Extra Notes:

- Design for the worst case scenario.
- looked @ different commonly used functions

3.2.1 Comparing Growth Rates

To sum up, Table 3.1 shows, in order, each of the seven common functions used in algorithm analysis.

constant	logarithm	linear	*n*-log-*n*	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Table 3.1: Classes of functions. Here we assume that $a > 1$ is a constant.

Ideally, we would like data structure operations to run in times proportional to the constant or logarithm function, and we would like our algorithms to run in linear or *n*-log-*n* time. Algorithms with quadratic or cubic running times are less practical, and algorithms with exponential running times are infeasible for all but the smallest sized inputs. Plots of the seven functions are shown in Figure 3.4.

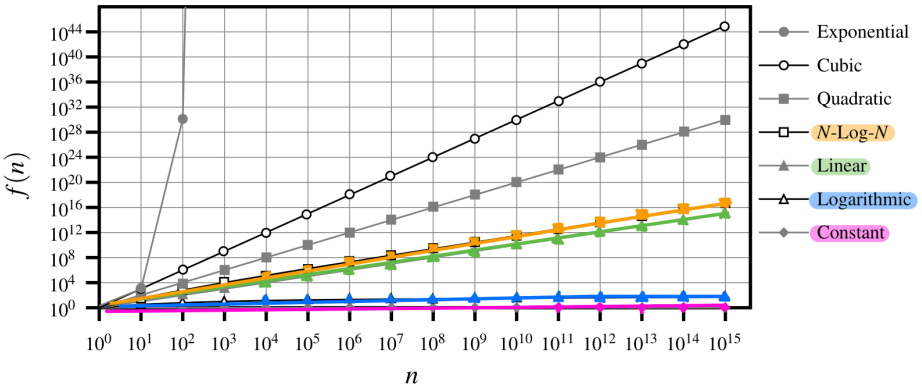
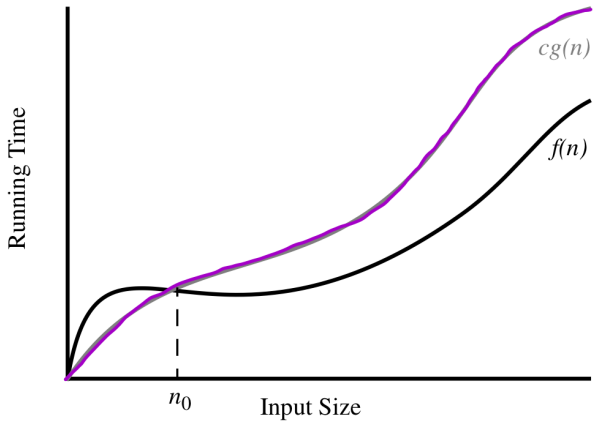


Figure 3.4: Growth rates for the seven fundamental functions used in algorithm analysis. We use base $a = 2$ for the exponential function. The functions are plotted on a log-log chart, to compare the growth rates primarily as slopes. Even so, the exponential function grows too fast to display all its values on the chart.

The "Big-Oh" Notation

$$f(n) \leq cg(n), \quad \text{for} \quad n \geq n_0.$$



f(n)	Name
1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Log Linear
n^2	Quadratic
n^3	Cubic
2^n	Exponential

Table 2.1: Common Functions for Big-O

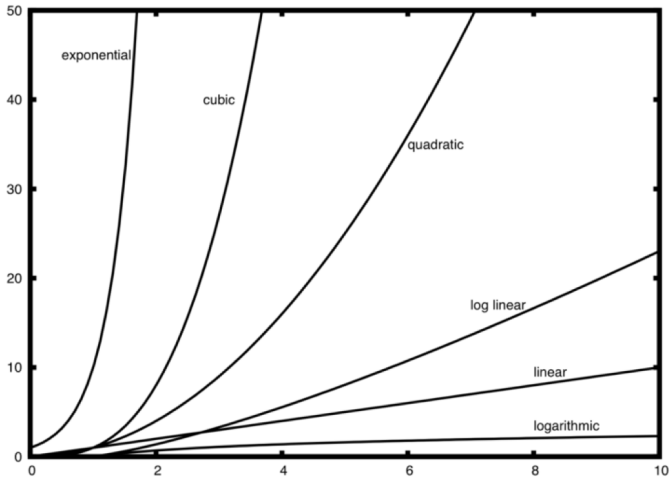


Figure 2.1: Plot of Common Big-O Functions

The simplest definition I can give for Big-O notation is this:

Big-O notation is a relative representation of the complexity of an algorithm.

