4.4 **Check Balanced:** Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.

Hints: #21, #33, #49, #105, #124

pg 244

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

Hints: #35, #57, #86, #113, #128

pg 245

4.6 **Successor:** Write an algorithm to find the "next" node (i.e., in-order successor) of a given node in a binary search tree. You may assume that each node has a link to its parent.

Hints: #79, #91

pg 248

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

EXAMPLE

Input:

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

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

Hints: #26, #47, #60, #85, #125, #133

pg 250

4.8 **First Common Ancestor:** Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.

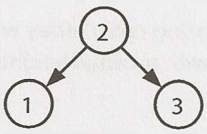
Hints: #10, #16, #28, #36, #46, #70, #80, #96

pg 257

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

EXAMPLE

Input:



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

Hints: #39, #48, #66, #82

pg 262

4.10 **Check Subtree:** T1 and T2 are two very large binary trees, with T1 much bigger than T2. Create an algorithm to determine if T2 is a subtree of T1.

A tree T2 is a subtree of T1 if there exists a node n in T1 such that the subtree of n is identical to T2. That is, if you cut off the tree at node n, the two trees would be identical.

Hints: #4, #11, #18, #31, #37

pg 265

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

Hints: #42, #54, #62, #75, #89, #99, #112, #119

pg 268

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

Hints: #6, #14, #52, #68, #77, #87, #94, #103, #108, #115

pg 272

Additional Questions: Recursion (#8.10), System Design and Scalability (#9.2, #9.3), Sorting and Searching (#10.10), Hard Problems (#17.7, #17.12, #17.13, #17.14, #17.17, #17.20, #17.22, #17.25).

Hints start on page 653.

5

Bit Manipulation

Bit manipulation is used in a variety of problems. Sometimes, the question explicitly calls for bit manipulation. Other times, it's simply a useful technique to optimize your code. You should be comfortable doing bit manipulation by hand, as well as with code. Be careful; it's easy to make little mistakes.

Bit Manipulation By Hand

If you're rusty on bit manipulation, try the following exercises by hand. The items in the third column can be solved manually or with "tricks" (described below). For simplicity, assume that these are four-bit numbers.

If you get confused, work them through as a base 10 number. You can then apply the same process to a binary number. Remember that ^ indicates an XOR, and ~ is a NOT (negation).

`0110 + 0010`	`0011 * 0101`	`0110 + 0110`
`0011 + 0010`	`0011 * 0011`	`0100 * 0011`
`0110 - 0011`	`1101 >> 2`	`1101 ^ (~1101)`
`1000 - 0110`	`1101 ^ 0101`	`1011 & (~0 << 2)`

Solutions: line 1 (1000, 1111, 1100); line 2 (0101, 1001, 1100); line 3 (0011, 0011, 1111); line 4 (0010, 1000, 1000).

The tricks in Column 3 are as follows:

1. `0110 + 0110` is equivalent to `0110 * 2`, which is equivalent to shifting `0110` left by 1.
2. `0100` equals 4, and multiplying by 4 is just left shifting by 2. So we shift `0011` left by 2 to get `1100`.
3. Think about this operation bit by bit. If you XOR a bit with its own negated value, you will always get 1. Therefore, the solution to `a^(~a)` will be a sequence of 1s.
4. `~0` is a sequence of 1s, so `~0 << 2` is 1s followed by two 0s. ANDing that with another value will clear the last two bits of the value.

If you didn't see these tricks immediately, think about them logically.

Bit Facts and Tricks

The following expressions are useful in bit manipulation. Don't just memorize them, though; think deeply about why each of these is true. We use "1s" and "0s" to indicate a sequence of 1s or 0s, respectively.

```
x ^ 0s = x      x & 0s = 0      x | 0s = x
x ^ 1s = ~x     x & 1s = x      x | 1s = 1s
 x ^ x = 0       x & x = x       x | x = x
```

To understand these expressions, recall that these operations occur bit-by-bit, with what's happening on one bit never impacting the other bits. This means that if one of the above statements is true for a single bit, then it's true for a sequence of bits.

Two's Complement and Negative Numbers

Computers typically store integers in two's complement representation. A positive number is represented as itself while a negative number is represented as the two's complement of its absolute value (with a 1 in its sign bit to indicate that a negative value). The two's complement of an N-bit number (where N is the number of bits used for the number, *excluding* the sign bit) is the complement of the number with respect to 2^N.

