What It Means

The most straightforward way is to think about what this means. This code touches each node in the tree once and does a constant time amount of work with each "touch" (excluding the recursive calls).

Therefore, the runtime will be linear in terms of the number of nodes. If there are N nodes, then the runtime is $O(N)$.

Recursive Pattern

On page 44, we discussed a pattern for the runtime of recursive functions that have multiple branches. Let's try that approach here.

We said that the runtime of a recursive function with multiple branches is typically $O(branches^{depth})$. There are two branches at each call, so we're looking at $O(2^{depth})$.

At this point many people might assume that something went wrong since we have an exponential algorithm—that something in our logic is flawed or that we've inadvertently created an exponential time algorithm (yikes!).

The second statement is correct. We do have an exponential time algorithm, but it's not as bad as one might think. Consider what variable it's exponential with respect to.

What is depth? The tree is a balanced binary search tree. Therefore, if there are N total nodes, then depth is roughly $\log N$.

By the equation above, we get $O(2^{\log N})$.

Recall what $\log_2$ means:

$2^P = Q \rightarrow \log_2 Q = P$

What is $2^{\log N}$? There is a relationship between 2 and log, so we should be able to simplify this.

Let $P = 2^{\log N}$. By the definition of $\log_2$, we can write this as $\log_2 P = \log_2 N$. This means that $P = N$.

$$\begin{aligned} &\text{Let } P = 2^{\log N} \\ &\quad \rightarrow \log_2 P = \log_2 N \\ &\quad \rightarrow P = N \\ &\quad \rightarrow 2^{\log N} = N \end{aligned}$$

Therefore, the runtime of this code is $O(N)$, where N is the number of nodes.

Example 10

The following method checks if a number is prime by checking for divisibility on numbers less than it. It only needs to go up to the square root of n because if n is divisible by a number greater than its square root then it's divisible by something smaller than it.

For example, while 33 is divisible by 11 (which is greater than the square root of 33), the "counterpart" to 11 is 3 (3 * 11 = 33). 33 will have already been eliminated as a prime number by 3.

What is the time complexity of this function?

```
1  boolean isPrime(int n) {
2    for (int x = 2; x * x <= n; x++) {
3      if (n % x == 0) {
4        return false;
5      }
6    }
7    return true;
```

```
8   }
```

Many people get this question wrong. If you're careful about your logic, it's fairly easy.

The work inside the for loop is constant. Therefore, we just need to know how many iterations the for loop goes through in the worst case.

The for loop will start when `x = 2` and end when `x*x = n`. Or, in other words, it stops when $x = \sqrt{n}$ (when x equals the square root of n).

This for loop is really something like this:

```
1   boolean isPrime(int n) {
2     for (int x = 2; x <= sqrt(n); x++) {
3       if (n % x == 0) {
4         return false;
5       }
6     }
7     return true;
8   }
```

This runs in $O(\sqrt{n})$ time.

Example 11

The following code computes n! (n factorial). What is its time complexity?

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

This is just a straight recursion from n to n-1 to n-2 down to 1. It will take $O(n)$ time.

Example 12

This code counts all permutations of a string.

```
1   void permutation(String str) {
2     permutation(str, "");
3   }
4
5   void permutation(String str, String prefix) {
6     if (str.length() == 0) {
7       System.out.println(prefix);
8     } else {
9       for (int i = 0; i < str.length(); i++) {
10        String rem = str.substring(0, i) + str.substring(i + 1);
11        permutation(rem, prefix + str.charAt(i));
12      }
13    }
14  }
```

This is a (very!) tricky one. We can think about this by looking at how many times `permutation` gets called and how long each call takes. We'll aim for getting as tight of an upper bound as possible.

How many times does permutation get called in its base case?

If we were to generate a permutation, then we would need to pick characters for each "slot." Suppose we had 7 characters in the string. In the first slot, we have 7 choices. Once we pick the letter there, we have 6 choices for the next slot. (Note that this is 6 choices *for each* of the 7 choices earlier.) Then 5 choices for the next slot, and so on.

Therefore, the total number of options is 7 * 6 * 5 * 4 * 3 * 2 * 1, which is also expressed as 7! (7 factorial).

This tells us that there are n! permutations. Therefore, `permutation` is called `n!` times in its base case (when `prefix` is the full permutation).

How many times does permutation get called before its base case?

