute each one of these using a very similar approach:

```
ways to arrange 8 queens on an 8x8 board with queen at (7, 3) =
   ways to ... with queens at (7, 3) and (6, 0) +
   ways to ... with queens at (7, 3) and (6, 1) +
   ways to ... with queens at (7, 3) and (6, 2) +
   ways to ... with queens at (7, 3) and (6, 4) +
   ways to ... with queens at (7, 3) and (6, 5) +
   ways to ... with queens at (7, 3) and (6, 6) +
   ways to ... with queens at (7, 3) and (6, 7)
```

Note that we don't need to consider combinations with queens at `(7, 3)` and `(6, 3)`, since this is a violation of the requirement that every queen is in its own row, column and diagonal.

Implementing this is now reasonably straightforward.

```
1  int GRID_SIZE = 8;
2
3  void placeQueens(int row, Integer[] columns, ArrayList<Integer[]> results) {
4    if (row == GRID_SIZE) { // Found valid placement
5      results.add(columns.clone());
6    } else {
7      for (int col = 0; col < GRID_SIZE; col++) {
8        if (checkValid(columns, row, col)) {
9          columns[row] = col;  // Place queen
10         placeQueens(row + 1, columns, results);
11       }
12     }
13   }
14 }
15
16 /* Check if (row1, column1) is a valid spot for a queen by checking if there is a
17  * queen in the same column or diagonal. We don't need to check it for queens in
18  * the same row because the calling placeQueen only attempts to place one queen at
19  * a time. We know this row is empty. */
20 boolean checkValid(Integer[] columns, int row1, int column1) {
21   for (int row2 = 0; row2 < row1; row2++) {
22     int column2 = columns[row2];
23     /* Check if (row2, column2) invalidates (row1, column1) as a
24      * queen spot. */
25
26     /* Check if rows have a queen in the same column */
27     if (column1 == column2) {
28       return false;
29     }
30
31     /* Check diagonals: if the distance between the columns equals the distance
32      * between the rows, then they're in the same diagonal. */
33     int columnDistance = Math.abs(column2 - column1);
34
35     /* row1 > row2, so no need for abs */
36     int rowDistance = row1 - row2;
37     if (columnDistance == rowDistance) {
38       return false;
39     }
40   }
41   return true;
42 }
```

Observe that since each row can only have one queen, we don't need to store our board as a full 8x8 matrix. We only need a single array where `column[r] = c` indicates that row `r` has a queen at column `c`.

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

*pg 136*

**SOLUTION**

To tackle this problem, we need to recognize the relationship between the different subproblems.

**Solution #1**

Imagine we had the following boxes: $b_1, b_2, \ldots, b_n$. The biggest stack that we can build with all the boxes equals the max of (biggest stack with bottom $b_1$, biggest stack with bottom $b_2$, ..., biggest stack with bottom $b_n$). That is, if we experimented with each box as a bottom and built the biggest stack possible with each, we would find the biggest stack possible.

But, how would we find the biggest stack with a particular bottom? Essentially the same way. We experiment with different boxes for the second level, and so on for each level.

Of course, we only experiment with valid boxes. If $b_5$ is bigger than $b_1$, then there's no point in trying to build a stack that looks like $\{b_1, b_5, \ldots\}$. We already know $b_1$ can't be below $b_5$.

We can perform a small optimization here. The requirements of this problem stipulate that the lower boxes must be strictly greater than the higher boxes in all dimensions. Therefore, if we sort (descending order) the boxes on a dimension—any dimension—then we know we don't have to look backwards in the list. The box $b_1$ cannot be on top of box $b_5$, since its height (or whatever dimension we sorted on) is greater than $b_5$'s height.

