Let's look at this from the perspective of quick sort. Quick sort picks a random element as a "pivot" and then swaps values in the array such that the elements less than pivot appear before elements greater than pivot. This gives a "partial sort." Then it recursively sorts the left and right sides using a similar process.

- **Best Case:** If all elements are equal, then quick sort will, on average, just traverse through the array once. This is O(N). (This actually depends slightly on the implementation of quick sort. There are implementations, though, that will run very quickly on a sorted array.)
- **Worst Case:** What if we get really unlucky and the pivot is repeatedly the biggest element in the array? (Actually, this can easily happen. If the pivot is chosen to be the first element in the subarray and the array is sorted in reverse order, we'll have this situation.) In this case, our recursion doesn't divide the array in half and recurse on each half. It just shrinks the subarray by one element. This will degenerate to an O(N^2) runtime.
- **Expected Case:** Usually, though, these wonderful or terrible situations won't happen. Sure, sometimes the pivot will be very low or very high, but it won't happen over and over again. We can expect a runtime of O(N log N).

We rarely ever discuss best case time complexity, because it's not a very useful concept. After all, we could take essentially any algorithm, special case some input, and then get an O(1) time in the best case.

For many—probably most—algorithms, the worst case and the expected case are the same. Sometimes they're different, though, and we need to describe both of the runtimes.

What is the relationship between best/worst/expected case and big O/theta/omega?

It's easy for candidates to muddle these concepts (probably because both have some concepts of "higher", "lower" and "exactly right"), but there is no particular relationship between the concepts.

Best, worst, and expected cases describe the big O (or big theta) time for particular inputs or scenarios.

Big O, big omega, and big theta describe the upper, lower, and tight bounds for the runtime.

Space Complexity

Time is not the only thing that matters in an algorithm. We might also care about the amount of memory—or space—required by an algorithm.

Space complexity is a parallel concept to time complexity. If we need to create an array of size n, this will require O(n) space. If we need a two-dimensional array of size nxn, this will require O(n^2) space.

Stack space in recursive calls counts, too. For example, code like this would take O(n) time and O(n) space.

```
int sum(int n) { /* Ex 1.*/
  if (n <= 0) {
    return 0;
  }
  return n + sum(n-1);
}
```

Each call adds a level to the stack.

```
sum(4)
  -> sum(3)
    -> sum(2)
      -> sum(1)
        -> sum(0)
```

Each of these calls is added to the call stack and takes up actual memory.

However, just because you have n calls total doesn't mean it takes O(n) space. Consider the below function, which adds adjacent elements between 0 and n:

```
int pairSumSequence(int n) { /* Ex 2.*/
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += pairSum(i, i + 1);
  }
  return sum;
}

int pairSum(int a, int b) {
  return a + b;
}
```

There will be roughly O(n) calls to pairSum. However, those calls do not exist simultaneously on the call stack, so you only need O(1) space.

Drop the Constants

It is very possible for O(N) code to run faster than O(1) code for specific inputs. Big O just describes the rate of increase.

For this reason, we drop the constants in runtime. An algorithm that one might have described as O(2N) is actually O(N).

Many people resist doing this. They will see code that has two (non-nested) for loops and continue this O(2N). They think they're being more "precise." They're not.

Consider the below code:

Min and Max 1

```
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int x : array) {
  if (x < min) min = x;
  if (x > max) max = x;
}
```

Min and Max 2

```
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int x : array) {
  if (x < min) min = x;
}
for (int x : array) {
  if (x > max) max = x;
}
```

Which one is faster? The first one does one for loop and the other one does two for loops. But then, the first solution has two lines of code per for loop rather than one.

If you're going to count the number of instructions, then you'd have to go to the assembly level and take into account that multiplication requires more instructions than addition, how the compiler would optimize something, and all sorts of other details.

This would be horrendously complicated, so don't even start going down this road. Big O allows us to express how the runtime scales. We just need to accept that it doesn't mean that O(N) is always better than $O(N^2)$.

Drop the Non-Dominant Terms

What do you do about an expression such as O(N^2 + N)? That second N isn't exactly a constant. But it's not especially important.