Let's look at the 4-bit integer -3 as an example. If it's a 4-bit number, we have one bit for the sign and three bits for the value. We want the complement with respect to 2^3, which is 8. The complement of 3 (the absolute value of -3) with respect to 8 is 5. 5 in binary is 101. Therefore, -3 in binary as a 4-bit number is 1101, with the first bit being the sign bit.

In other words, the binary representation of -K (negative K) as a N-bit number is `concat(1, 2^(N-1) - K)`.

Another way to look at this is that we invert the bits in the positive representation and then add 1. 3 is 011 in binary. Flip the bits to get 100, add 1 to get 101, then prepend the sign bit (1) to get 1101.

In a four-bit integer, this would look like the following.

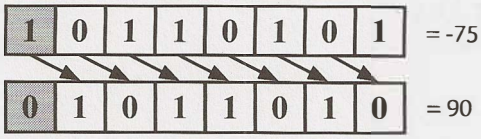
Positive Values		Negative Values	
7	0 111	-1	1 111
6	0 110	-2	1 110
5	0 101	-3	1 101
4	0 100	-4	1 100
3	0 011	-5	1 011
2	0 010	-6	1 010
1	0 001	-7	1 001
0	0 000		

Observe that the absolute values of the integers on the left and right always sum to 2^3, and that the binary values on the left and right sides are identical, other than the sign bit. Why is that?

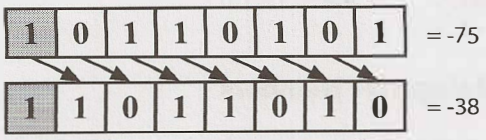
Arithmetic vs. Logical Right Shift

There are two types of right shift operators. The arithmetic right shift essentially divides by two. The logical right shift does what we would visually see as shifting the bits. This is best seen on a negative number.

In a logical right shift, we shift the bits and put a 0 in the most significant bit. It is indicated with a >>> operator. On an 8-bit integer (where the sign bit is the most significant bit), this would look like the image below. The sign bit is indicated with a gray background.



In an arithmetic right shift, we shift values to the right but fill in the new bits with the value of the sign bit. This has the effect of (roughly) dividing by two. It is indicated by a >> operator.



What do you think these functions would do on parameters `x = -93242` and `count = 40`?

```
1  int repeatedArithmeticShift(int x, int count) {
2    for (int i = 0; i < count; i++) {
3      x >>= 1; // Arithmetic shift by 1
4    }
5    return x;
6  }
7
8  int repeatedLogicalShift(int x, int count) {
9    for (int i = 0; i < count; i++) {
10     x >>>= 1; // Logical shift by 1
11   }
12   return x;
13 }
```

With the logical shift, we would get `0` because we are shifting a zero into the most significant bit repeatedly.

With the arithmetic shift, we would get `-1` because we are shifting a one into the most significant bit repeatedly. A sequence of all 1s in a (signed) integer represents `-1`.

▸ Common Bit Tasks: Getting and Setting

The following operations are very important to know, but do not simply memorize them. Memorizing leads to mistakes that are impossible to recover from. Rather, understand *how* to implement these methods, so that you can implement these, and other, bit problems.

Get Bit

This method shifts 1 over by `i` bits, creating a value that looks like `00010000`. By performing an AND with `num`, we clear all bits other than the bit at bit `i`. Finally, we compare that to `0`. If that new value is not zero, then bit `i` must have a 1. Otherwise, bit `i` is a 0.

```
1  boolean getBit(int num, int i) {
2    return ((num & (1 << i)) != 0);
3  }
```

Set Bit

`SetBit` shifts 1 over by `i` bits, creating a value like `00010000`. By performing an OR with `num`, only the value at bit `i` will change. All other bits of the mask are zero and will not affect `num`.

```
1  int setBit(int num, int i) {
2    return num | (1 << i);
3  }
```

Clear Bit

This method operates in almost the reverse of `setBit`. First, we create a number like `11101111` by creating the reverse of it (`00010000`) and negating it. Then, we perform an AND with `num`. This will clear the `i`th bit and leave the remainder unchanged.

```
1  int clearBit(int num, int i) {
2    int mask = ~(1 << i);
3    return num & mask;
4  }
```