But, of course, we also need to consider how many times lines 9 through 12 are hit. Picture a large call tree representing all the calls. There are `n!` leaves, as shown above. Each leaf is attached to a path of length n. Therefore, we know there will be no more than `n * n!` nodes (function calls) in this tree.

How long does each function call take?

Executing line 7 takes `O(n)` time since each character needs to be printed.

Line 10 and line 11 will also take `O(n)` time combined, due to the string concatenation. Observe that the sum of the lengths of `rem`, `prefix`, and `str.charAt(i)` will always be n.

Each node in our call tree therefore corresponds to `O(n)` work.

What is the total runtime?

Since we are calling `permutation` `O(n * n!)` times (as an upper bound), and each one takes `O(n)` time, the total runtime will not exceed $O(n^2 * n!)$.

Through more complex mathematics, we can derive a tighter runtime equation (though not necessarily a nice closed-form expression). This would almost certainly be beyond the scope of any normal interview.

Example 13

The following code computes the Nth Fibonacci number.

```
1  int fib(int n) {
2    if (n <= 0) return 0;
3    else if (n == 1) return 1;
4    return fib(n - 1) + fib(n - 2);
5  }
```

We can use the earlier pattern we'd established for recursive calls: $O(branches^{depth})$.

There are 2 branches per call, and we go as deep as N, therefore the runtime is $O(2^N)$.

> Through some very complicated math, we can actually get a tighter runtime. The time is indeed exponential, but it's actually closer to $O(1.6^N)$. The reason that it's not exactly $O(2^N)$ is that, at the bottom of the call stack, there is sometimes only one call. It turns out that a lot of the nodes are at the bottom (as is true in most trees), so this single versus double call actually makes a big difference. Saying $O(2^N)$ would suffice for the scope of an interview, though (and is still technically correct, if you read the note about big theta on page 39). You might get "bonus points" if you can recognize that it'll actually be less than that.

Generally speaking, when you see an algorithm with multiple recursive calls, you're looking at exponential runtime.

Example 14

The following code prints all Fibonacci numbers from 0 to n. What is its time complexity?

```
1  void allFib(int n) {
2    for (int i = 0; i < n; i++) {
3      System.out.println(i + ": " + fib(i));
4    }
5  }
6
7  int fib(int n) {
8    if (n <= 0) return 0;
9    else if (n == 1) return 1;
10   return fib(n - 1) + fib(n - 2);
11 }
```

Many people will rush to concluding that since `fib(n)` takes $O(2^n)$ time and it's called n times, then it's $O(n2^n)$.

Not so fast. Can you find the error in the logic?

The error is that the n is changing. Yes, `fib(n)` takes $O(2^n)$ time, but it matters what that value of n is.

Instead, let's walk through each call.

```
fib(1) -> 2^1 steps
fib(2) -> 2^2 steps
fib(3) -> 2^3 steps
fib(4) -> 2^4 steps
...
fib(n) -> 2^n steps
```

Therefore, the total amount of work is:

$$2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^n$$

As we showed on page 44, this is 2^{n+1}. Therefore, the runtime to compute the first n Fibonacci numbers (using this terrible algorithm) is still $O(2^n)$.

Example 15

The following code prints all Fibonacci numbers from 0 to n. However, this time, it stores (i.e., caches) previously computed values in an integer array. If it has already been computed, it just returns the cache. What is its runtime?

```
1  void allFib(int n) {
2    int[] memo = new int[n + 1];
3    for (int i = 0; i < n; i++) {
4      System.out.println(i + ": " + fib(i, memo));
5    }
6  }
7
8  int fib(int n, int[] memo) {
9    if (n <= 0) return 0;
10   else if (n == 1) return 1;
11   else if (memo[n] > 0) return memo[n];
12
13   memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
```

```
14    return memo[n];
15  }
```

Let's walk through what this algorithm does.

```
fib(1) -> return 1
fib(2)
    fib(1) -> return 1
    fib(0) -> return 0
    store 1 at memo[2]
fib(3)
    fib(2) -> lookup memo[2] -> return 1
    fib(1) -> return 1
    store 2 at memo[3]
fib(4)
    fib(3) -> lookup memo[3] -> return 2
    fib(2) -> lookup memo[2] -> return 1
    store 3 at memo[4]
fib(5)
    fib(4) -> lookup memo[4] -> return 3
    fib(3) -> lookup memo[3] -> return 2
    store 5 at memo[5]
...
```

At each call to `fib(i)`, we have already computed and stored the values for `fib(i-1)` and `fib(i-2)`. We just look up those values, sum them, store the new result, and return. This takes a constant amount of time.