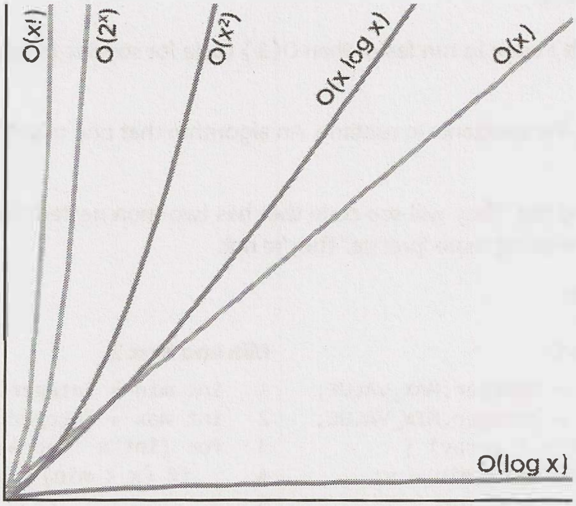
We already said that we drop constants. Therefore, O(N^2 + N^2) would be O(N^2). If we don't care about that latter N^2 term, why would we care about N? We don't.

You should drop the non-dominant terms.

- O(N^2 + N) becomes O(N^2).
- O(N + log N) becomes O(N).
- O(5*2^N + 1000N^{100}) becomes O(2^N).

We might still have a sum in a runtime. For example, the expression O(B^2 + A) cannot be reduced (without some special knowledge of A and B).

The following graph depicts the rate of increase for some of the common big O times.



As you can see, O(x^2) is much worse than O(x), but it's not nearly as bad as O(2^x) or O(x!). There are lots of runtimes worse than O(x!) too, such as O(x^x) or O(2^x * x!).

Multi-Part Algorithms: Add vs. Multiply

Suppose you have an algorithm that has two steps. When do you multiply the runtimes and when do you add them?

This is a common source of confusion for candidates.

Add the Runtimes: O(A + B)

```
for (int a : arrA) {
  print(a);
}

for (int b : arrB) {
  print(b);
}
```

Multiply the Runtimes: O(A*B)

```
for (int a : arrA) {
  for (int b : arrB) {
    print(a + "," + b);
  }
}
```

In the example on the left, we do A chunks of work then B chunks of work. Therefore, the total amount of work is O(A + B).

In the example on the right, we do B chunks of work for each element in A. Therefore, the total amount of work is O(A * B).

In other words:

- If your algorithm is in the form "do this, then, when you're all done, do that" then you add the runtimes.
- If your algorithm is in the form "do this for each time you do that" then you multiply the runtimes.

It's very easy to mess this up in an interview, so be careful.

Amortized Time

An ArrayList, or a dynamically resizing array, allows you to have the benefits of an array while offering flexibility in size. You won't run out of space in the ArrayList since its capacity will grow as you insert elements.

An ArrayList is implemented with an array. When the array hits capacity, the ArrayList class will create a new array with double the capacity and copy all the elements over to the new array.

How do you describe the runtime of insertion? This is a tricky question.

The array could be full. If the array contains N elements, then inserting a new element will take O(N) time. You will have to create a new array of size 2N and then copy N elements over. This insertion will take O(N) time.

However, we also know that this doesn't happen very often. The vast majority of the time insertion will be in O(1) time.

We need a concept that takes both into account. This is what amortized time does. It allows us to describe that, yes, this worst case happens every once in a while. But once it happens, it won't happen again for so long that the cost is "amortized."

In this case, what is the amortized time?

As we insert elements, we double the capacity when the size of the array is a power of 2. So after X elements, we double the capacity at array sizes 1, 2, 4, 8, 16, ..., X. That doubling takes, respectively, 1, 2, 4, 8, 16, 32, 64, ..., X copies.

What is the sum of 1 + 2 + 4 + 8 + 16 + ... + X? If you read this sum left to right, it starts with 1 and doubles until it gets to X. If you read right to left, it starts with X and halves until it gets to 1.

What then is the sum of $X + {}^{X}\!/_{2} + {}^{X}\!/_{4} + {}^{X}\!/_{8} + \ldots + 1$? This is roughly 2X.

Therefore, X insertions take O(2X) time. The amortized time for each insertion is O(1).