The code below implements this algorithm recursively.

```
1  int createStack(ArrayList<Box> boxes) {
2    /* Sort in decending order by height. */
3    Collections.sort(boxes, new BoxComparator());
4    int maxHeight = 0;
5    for (int i = 0; i < boxes.size(); i++) {
6      int height = createStack(boxes, i);
7      maxHeight = Math.max(maxHeight, height);
8    }
9    return maxHeight;
10 }
11
12 int createStack(ArrayList<Box> boxes, int bottomIndex) {
13   Box bottom = boxes.get(bottomIndex);
14   int maxHeight = 0;
15   for (int i = bottomIndex + 1; i < boxes.size(); i++) {
16     if (boxes.get(i).canBeAbove(bottom)) {
17       int height = createStack(boxes, i);
18       maxHeight = Math.max(height, maxHeight);
19     }
20   }
21   maxHeight += bottom.height;
22   return maxHeight;
23 }
24
25 class BoxComparator implements Comparator<Box> {
```

```
26    @Override
27    public int compare(Box x, Box y){
28       return y.height - x.height;
29    }
30 }
```

The problem in this code is that it gets very inefficient. We try to find the best solution that looks like $\{b_3, b_4, \ldots\}$ even though we may have already found the best solution with $b_4$ at the bottom. Instead of generating these solutions from scratch, we can cache these results using memoization.

```
1  int createStack(ArrayList<Box> boxes) {
2    Collections.sort(boxes, new BoxComparator());
3    int maxHeight = 0;
4    int[] stackMap = new int[boxes.size()];
5    for (int i = 0; i < boxes.size(); i++) {
6       int height = createStack(boxes, i, stackMap);
7       maxHeight = Math.max(maxHeight, height);
8    }
9    return maxHeight;
10 }
11
12 int createStack(ArrayList<Box> boxes, int bottomIndex, int[] stackMap) {
13   if (bottomIndex < boxes.size() && stackMap[bottomIndex] > 0) {
14      return stackMap[bottomIndex];
15   }
16
17   Box bottom = boxes.get(bottomIndex);
18   int maxHeight = 0;
19   for (int i = bottomIndex + 1; i < boxes.size(); i++) {
20      if (boxes.get(i).canBeAbove(bottom)) {
21         int height = createStack(boxes, i, stackMap);
22         maxHeight = Math.max(height, maxHeight);
23      }
24   }
25   maxHeight += bottom.height;
26   stackMap[bottomIndex] = maxHeight;
27   return maxHeight;
28 }
```

Because we're only mapping from an index to a height, we can just use an integer array for our "hash table."

Be very careful here with what each spot in the hash table represents. In this code, `stackMap[i]` represents the tallest stack with box `i` at the bottom. Before pulling the value from the hash table, you have to ensure that box `i` can be placed on top of the current bottom.

It helps to keep the line that recalls from the hash table symmetric with the one that inserts. For example, in this code, we recall from the hash table with `bottomIndex` at the start of the method. We insert into the hash table with `bottomIndex` at the end.

#### Solution #2

Alternatively, we can think about the recursive algorithm as making a choice, at each step, whether to put a particular box in the stack. (We will again sort our boxes in descending order by a dimension, such as height.)

First, we choose whether or not to put box 0 in the stack. Take one recursive path with box 0 at the bottom and one recursive path without box 0. Return the better of the two options.

Then, we choose whether or not to put box 1 in the stack. Take one recursive path with box 1 at the bottom and one path without box 1. Return the better of the two options.

We will again use memoization to cache the height of the tallest stack with a particular bottom.

```
1  int createStack(ArrayList<Box> boxes) {
2    Collections.sort(boxes, new BoxComparator());
3    int[] stackMap = new int[boxes.size()];
4    return createStack(boxes, null, 0, stackMap);
5  }
6
7  int createStack(ArrayList<Box> boxes, Box bottom, int offset, int[] stackMap) {
8    if (offset >= boxes.size()) return 0; // Base case
9
10   /* height with this bottom */
11   Box newBottom = boxes.get(offset);
12   int heightWithBottom = 0;
13   if (bottom == null || newBottom.canBeAbove(bottom)) {
14     if (stackMap[offset] == 0) {
15       stackMap[offset] = createStack(boxes, newBottom, offset + 1, stackMap);
16       stackMap[offset] += newBottom.height;
17     }
18     heightWithBottom = stackMap[offset];
19   }
20
21   /* without this bottom */
22   int heightWithoutBottom = createStack(boxes, bottom, offset + 1, stackMap);
23
24   /* Return better of two options. */
25   return Math.max(heightWithBottom, heightWithoutBottom);
26 }
```

Again, pay close attention to when you recall and insert values into the hash table. It's typically best if these are symmetric, as they are in lines 15 and 16-18.

