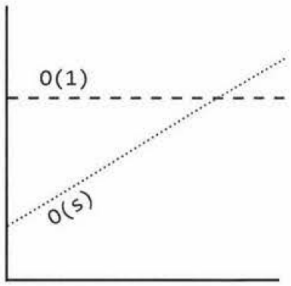
No matter how big the constant is and how slow the linear increase is, linear will at some point surpass constant.



There are many more runtimes than this. Some of the most common ones are $O(\log N)$, $O(N \log N)$, $O(N)$, $O(N^2)$ and $O(2^N)$. There's no fixed list of possible runtimes, though.

You can also have multiple variables in your runtime. For example, the time to paint a fence that's w meters wide and h meters high could be described as $O(wh)$. If you needed p layers of paint, then you could say that the time is $O(whp)$.

▸ 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.

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.

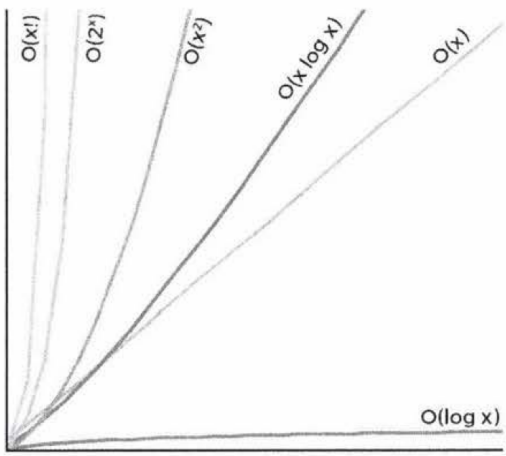
We already said that we drop constants. Therefore, $O(N^2 + N^2)$ would be $O(N^2)$. If we don't care about that latter N^2 term, why would we care about N? We don't.

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 $O(2^x * x!)$.

Add the Runtimes: $O(A + B)$

```
1  for (int a : arrA) {
2    print(a);
3  }
4
5  for (int b : arrB) {
6    print(b);
7  }
```

Multiply the Runtimes: $O(A*B)$

```
1  for (int a : arrA) {
2    for (int b : arrB) {
3      print(a + "," + b);
4    }
5  }
```

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 $X + {}^{X}/_{2} + {}^{X}/_{4} + {}^{X}/_{8} + ... + 1$? This is roughly 2X.

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

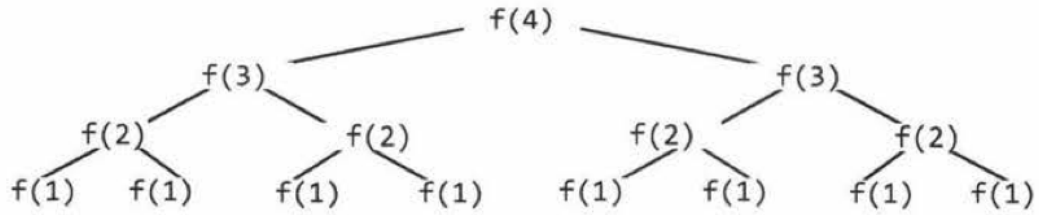
Recursive Runtimes

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

```
1  int f(int n) {
2    if (n <= 1) {
3      return 1;
4    }
5    return f(n - 1) + f(n - 1);
6  }
```

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 $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^N$ (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 $O(branches^{depth})$, 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.

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.