Log N Runtimes

We commonly see O(log N) in runtimes. Where does this come from?

Let's look at binary search as an example. In binary search, we are looking for an example x in an N-element sorted array. We first compare x to the midpoint of the array. If x == middle, then we return. If x < middle, then we search on the left side of the array. If x > middle, then we search on the right side of the array.

```
search 9 within {1, 5, 8, 9, 11, 13, 15, 19, 21}
    compare 9 to 11 -> smaller.
    search 9 within {1, 5, 8, 9, 11}
        compare 9 to 8 -> bigger
        search 9 within {9, 11}
            compare 9 to 9
            return
```

We start off with an N-element array to search. Then, after a single step, we're down to $\frac{N}{2}$ elements. One more step, and we're down to $\frac{N}{4}$ elements. We stop when we either find the value or we're down to just one element.

The total runtime is then a matter of how many steps (dividing N by 2 each time) we can take until N becomes 1.

```
N = 16
N = 8       /* divide by 2 */
N = 4       /* divide by 2 */
N = 2       /* divide by 2 */
N = 1       /* divide by 2 */
```

We could look at this in reverse (going from 1 to 16 instead of 16 to 1). How many times we can multiply 1 by 2 until we get N?

```
N = 1
N = 2       /* multiply by 2 */
N = 4       /* multiply by 2 */
N = 8       /* multiply by 2 */
N = 16      /* multiply by 2 */
```

What is k in the expression $2^k = N$? This is exactly what log expresses.

$$2^4 = 16 \rightarrow \log_2 16 = 4$$
$$\log_2 N = k \rightarrow 2^k = N$$

This is a good takeaway for you to have. When you see a problem where the number of elements in the problem space gets halved each time, that will likely be a O(log N) runtime.

This is the same reason why finding an element in a balanced binary search tree is O(log N). With each comparison, we go either left or right. Half the nodes are on each side, so we cut the problem space in half each time.

> What's the base of the log? That's an excellent question! The short answer is that it doesn't matter for the purposes of big O. The longer explanation can be found at "Bases of Logs" on page 630.

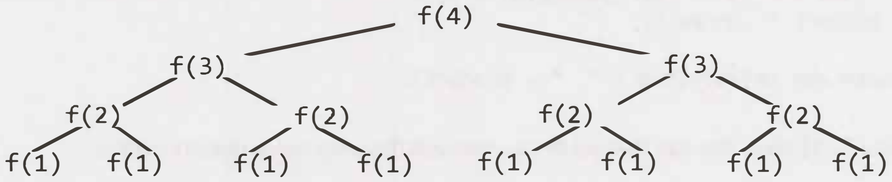
Recursive Runtimes

Here's a tricky one. What's the runtime of this code?

```
int f(int n) {
    if (n <= 1) {
       return 1;
    }
    return f(n - 1) + f(n - 1);
 }
```

A lot of people will, for some reason, see the two calls to f and jump to $O(N^2)$. This is completely incorrect.

Rather than making assumptions, let's derive the runtime by walking through the code. Suppose we call f(4). This calls f(3) twice. Each of those calls to f(3) calls f(2), until we get down to f(1).



How many calls are in this tree? (Don't count!)

The tree will have depth N. Each node (i.e., function call) has two children. Therefore, each level will have twice as many calls as the one above it. The number of nodes on each level is:

Level	# Nodes	Also expressed as...	Or...
0	1		2^0
1	2	2 * previous level = 2	2^1
2	4	2 * previous level = $2 * 2^1 = 2^2$	2^2
3	8	2 * previous level = $2 * 2^2 = 2^3$	2^3
4	16	2 * previous level = $2 * 2^3 = 2^4$	2^4

Therefore, there will be $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + ... + 2^N$ (which is $2^{N+1} - 1$) nodes. (See "Sum of Powers of 2" on page 630.)

Try to remember this pattern. When you have a recursive function that makes multiple calls, the runtime will often (but not always) look like $O(\text{branches}^{\text{depth}})$, where branches is the number of times each recursive call branches. In this case, this gives us $O(2^N)$.