To clear all bits from the most significant bit through `i` (inclusive), we create a mask with a `1` at the `i`th bit (`1 << i`). Then, we subtract `1` from it, giving us a sequence of `0`s followed by `i` `1`s. We then AND our number with this mask to leave just the last `i` bits.

```
1  int clearBitsMSBthroughI(int num, int i) {
2    int mask = (1 << i) - 1;
3    return num & mask;
4  }
```

To clear all bits from `i` through `0` (inclusive), we take a sequence of all `1`s (which is `-1`) and shift it left by `i + 1` bits. This gives us a sequence of `1`s (in the most significant bits) followed by `i` `0` bits.

```
1  int clearBitsIthrough0(int num, int i) {
2    int mask = (-1 << (i + 1));
3    return num & mask;
4  }
```

Update Bit

To set the `i`th bit to a value `v`, we first clear the bit at position `i` by using a mask that looks like `11101111`. Then, we shift the intended value, `v`, left by `i` bits. This will create a number with bit `i` equal to `v` and all other bits equal to `0`. Finally, we OR these two numbers, updating the `i`th bit if `v` is `1` and leaving it as `0` otherwise.

```
1  int updateBit(int num, int i, boolean bitIs1) {
2    int value = bitIs1 ? 1 : 0;
3    int mask = ~(1 << i);
4    return (num & mask) | (value << i);
5  }
```

Interview Questions

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`

Hints: #137, #169, #215

pg 276

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."

Hints: #143, #167, #173, #269, #297

pg 277

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 `1`s you could create.

EXAMPLE

Input: `1775 (or: 11011101111)`

Output: `8`

Hints: #159, #226, #314, #352

pg 278

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.

Hints: #147, #175, #242, #312, #339, #358, #375, #390

pg 280

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

Hints: #151, #202, #261, #302, #346, #372, #383, #398

pg 285

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`

Hints: #336, #369

pg 286

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).

Hints: #145, #248, #328, #355

pg 286

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)
```

Hints: #366, #381, #384, #391

pg 287

Additional Questions: Arrays and Strings (#1.1, #1.4, #1.8), Math and Logic Puzzles (#6.10), Recursion (#8.4, #8.14), Sorting and Searching (#10.7, #10.8), C++ (#12.10), Moderate Problems (#16.1, #16.7), Hard Problems (#17.1).

Hints start on page 662.

6

Math and Logic Puzzles

So-called "puzzles" (or brain teasers) are some of the most hotly debated questions, and many companies have policies banning them. Unfortunately, even when these questions are banned, you still may find yourself being asked one of them. Why? Because no one can agree on a definition of what a brainteaser is.

The good news is that if you are asked a puzzle or brainteaser, it's likely to be a reasonably fair one. It probably won't rely on a trick of wording, and it can almost always be logically deduced. Many have their foundations in mathematics or computer science, and almost all have solutions that can be logically deduced.

We'll go through some common approaches for tackling these questions, as well as some of the essential knowledge.

Prime Numbers

As you probably know, every positive integer can be decomposed into a product of primes. For example:

$$84 = 2^2 * 3^1 * 5^0 * 7^1 * 11^0 * 13^0 * 17^0 * \ldots$$

Note that many of these primes have an exponent of zero.

Divisibility

The prime number law stated above means that, in order for a number x to divide a number y (written $x \backslash y$, or $\text{mod}(y, x) = 0$), all primes in x's prime factorization must be in y's prime factorization. Or, more specifically:

Let $x = 2^{j0} * 3^{j1} * 5^{j2} * 7^{j3} * 11^{j4} * \ldots$

Let $y = 2^{k0} * 3^{k1} * 5^{k2} * 7^{k3} * 11^{k4} * \ldots$

If $x \backslash y$, then for all i, $ji <= ki$.

In fact, the greatest common divisor of x and y will be:

$$\text{gcd}(x, y) = 2^{\min(j0, k0)} * 3^{\min(j1, k1)} * 5^{\min(j2, k2)} * \ldots$$

The least common multiple of x and y will be:

$$\text{lcm}(x, y) = 2^{\max(j0, k0)} * 3^{\max(j1, k1)} * 5^{\max(j2, k2)} * \ldots$$

As a fun exercise, stop for a moment and think what would happen if you did $\text{gcd} * \text{lcm}$:

$$\begin{aligned}\text{gcd} * \text{lcm} &= 2^{\min(j0, k0)} * 2^{\max(j0, k0)} * 3^{\min(j1, k1)} * 3^{\max(j1, k1)} * \ldots \\ &= 2^{\min(j0, k0) + \max(j0, k0)} * 3^{\min(j1, k1) + \max(j1, k1)} * \ldots \\ &= 2^{j0 + k0} * 3^{j1 + k1} * \ldots \\ &= 2^{j0} * 2^{k0} * 3^{j1} * 3^{k1} * \ldots\end{aligned}$$

$= xy$

Checking for Primality

This question is so common that we feel the need to specifically cover it. The naive way is to simply iterate from 2 through n-1, checking for divisibility on each iteration.

