Complexity Analysis

Big O time is the language and metric most commonly used to describe the efficiency of algorithms.

An Analogy

Imagine the following scenario: You have a file on a hard drive, and you need to send it to your friend who lives across the country. You need to get the file to your friend as fast as possible. How should you send it?

Most people's first thought would be email, FTP, or some other means of electronic transfer. That thought is reasonable, but only half correct.

If it's a small file, you're certainly right. It would take 5 – 10 hours to get to an airport, hop on a flight, and then deliver it to your friend.

But what if the file were really, really large? Is it possible that it's faster to physically deliver it via plane?

Yes, actually it is. A one-terabyte (1 TB) file could take more than a day to transfer electronically. A fiber optic network allowing for 75 Mbps bandwidth will take 1 day, 8 hours, 34 minutes, and 41 seconds to transfer a 1TB file across the network.

It would be much faster to just fly it across the country. If your file is that urgent (and cost isn't an issue), you might just want to do that.

What if there were no flights, and instead you had to drive across the country? Even then, for a really huge file, it would be faster to drive.

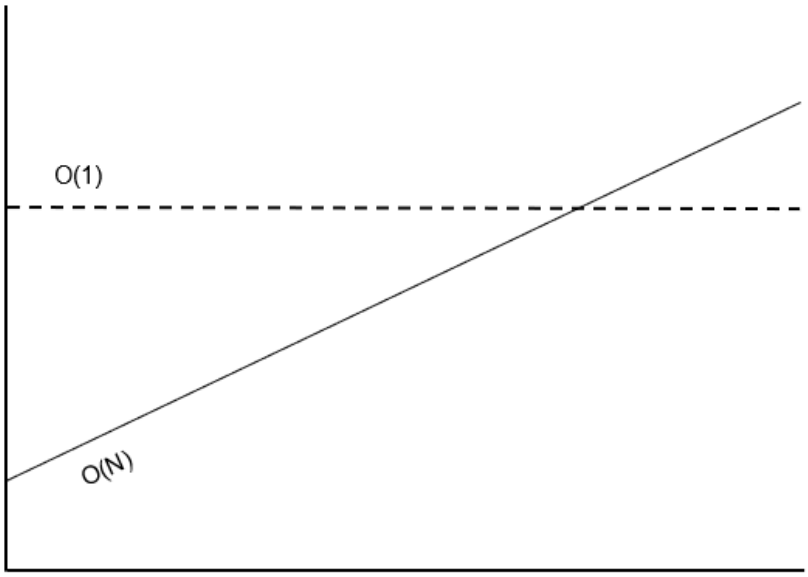
Time Complexity

This is what the concept of asymptotic runtime, or "big O" time, means. We could describe the data transfer "algorithm" runtime as:

- **Electronic Transfer:** O(N), where N is the size of the file. This means that the time to transfer the file increases linearly with the size of the file. (Yes, this is a bit of a simplification, but that's okay for these purposes.)

- **Airplane Transfer:** O(1) with respect to the size of the file. As the size of the file increases, it won't take any longer to get the file to your friend. The time is constant.

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 listed in the table below, in order of increasing runtime. There's no fixed list of possible runtimes though.

Notation	Name
O(1)	Constant
O(log N)	Logarithmic
O(N)	Linear
O(N log N)	Log-linear
$O(N^2)$	Quadratic
$O(N^3)$	Cubic
$O(N^k)$	Polynomial
$O(2^N)$	Exponential
O(N!)	Factorial

You can also have multiple variables in your runtime. For example, the time required to paint a fence that is 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).

Big O, Big Omega, and Big Theta

Academics use big O, big Θ (theta), and big Ω (omega) to describe runtimes.

- **O (big O):** In academia, big O describes an upper bound on the time. An algorithm that prints all the values in an array could be described as O(N), but it could also be described as $O(N^2)$, $O(N^3)$, or $O(2^N)$ (or many other big O times). The algorithm is at least as fast as each of these; therefore, they are upper bounds on the runtime. This is similar to a less-than-or-equal relationship. If Bob is X years old (I'll assume no one lives past age 130), then you could say $X \leq 130$. It would also be correct to say that $X \leq 1{,}000$ or $X \leq 1{,}000{,}000$. It's technically true (although not terribly useful). Likewise, a simple algorithm to print the values in an array is O(N) as well as $O(N^3)$ or any runtime bigger than O(N).

- **Ω (big omega):** In academia, Ω is the equivalent concept but for lower bound. Printing the values in an array is Ω(N) as well as Ω(log N) and Ω(1). After all, you know it won't be *faster* than those runtimes.

- **Θ (big theta):** In academia, Θ means both O and Ω. That is, an algorithm is Θ(N) if it is both O(N) and Ω(N). Θ gives a tight bound on runtime.