> As you may recall, the base of a log doesn't matter for big O since logs of different bases are only different by a constant factor. However, this does not apply to exponents. The base of an exponent does matter. Compare 2^n and 8^n. If you expand 8^n, you get $(2^3)^n$, which equals 2^{3n}, which equals $2^{2n} * 2^n$. As you can see, 8^n and 2^n are different by a factor of 2^{2n}. That is very much not a constant factor!

The space complexity of this algorithm will be $O(N)$. Although we have $O(2^N)$ nodes in the tree total, only $O(N)$ exist at any given time. Therefore, we would only need to have $O(N)$ memory available.

Examples and Exercises

Big O time is a difficult concept at first. However, once it "clicks," it gets fairly easy. The same patterns come up again and again, and the rest you can derive.

We'll start off easy and get progressively more difficult.

Example 1

What is the runtime of the below code?

```
void foo(int[] array) {
  int sum = 0;
  int product = 1;
  for (int i = 0; i < array.length; i++) {
    sum += array[i];
  }
  for (int i = 0; i < array.length; i++) {
    product *= array[i];
  }
  System.out.println(sum + “, ” + product);
}
```

This will take O(N) time. The fact that we iterate through the array twice doesn't matter.

Example 2

What is the runtime of the below code?

```
void printPairs(int[] array) {
  for (int i = 0; i < array.length; i++) {
    for (int j = 0; j < array.length; j++) {
      System.out.println(array[i] + “,” + array[j]);
    }
  }
}
```

The inner for loop has O(N) iterations and it is called N times. Therefore, the runtime is $O(N^2)$.

Another way we can see this is by inspecting what the "meaning" of the code is. It is printing all pairs (two-element sequences). There are $O(N^2)$ pairs; therefore, the runtime is $O(N^2)$.

Example 3

This is very similar code to the above example, but now the inner for loop starts at `i + 1`.

```
void printUnorderedPairs(int[] array) {
  for (int i = 0; i < array.length; i++) {
    for (int j = i + 1; j < array.length; j++) {
      System.out.println(array[i] + “,” + array[j]);
    }
  }
}
```

We can derive the runtime several ways.

> This pattern of for loop is very common. It's important that you know the runtime and that you deeply understand it. You can't rely on just memorizing common runtimes. Deep comprehension is important.

Counting the Iterations

The first time through `j` runs for N-1 steps. The second time, it's N-2 steps. Then N-3 steps. And so on.

Therefore, the number of steps total is:

```
(N-1) + (N-2) + (N-3) + ... + 2 + 1
```

```
= 1 + 2 + 3 + ... + N-1
= sum of 1 through N-1
```

The sum of 1 through N-1 is $\frac{N(N-1)}{2}$ (see "Sum of Integers 1 through N" on page 630), so the runtime will be O(N^2).

What It Means

Alternatively, we can figure out the runtime by thinking about what the code "means." It iterates through each pair of values for (i, j) where j is bigger than i.

There are N^2 total pairs. Roughly half of those will have $i < j$ and the remaining half will have $i > j$. This code goes through roughly $N^2/2$ pairs so it does O(N^2) work.

Visualizing What It Does

The code iterates through the following (i, j) pairs when N = 8:

```
(0, 1) (0, 2) (0, 3) (0, 4) (0, 5) (0, 6) (0, 7)
       (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7)
              (2, 3) (2, 4) (2, 5) (2, 6) (2, 7)
                     (3, 4) (3, 5) (3, 6) (3, 7)
                            (4, 5) (4, 6) (4, 7)
                                   (5, 6) (5, 7)
                                          (6, 7)
```

This looks like half of an NxN matrix, which has size (roughly) $N^2/2$. Therefore, it takes O(N^2) time.

Average Work

We know that the outer loop runs N times. How much work does the inner loop do? It varies across iterations, but we can think about the average iteration.

What is the average value of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10? The average value will be in the middle, so it will be *roughly* 5. (We could give a more precise answer, of course, but we don't need to for big O.)

What about for 1, 2, 3, ..., N? The average value in this sequence is N/2.

Therefore, since the inner loop does $N/2$ work on average and it is run N times, the total work is $N^2/2$ which is O(N^2).