We're doing a constant amount of work `N` times, so this is `O(n)` time.

This technique, called memoization, is a very common one to optimize exponential time recursive algorithms.

Example 16

The following function prints the powers of 2 from 1 through n (inclusive). For example, if n is 4, it would print 1, 2, and 4. What is its runtime?

```
1   int powersOf2(int n) {
2     if (n < 1) {
3       return 0;
4     } else if (n == 1) {
5       System.out.println(1);
6       return 1;
7     } else {
8       int prev = powersOf2(n / 2);
9       int curr = prev * 2;
10      System.out.println(curr);
11      return curr;
12    }
13  }
```

There are several ways we could compute this runtime.

What It Does

Let's walk through a call like `powersOf2(50)`.

```
powersOf2(50)
    -> powersOf2(25)
```

```
-> powersOf2(12)
    -> powersOf2(6)
        -> powersOf2(3)
            -> powersOf2(1)
                -> print & return 1
            print & return 2
        print & return 4
    print & return 8
print & return 16
print & return 32
```

The runtime, then, is the number of times we can divide 50 (or n) by 2 until we get down to the base case (1). As we discussed on page 44, the number of times we can halve n until we get 1 is $O(\log n)$.

What It Means

We can also approach the runtime by thinking about what the code is supposed to be doing. It's supposed to be computing the powers of 2 from 1 through n.

Each call to `powersOf2` results in exactly one number being printed and returned (excluding what happens in the recursive calls). So if the algorithm prints 13 values at the end, then `powersOf2` was called 13 times.

In this case, we are told that it prints all the powers of 2 between 1 and n. Therefore, the number of times the function is called (which will be its runtime) must equal the number of powers of 2 between 1 and n.

There are $\log N$ powers of 2 between 1 and n. Therefore, the runtime is $O(\log n)$.

Rate of Increase

A final way to approach the runtime is to think about how the runtime changes as n gets bigger. After all, this is exactly what big O time means.

If N goes from P to P+1, the number of calls to `powersOfTwo` might not change at all. When will the number of calls to `powersOfTwo` increase? It will increase by 1 each time n doubles in size.

So, each time n doubles, the number of calls to `powersOfTwo` increases by 1. Therefore, the number of calls to `powersOfTwo` is the number of times you can double 1 until you get n. It is x in the equation $2^x = n$.

What is x? The value of x is $\log n$. This is exactly what meant by $x = \log n$.

Therefore, the runtime is $O(\log n)$.

Additional Problems

VI.1 The following code computes the product of a and b. What is its runtime?

```
int product(int a, int b) {
    int sum = 0;
    for (int i = 0; i < b; i++) {
        sum += a;
    }
    return sum;
}
```

VI.2 The following code computes a^b. What is its runtime?

```
int power(int a, int b) {
    if (b < 0) {
```

```
            return 0; // error
        } else if (b == 0) {
            return 1;
        } else {
            return a * power(a, b - 1);
        }
    }
```

VI.3 The following code computes `a % b`. What is its runtime?

```
int mod(int a, int b) {
    if (b <= 0) {
        return -1;
    }
    int div = a / b;
    return a - div * b;
}
```

VI.4 The following code performs integer division. What is its runtime (assume a and b are both positive)?

```
int div(int a, int b) {
    int count = 0;
    int sum = b;
    while (sum <= a) {
        sum += b;
        count++;
    }
    return count;
}
```

VI.5 The following code computes the [integer] square root of a number. If the number is not a perfect square (there is no integer square root), then it returns -1. It does this by successive guessing. If n is 100, it first guesses 50. Too high? Try something lower – halfway between 1 and 50. What is its runtime?

```
int sqrt(int n) {
    return sqrt_helper(n, 1, n);
}

int sqrt_helper(int n, int min, int max) {
    if (max < min) return -1; // no square root

    int guess = (min + max) / 2;
    if (guess * guess == n) { // found it!
        return guess;
    } else if (guess * guess < n) { // too low
        return sqrt_helper(n, guess + 1, max); // try higher
    } else { // too high
        return sqrt_helper(n, min, guess - 1); // try lower
    }
}
```

