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

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

Optimal Algorithm

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

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

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

Update `maxLength` as we go.

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

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

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

pg 116

SOLUTION

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

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

The Brute Force Approach

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

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

Bit Manipulation Approach for Get Next Number

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

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

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

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

We know the following:

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

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

Step 1: Flip rightmost non-trailing zero

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

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

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

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

Let's walk through this with an example.

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

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

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

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

Or, more concisely, we do:

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

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

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

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

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

Or, more concisely:

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

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

The code for `getNext` is below.

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

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

Bit Manipulation Approach for Get Previous Number

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

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

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

Let's walk through this with an example.

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

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

Steps 2 & 3: Clear bits 0 through p.

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

We can do this as follows:

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

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

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

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

We can do this as follows:

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

The code to implement this is below.

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

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

Arithmetic Approach to Get Next Number

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

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

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

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

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

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

This math reduces to:

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

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

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

Arithmetic Approach to Get Previous Number

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

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

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

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

This reduces mathematically to:

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

Again, this is very easy to implement.

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

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

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

pg 116

SOLUTION

We can work backwards to solve this question.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

EXAMPLE

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

Output: `2`

pg 116

SOLUTION

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

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

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

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

The code below utilizes this approach.

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

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

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

pg 116

SOLUTION

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

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

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

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

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

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

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

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

The method signature should look something like:

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

pg 116

SOLUTION

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

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

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

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

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

6

Solutions to Math and Logic Puzzles

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

pg 122

SOLUTION

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

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

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

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

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

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

This formula will tell you the bottle number:

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

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