Example 4

This is similar to the above, but now we have two different arrays.

```
void printUnorderedPairs(int[] arrayA, int[ ] arrayB) {
  for (int i = 0; i < arrayA.length; i++) {
    for (int j = 0; j < arrayB.length; j++) {
      if (arrayA[i] < arrayB[j]) {
        System.out.println(arrayA[i] + "," + arrayB[j]);
      }
    }
  }
}
```

We can break up this analysis. The if-statement within j's for loop is O(1) time since it's just a sequence of constant-time statements.

We now have this:

```
void printUnorderedPairs(int[] arrayA, int[] arrayB) {
  for (int i = 0; i < arrayA.length; i++) {
    for (int j = 0; j < arrayB.length; j++) {
      /* O(1) work */
    }
  }
}
```

For each element of arrayA, the inner for loop goes through b iterations, where b = arrayB.length. If a = arrayA.length, then the runtime is O(ab).

If you said O(N^2), then remember your mistake for the future. It's not O(N^2) because there are two different inputs. Both matter. This is an extremely common mistake.

Example 5

What about this strange bit of code?

```
void printUnorderedPairs(int[] arrayA, int[] arrayB) {
  for (int i = 0; i < arrayA.length; i++) {
    for (int j = 0; j < arrayB.length; j++) {
      for (int k = 0; k < 100000; k++) {
        System.out.println(arrayA[i] + “,” + arrayB[j]);
      }
    }
  }
}
```

Nothing has really changed here. 100,000 units of work is still constant, so the runtime is O(ab).

Example 6

The following code reverses an array. What is its runtime?

```
void reverse(int[] array) {
  for (int i = 0; i < array.length / 2; i++) {
    int other = array.length - i - 1;
    int temp = array[i];
    array[i] = array[other];
    array[other] = temp;
  }
}
```

This algorithm runs in O(N) time. The fact that it only goes through half of the array (in terms of iterations) does not impact the big O time.

Example 7

Which of the following are equivalent to O(N)? Why?

- O(N + P), where $P < N/2$
- O(2N)
- O(N + log N)
- O(N + M)

Let's go through these.

- If $P < N/2$, then we know that N is the dominant term so we can drop the O(P).
- O(2N) is O(N) since we drop constants.

- O(N) dominates O(log N), so we can drop the O(log N).
- There is no established relationship between N and M, so we have to keep both variables in there.

Therefore, all but the last one are equivalent to O(N).

Example 8

Suppose we had an algorithm that took in an array of strings, sorted each string, and then sorted the full array. What would the runtime be?

Many candidates will reason the following: sorting each string is O(N log N) and we have to do this for each string, so that's O(N*N log N). We also have to sort this array, so that's an additional O(N log N) work. Therefore, the total runtime is O(N^2 log N + N log N), which is just O(N^2 log N).

This is completely incorrect. Did you catch the error?

The problem is that we used N in two different ways. In one case, it's the length of the string (which string?). And in another case, it's the length of the array.

In your interviews, you can prevent this error by either not using the variable "N" at all, or by only using it when there is no ambiguity as to what N could represent.

In fact, I wouldn't even use a and b here, or m and n. It's too easy to forget which is which and mix them up. An O(a^2) runtime is completely different from an O(a*b) runtime.

Let's define new terms—and use names that are logical.

- Let s be the length of the longest string.
- Let a be the length of the array.

Now we can work through this in parts:

- Sorting each string is O(s log s).
- We have to do this for every string (and there are a strings), so that's O(a*s log s).
- Now we have to sort all the strings. There are a strings, so you'll may be inclined to say that this takes O(a log a) time. This is what most candidates would say. You should also take into account that you need to compare the strings. Each string comparison takes O(s) time. There are O(a log a) comparisons, therefore this will take O(a*s log a) time.

If you add up these two parts, you get O(a*s(log a + log s)).

This is it. There is no way to reduce it further.

Example 9

The following simple code sums the values of all the nodes in a balanced binary search tree. What is its runtime?

```
int sum(Node node) {
  if (node == null) {
    return 0;
  }
  return sum(node.left) + node.value + sum(node.right);
}
```

Just because it's a binary search tree doesn't mean that there is a log in it!

We can look at this two ways.