In industry (and therefore in job interviews), people seem to have merged Θ and O together. Industry's meaning of O is closer to what academics mean by Θ, in that it would be seen as incorrect to describe printing an array as $O(N^2)$. Industry would just say this is O(N).

Best Case, Worst Case, and Expected (or Average) Case

We can actually describe our runtime for an algorithm in three different ways.

Let's look at this from the perspective of quick sort.

- **Best Case:** If all elements of the array 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 that will run very quickly on a sorted array.)

- **Worst Case:** What if we get really unlucky and the 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 just this situation.) In this case, our recursion doesn't divide the array in half and recursively sort each half, it just shrinks the subarray by one element. We end up with something similar to selection sort and the runtime degenerates to $O(N^2)$.

- **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 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 a O(1) runtime 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.

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 the 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 n x n, 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.

```
1   // Example 1
2   int sum(int n)
3   {
4       if (n <= 0)
5           return 0;
6       else
7           return n + sum(n - 1);
8   }
```

Each call adds a level to the stack.

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

Each of these calls results in a stack frame with a copy of the variable n being pushed onto the program 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 function below, which adds adjacent elements between 0 and n:

```
1   // Example 2
2   int pair_sum_sequence(int n)
3   {
4       int sum = 0;
5       for (int i = 0; i < n; i++)
6       {
7           sum += pair_sum(i, i + 1);
8       }
9       return sum;
10  }
11
12  int pair_sum(int a, int b)
13  {
14      return a + b;
15  }
```

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

Drop the Constants

It is entirely possible for O(N) code to run faster than O(1) code for specific inputs. Big O just describes the *rate of increase*, not the specific time required.

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

Consider the code examples below:

```
1   // Min and Max 1
2   int min = INT_MAX;
3   int max = INT_MIN;
4
5   for (int i = 0; i < n; i++)
6   {
7       if (array[i] < min)
8           min = array[i];
9       if (array[i] > max)
10          max = array[i];
11  }
```

```
1   // Min and Max 2
2   int min = INT_MAX;
3   int max = INT_MIN;
4
5   for (int i = 0; i < n; i++)
6   {
7       if (array[i] < min)
8           min = array[i];
9   }
10  for (int i = 0; i < n; i++)
11  {
12      if (array[i] > max)
13          max = array[i];
14  }
```

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 iteration of its `for` loop rather than one.

If you're going to try 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.

That would be horrendously complicated, so don't even start going down that 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.

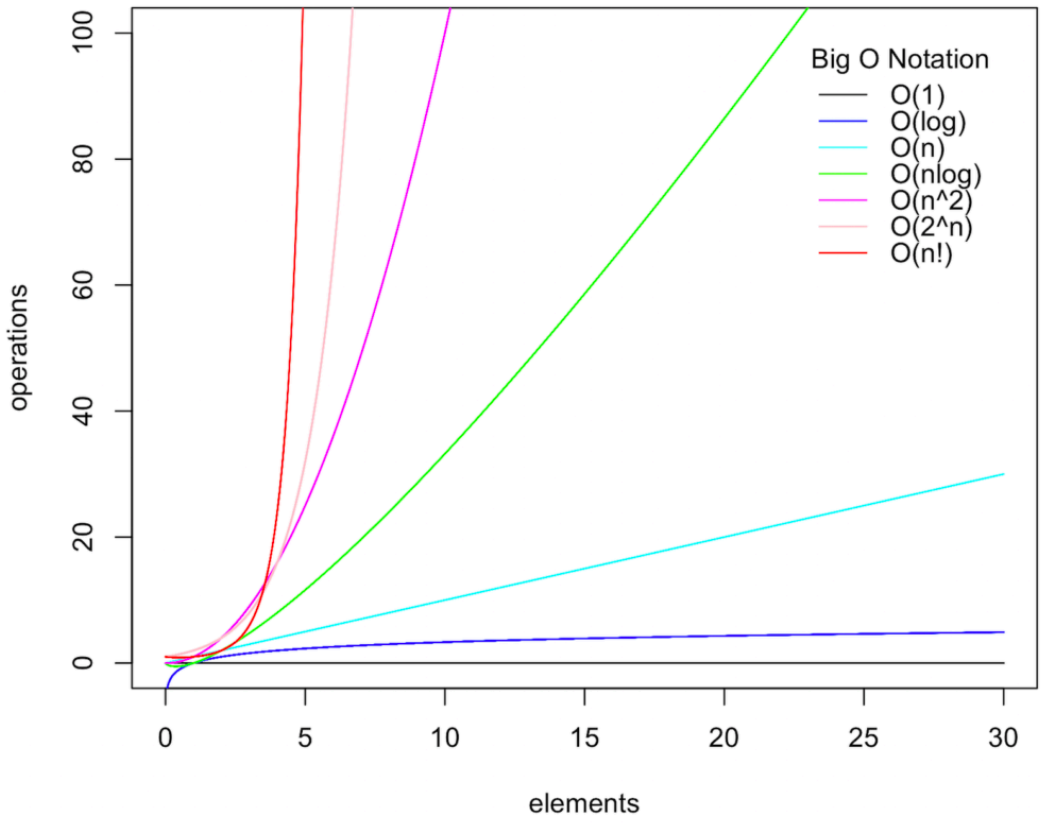
We already said that we drop constants. $O(N^2 + N^2)$ is $O(2N^2)$, and therefore it would be $O(N^2)$. If we don't care about the latter N^2 term, why would we care about N? We don't.