**8.14 Boolean Evaluation:** Given a boolean expression consisting of the symbols `0` (false), `1` (true), `&` (AND), `|` (OR), and `^` (XOR), and a desired boolean result value `result`, implement a function to count the number of ways of parenthesizing the expression such that it evaluates to `result`. The expression should be fully parenthesized (e.g., `(0)^(1)`) but not extraneously (e.g., `(((0))^(1))`).

EXAMPLE

```
countEval("1^0|0|1", false) -> 2
countEval("0&0&0&1^1|0", true) -> 10
```

*pg 136*

**SOLUTION**

As in other recursive problems, the key to this problem is to figure out the relationship between a problem and its subproblems.

**Brute Force**

Consider an expression like `0^0&0^1|1` and the target result `true`. How can we break down `countEval(0^0&0^1|1, true)` into smaller problems?

We could just essentially iterate through each possible place to put a parenthesis.

```
countEval(0^0&0^1|1, true) =
   countEval(0^0&0^1|1 where paren around char 1, true)
 + countEval(0^0&0^1|1 where paren around char 3, true)
 + countEval(0^0&0^1|1 where paren around char 5, true)
 + countEval(0^0&0^1|1 where paren around char 7, true)
```

Now what? Let's look at just one of those expressions—the paren around char 3. This gives us `(0^0)&(0^1)`.

In order to make that expression true, both the left and right sides must be true. So:

```
left = "0^0"
right = "0^1|1"
countEval(left & right, true) = countEval(left, true) * countEval(right, true)
```

The reason we multiply the results of the left and right sides is that each result from the two sides can be paired up with each other to form a unique combination.

Each of those terms can now be decomposed into smaller problems in a similar process.

What happens when we have an "|" (OR)? Or an "^" (XOR)?

If it's an OR, then either the left or the right side must be true—or both.

```
countEval(left | right, true) = countEval(left, true)  * countEval(right, false)
                              + countEval(left, false) * countEval(right, true)
                              + countEval(left, true)  * countEval(right, true)
```

If it's an XOR, then the left or the right side can be true, but not both.

```
countEval(left ^ right, true) = countEval(left, true) * countEval(right, false)
                              + countEval(left, false) * countEval(right, true)
```

What if we were trying to make the result `false` instead? We can switch up the logic from above:

```
countEval(left & right, false) = countEval(left, true)  * countEval(right, false)
                               + countEval(left, false) * countEval(right, true)
                               + countEval(left, false) * countEval(right, false)
countEval(left | right, false) = countEval(left, false) * countEval(right, false)
countEval(left ^ right, false) = countEval(left, false) * countEval(right, false)
                               + countEval(left, true)  * countEval(right, true)
```

Alternatively, we can just use the same logic from above and subtract it out from the total number of ways of evaluating the expression.

```
totalEval(left) = countEval(left, true) + countEval(left, false)
totalEval(right) = countEval(right, true) + countEval(right, false)
totalEval(expression) = totalEval(left) * totalEval(right)
countEval(expression, false) = totalEval(expression) - countEval(expression, true)
```

This makes the code a bit more concise.

```
1  int countEval(String s, boolean result) {
2    if (s.length() == 0) return 0;
3    if (s.length() == 1) return stringToBool(s) == result ? 1 : 0;
4
5    int ways = 0;
6    for (int i = 1; i < s.length(); i += 2) {
7      char c = s.charAt(i);
8      String left = s.substring(0, i);
9      String right = s.substring(i + 1, s.length());
10
11     /* Evaluate each side for each result. */
12     int leftTrue = countEval(left, true);
13     int leftFalse = countEval(left, false);
14     int rightTrue = countEval(right, true);
```

```
15      int rightFalse = countEval(right, false);
16      int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);
17
18      int totalTrue = 0;
19      if (c == '^') { // required: one true and one false
20        totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
21      } else if (c == '&') { // required: both true
22        totalTrue = leftTrue * rightTrue;
23      } else if (c == '|') { // required: anything but both false
24        totalTrue = leftTrue * rightTrue + leftFalse * rightTrue +
25                    leftTrue * rightFalse;
26      }
27
28      int subWays = result ? totalTrue : total - totalTrue;
29      ways += subWays;
30    }
31
32    return ways;
33  }
34
35  boolean stringToBool(String c) {
36    return c.equals("1") ? true : false;
37  }
```