There are some important and deliberately chosen words in that sentence:

> - **relative:** you can only compare apples to apples. You can't compare an algorithm that do arithmetic multiplication to an algorithm that sorts a list of integers. But a comparison of two algorithms to do arithmetic operations (one multiplication, one addition) will tell you something meaningful;
> - **representation:** Big-O (in its simplest form) reduces the comparison between algorithms to a single variable. That variable is chosen based on observations or assumptions. For example, sorting algorithms are typically compared based on comparison operations (comparing two nodes to determine their relative ordering). This assumes that comparison is expensive. But what if the comparison is cheap but swapping is expensive? It changes the comparison; and
> - **complexity:** if it takes me one second to sort 10,000 elements, how long will it take me to sort one million? Complexity in this instance is a relative measure to something else.

Come back and reread the above when you've read the rest.

The best example of Big-O I can think of is doing arithmetic. Take two numbers (123456 and 789012). The basic arithmetic operations we learned in school were:

> - addition;
> - subtraction;
> - multiplication; and
> - division.

Each of these is an operation or a problem. A method of solving these is called an **algorithm**.

The addition is the simplest. You line the numbers up (to the right) and add the digits in a column writing the last number of that addition in the result. The 'tens' part of that number is carried over to the next column.

Let's assume that the addition of these numbers is the most expensive operation in this algorithm. It stands to reason that to add these two numbers together we have to add together 6 digits (and possibly carry a 7th). If we add two 100 digit numbers together we have to do 100 additions. If we add **two** 10,000 digit numbers we have to do 10,000 additions.

See the pattern? The **complexity** (being the number of operations) is directly proportional to the number of digits *n* in the larger number. We call this **O(n)** or **linear complexity**.

Subtraction is similar (except you may need to borrow instead of carry).

Multiplication is different. You line the numbers up, take the first digit in the bottom number and multiply it in turn against each digit in the top number and so on through each digit. So to multiply our two 6 digit numbers we must do 36 multiplications. We may need to do as many as 10 or 11 column adds to get the end result too.

If we have two 100-digit numbers we need to do 10,000 multiplications and 200 adds. For two one million digit numbers we need to do one trillion (10^{12}) multiplications and two million adds.

As the algorithm scales with n-*squared*, this is **O(n^2)** or **quadratic complexity**. This is a good time to introduce another important concept:

We only care about the most significant portion of complexity.

The astute may have realized that we could express the number of operations as: $n^2 + 2n$. But as you saw from our example with two numbers of a million digits apiece, the second term (2n) becomes insignificant (accounting for 0.0002% of the total operations by that stage).

One can notice that we've assumed the worst case scenario here. While multiplying 6 digit numbers, if one of them has 4 digits and the other one has 6 digits, then we only have 24 multiplications. Still, we calculate the worst case scenario for that 'n', i.e when both are 6 digit numbers. Hence Big-O notation is about the Worst-case scenario of an algorithm.

The Telephone Book

The next best example I can think of is the telephone book, normally called the White Pages or similar but it varies from country to country. But I'm talking about the one that lists people by surname and then initials or first name, possibly address and then telephone numbers.

Now if you were instructing a computer to look up the phone number for "John Smith" in a telephone book that contains 1,000,000 names, what would you do? Ignoring the fact that you could guess how far in the S's started (let's assume you can't), what would you do?

A typical implementation might be to open up to the middle, take the $500,000^{th}$ and compare it to "Smith". If it happens to be "Smith, John", we just got really lucky. Far more likely is that "John Smith" will be before or after that name. If it's after we then divide the last half of the phone book in half and repeat. If it's before then we divide the first half of the phone book in half and repeat. And so on.