You should drop the non-dominant terms. For example:

- $O(N^2 + N)$ 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, $O(n^2)$ is much worse than O(n), but it's not nearly as bad as $O(2^n)$ or O(n!). There are lots of runtimes worse than O(n!) too, such as $O(n^n)$ or $O(2^n * n!)$.

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?

```
1   // Add the runtimes: O(A + B)
2   for (int i = 0; i < A; i++)
3       cout << arrayA[i];
4
5   for (int i = 0; i < B; i++)
6       cout << arrayB[i];
```

```
1   // Multiply the runtimes: O(A * B)
2   for (int i = 0; i < A; i++)
3       for (int j = 0; j < B; j++)
4           cout << arrayA[i] + ", "
5             + arrayB[j];
```

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.

Amortized Time

A C++ `vector` object allows you to have the benefits of an array while offering flexibility in size. You won’t run out of space in the `vector` since its capacity will grow as you insert elements.

A vector is implemented with a dynamic array. When the number of stored in the array hits the array’s capacity, the vector class will create a new array with double the capacity and copy all of the elements over to the new array. The old array is then deleted.

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 capacity 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 possibilities 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.

(It's 2X - 1 to be exact, but this is big O notation, so we can drop the constant.).

Therefore, X insertions take O(2X) time. The amortized time for each insertion is therefore 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 item `search_key` in an `N` element sorted array. We first compare `search_key` to the midpoint of the

array. If `search_key == array[mid]`, then we return. If `search_key < array[mid]`, then we search on the left side of the array. If `search_key > array[mid]`, then we search on the right side of the array.

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

We start off with with an N-element array to search. Then, after a single step, we're down to N/2 elements. One more step, and we're down to 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 can we multiply N 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 is number of elements in the problem space gets halved each time, that will likely by 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! For binary search it's clearly base 2, but the reality is that it doesn't matter for purposes of big O. Since $\log_a N$ and $\log_b N$ are related by a constant multiplier, and such a multiplier is irrelevant to big O classification, the standard usage for logarithmic-time algorithms is O(log N) regardless of the base of the logarithm.

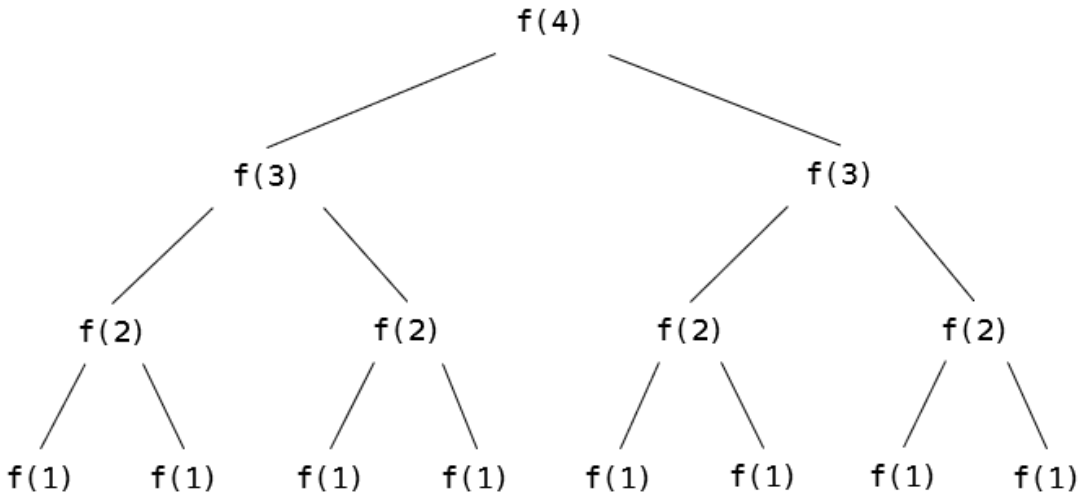
Recursive Runtimes

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

```
1   int f(int n)
2   {
3       if (n <= 1)
4           return 1;
5       else
6           return f(n - 1) + f(n - 1);
7   }
```

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.

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 discussed above, 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 fact of 2^{2N}. That is very much not a constant factor!

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

References

The text of this document is drawn almost entirely verbatim (with minor additions and modifications) from the following book:

McDowell, G. L. (2016). *Cracking the coding interview, 6th edition*. Palo Alto, CA: CareerCup, LLC.