Note that the tradeoff of computing the `false` results from the `true` ones, and of computing the {`leftTrue, rightTrue, leftFalse, and rightFalse`} values upfront, is a small amount of extra work in some cases. For example, if we're looking for the ways that an AND (&) can result in `true`, we never would have needed the `leftFalse` and `rightFalse` results. Likewise, if we're looking for the ways that an OR (|) can result in `false`, we never would have needed the `leftTrue` and `rightTrue` results.

Our current code is blind to what we do and don't actually need to do and instead just computes all of the values. This is probably a reasonable tradeoff to make (especially given the constraints of whiteboard coding) as it makes our code substantially shorter and less tedious to write. Whichever approach you make, you should discuss the tradeoffs with your interviewer.

That said, there are more important optimizations we can make.

#### Optimized Solutions

If we follow the recursive path, we'll note that we end up doing the same computation repeatedly.

Consider the expression `0^0&0^1|1` and these recursion paths:

- Add parens around char 1. `(0)^(0&0^1|1)`
  - » Add parens around char 3. `(0)^((0)&(0^1|1))`
- Add parens around char 3. `(0^0)&(0^1|1)`
  - » Add parens around char 1. `((0)^(0))&(0^1|1)`

Although these two expressions are different, they have a similar component: `(0^1|1)`. We should reuse our effort on this.

We can do this by using memoization, or a hash table. We just need to store the result of `countEval(expression, result)` for each expression and result. If we see an expression that we've calculated before, we just return it from the cache.

```
1  int countEval(String s, boolean result, HashMap<String, Integer> memo) {
2    if (s.length() == 0) return 0;
3    if (s.length() == 1) return stringToBool(s) == result ? 1 : 0;
```

```
4     if (memo.containsKey(result + s)) return memo.get(result + s);
5
6     int ways = 0;
7
8     for (int i = 1; i < s.length(); i += 2) {
9       char c = s.charAt(i);
10      String left = s.substring(0, i);
11      String right = s.substring(i + 1, s.length());
12      int leftTrue = countEval(left, true, memo);
13      int leftFalse = countEval(left, false, memo);
14      int rightTrue = countEval(right, true, memo);
15      int rightFalse = countEval(right, false, memo);
16      int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);
17
18      int totalTrue = 0;
19      if (c == '^') {
20        totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
21      } else if (c == '&') {
22        totalTrue = leftTrue * rightTrue;
23      } else if (c == '|') {
24        totalTrue = leftTrue * rightTrue + leftFalse * rightTrue +
25                    leftTrue * rightFalse;
26      }
27
28      int subWays = result ? totalTrue : total - totalTrue;
29      ways += subWays;
30    }
31
32    memo.put(result + s, ways);
33    return ways;
34  }
```

The added benefit of this is that we could actually end up with the same substring in multiple parts of the expression. For example, an expression like `0^1^0&0^1^0` has two instances of `0^1^0`. By caching the result of the substring value in a memoization table, we'll get to reuse the result for the right part of the expression after computing it for the left.

There is one further optimization we can make, but it's far beyond the scope of the interview. There *is* a closed form expression for the number of ways of parenthesizing an expression, but you wouldn't be expected to know it. It is given by the Catalan numbers, where n is the number of operators:

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

We could use this to compute the total ways of evaluating the expression. Then, rather than computing `leftTrue` and `leftFalse`, we just compute one of those and calculate the other using the Catalan numbers. We would do the same thing for the right side.