VI.6 The following code computes the [integer] square root of a number. If the number is not a perfect square (there is no integer square root), then it returns -1. It does this by trying increasingly large numbers until it finds the right value (or is too high). What is its runtime?

```
int sqrt(int n) {
    for (int guess = 1; guess * guess <= n; guess++) {
        if (guess * guess == n) {
            return guess;
```

```
            }
        }
        return -1;
    }
```

VI.7 **If a binary search tree is not balanced, how long might it take (worst case) to find an element in it?**

VI.8 **You are looking for a specific value in a binary tree, but the tree is not a binary search tree. *What is the time complexity of this?***

VI.9 **The `appendToNew` method appends a value to an array by creating a new, longer array and returning this longer array. You've used the `appendToNew` method to create a `copyArray` function that repeatedly calls `appendToNew`. How long does copying an array take?**

```
int[] copyArray(int[] array) {
    int[] copy = new int[0];
    for (int value : array) {
        copy = appendToNew(copy, value);
    }
    return copy;
}

int[] appendToNew(int[] array, int value) {
    // copy all elements over to new array
    int[] bigger = new int[array.length + 1];
    for (int i = 0; i < array.length; i++) {
        bigger[i] = array[i];
    }

    // add new element
    bigger[bigger.length - 1] = value;
    return bigger;
}
```

VI.10 **The following code sums the digits in a number. What is its big O time?**

```
int sumDigits(int n) {
    int sum = 0;
    while (n > 0) {
        sum += n % 10;
        n /= 10;
    }
    return sum;
}
```

VI.11 **The following code prints all strings of length k where the characters are in sorted order. It does this by generating all strings of length k and then checking if each is sorted. What is its runtime?**

```
int numChars = 26;

void printSortedStrings(int remaining) {
    printSortedStrings(remaining, "");
}

void printSortedStrings(int remaining, String prefix) {
    if (remaining == 0) {
        if (isInOrder(prefix)) {
            System.out.println(prefix);
        }
```

```
        } else {
            for (int i = 0; i < numChars; i++) {
                char c = ithLetter(i);
                printSortedStrings(remaining - 1, prefix + c);
            }
        }
    }

    boolean isInOrder(String s) {
        for (int i = 1; i < s.length(); i++) {
            int prev = ithLetter(s.charAt(i - 1));
            int curr = ithLetter(s.charAt(i));
            if (prev > curr) {
                return false;
            }
        }
        return true;
    }

    char ithLetter(int i) {
        return (char) (((int) 'a') + i);
    }
```

VI.12 The following code computes the intersection (the number of elements in common) of two arrays. It assumes that neither array has duplicates. It computes the intersection by sorting one array (array b) and then iterating through array a checking (via binary search) if each value is in b. What is its runtime?

```
int intersection(int[] a, int[] b) {
    mergesort(b);
    int intersect = 0;

    for (int x : a) {
        if (binarySearch(b, x) >= 0) {
            intersect++;
        }
    }

    return intersect;
}
```

Solutions

1. O(b). The for loop just iterates through b.
2. O(b). The recursive code iterates through b calls, since it subtracts one at each level.
3. O(1). It does a constant amount of work.
4. O(a/b). The variable `count` will eventually equal a/b. The while loop iterates count times. Therefore, it iterates a/b times.
5. O(log n). This algorithm is essentially doing a binary search to find the square root. Therefore, the runtime is O(log n).
6. O(sqrt(n)). This is just a straightforward loop that stops when `guess*guess > n` (or, in other words, when `guess > sqrt(n)`).

7. $O(n)$, where n is the number of nodes in the tree. The max time to find an element is the depth tree. The tree could be a straight list downwards and have depth n.

8. $O(n)$. Without any ordering property on the nodes, we might have to search through all the nodes.

9. $O(n^2)$, where n is the number of elements in the array. The first call to appendToNew takes 1 copy. The second call takes 2 copies. The third call takes 3 copies. And so on. The total time will be the sum of 1 through n, which is $O(n^2)$.

10. $O(\log n)$. The runtime will be the number of digits in the number. A number with d digits can have a value up to 10^d. If $n = 10^d$, then $d = \log n$. Therefore, the runtime is $O(\log n)$.

11. $O(kc^k)$, where k is the length of the string and c is the number of characters in the alphabet. It takes $O(c^k)$ time to generate each string. Then, we need to check that each of these is sorted, which takes $O(k)$ time.

12. $O(b \log b + a \log b)$. First, we have to sort array b, which takes $O(b \log b)$ time. Then, for each element in a, we do binary search in $O(\log b)$ time. The second part takes $O(a \log b)$ time.