This is called a **binary search** and is used every day in programming whether you realize it or not.

So if you want to find a name in a phone book of a million names you can actually find any name by doing this at most 20 times. In comparing search algorithms we decide that this comparison is our 'n'.

> - For a phone book of 3 names it takes 2 comparisons (at most).
> - For 7 it takes at most 3.
> - For 15 it takes 4.
> - …
> - For 1,000,000 it takes 20.

That is staggeringly good, isn't it?

In Big-O terms this is **O(log n)** or **logarithmic complexity**. Now the logarithm in question could be ln (base e), $\log_{10}$, $\log_2$ or some other base. It doesn't matter it's still O(log n) just like $O(2n^2)$ and $O(100n^2)$ are still both $O(n^2)$.

It's worthwhile at this point to explain that Big O can be used to determine three cases with an algorithm:

> - **Best Case:** In the telephone book search, the best case is that we find the name in one comparison. This is **O(1)** or **constant complexity**;
> - **Expected Case:** As discussed above this is O(log n); and
> - **Worst Case:** This is also O(log n).

Normally we don't care about the best case. We're interested in the expected and worst case. Sometimes one or the other of these will be more important.

Back to the telephone book.

What if you have a phone number and want to find a name? The police have a reverse phone book but such look-ups are denied to the general public. Or are they? Technically you can reverse look-up a number in an ordinary phone book. How?

You start at the first name and compare the number. If it's a match, great, if not, you move on to the next. You have to do it this way because the phone book is **unordered** (by phone number anyway).

So to find a name given the phone number (reverse lookup):

> - **Best Case:** O(1);
> - **Expected Case:** O(n) (for 500,000); and
> - **Worst Case:** O(n) (for 1,000,000).

The Traveling Salesman

This is quite a famous problem in computer science and deserves a mention. In this problem, you have N towns. Each of those towns is linked to 1 or more other towns by a road of a certain distance. The Traveling Salesman problem is to find the shortest tour that visits every town.

Sounds simple? Think again.

If you have 3 towns A, B, and C with roads between all pairs then you could go:

> - A → B → C
> - A → C → B
> - B → C → A
> - B → A → C
> - C → A → B
> - C → B → A

Well, actually there's less than that because some of these are equivalent (A → B → C and C → B → A are equivalent, for example, because they use the same roads, just in reverse).

In actuality, there are 3 possibilities.

> - Take this to 4 towns and you have (iirc) 12 possibilities.
> - With 5 it's 60.
> - 6 becomes 360.

This is a function of a mathematical operation called a **factorial**. Basically:

> - 5! = 5 × 4 × 3 × 2 × 1 = 120
> - 6! = 6 × 5 × 4 × 3 × 2 × 1 = 720
> - 7! = 7 × 6 × 5 × 4 × 3 × 2 × 1 = 5040
> - …
> - 25! = 25 × 24 × … × 2 × 1 = 15,511,210,043,330,985,984,000,000
> - …
> - 50! = 50 × 49 × … × 2 × 1 = $3.04140932 \times 10^{64}$

So the Big-O of the Traveling Salesman problem is **O(n!)** or **factorial or combinatorial complexity**.

By the time you get to 200 towns there isn't enough time left in the universe to solve the problem with traditional computers.

Something to think about.

Polynomial Time

Another point I wanted to make a quick mention of is that any algorithm that has a complexity of **$O(n^a)$** is said to have **polynomial complexity** or is solvable in **polynomial time**.

$O(n)$, $O(n^2)$ etc. are all polynomial time. Some problems cannot be solved in polynomial time. Certain things are used in the world because of this. Public Key Cryptography is a prime example. It is computationally hard to find two prime factors of a very large number. If it wasn't, we couldn't use the public key systems we use.

Anyway, that's it for my (hopefully plain English) explanation of Big O (revised).

Time Complexity and Big O Notation

- Big O classify how fast something is ran...

4

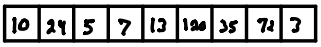
n = length of the data

O(n)

measuring complexity of algorithm

~ length of the data

O(n) Example



n = 9

~ To find "3" it takes 9 operations to find it

O(n)