```
1  boolean primeNaive(int n) {
2    if (n < 2) {
3      return false;
4    }
5    for (int i = 2; i < n; i++) {
6      if (n % i == 0) {
7        return false;
8      }
9    }
10   return true;
11 }
```

A small but important improvement is to iterate only up through the square root of n.

```
1  boolean primeSlightlyBetter(int n) {
2    if (n < 2) {
3      return false;
4    }
5    int sqrt = (int) Math.sqrt(n);
6    for (int i = 2; i <= sqrt; i++) {
7      if (n % i == 0) return false;
8    }
9    return true;
10 }
```

The $\sqrt{n}$ is sufficient because, for every number a which divides n evenly, there is a complement b, where a * b = n. If $a > \sqrt{n}$, then $b < \sqrt{n}$ (since $(\sqrt{n})^2 = n$). We therefore don't need a to check n's primality, since we would have already checked with b.

Of course, in reality, all we *really* need to do is to check if n is divisible by a prime number. This is where the Sieve of Eratosthenes comes in.

Generating a List of Primes: The Sieve of Eratosthenes

The Sieve of Eratosthenes is a highly efficient way to generate a list of primes. It works by recognizing that all non-prime numbers are divisible by a prime number.

We start with a list of all the numbers up through some value max. First, we cross off all numbers divisible by 2. Then, we look for the next prime (the next non-crossed off number) and cross off all numbers divisible by it. By crossing off all numbers divisible by 2, 3, 5, 7, 11, and so on, we wind up with a list of prime numbers from 2 through max.

The code below implements the Sieve of Eratosthenes.

```
1  boolean[] sieveOfEratosthenes(int max) {
2    boolean[] flags = new boolean[max + 1];
3    int count = 0;
4
5    init(flags); // Set all flags to true other than 0 and 1
6    int prime = 2;
7
8    while (prime <= Math.sqrt(max)) {
```

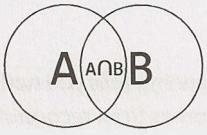
```
9        /* Cross off remaining multiples of prime */
10       crossOff(flags, prime);
11
12       /* Find next value which is true */
13       prime = getNextPrime(flags, prime);
14     }
15
16     return flags;
17   }
18
19   void crossOff(boolean[] flags, int prime) {
20     /* Cross off remaining multiples of prime. We can start with (prime*prime),
21      * because if we have a k * prime, where k < prime, this value would have
22      * already been crossed off in a prior iteration. */
23     for (int i = prime * prime; i < flags.length; i += prime) {
24       flags[i] = false;
25     }
26   }
27
28   int getNextPrime(boolean[] flags, int prime) {
29     int next = prime + 1;
30     while (next < flags.length && !flags[next]) {
31       next++;
32     }
33     return next;
34   }
```

Of course, there are a number of optimizations that can be made to this. One simple one is to only use odd numbers in the array, which would allow us to reduce our space usage by half.

Probability

Probability can be a complex topic, but it's based in a few basic laws that can be logically derived.

Let's look at a Venn diagram to visualize two events A and B. The areas of the two circles represent their relative probability, and the overlapping area is the event {A and B}.



Probability of A and B

Imagine you were throwing a dart at this Venn diagram. What is the probability that you would land in the intersection between A and B? If you knew the odds of landing in A, and you also knew the percent of A that's also in B (that is, the odds of being in B given that you were in A), then you could express the probability as:

```
P(A and B) = P(B given A) P(A)
```

For example, imagine we were picking a number between 1 and 10 (inclusive). What's the probability of picking an even number *and* a number between 1 and 5? The odds of picking a number between 1 and 5 is 50%, and the odds of a number between 1 and 5 being even is 40%. So, the odds of doing both are:

```
P(x is even and x <= 5)
```