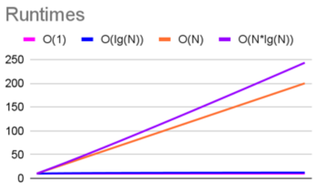
A Bit More About Big O

You just learned that one way to think about the efficiency of algorithms is Big-O. Before we delve into sets in Python, we are going to briefly talk about Big-O a bit more. We will note that if you take a class on the theory of algorithms, you will find that a big focus of such a class is use of a formal mathematical definition of Big-O and proving that various algorithms have certain Big-O runtimes. We are not going to go into the formal math, but want to mention that it exists.

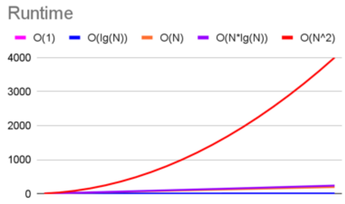
Instead, we want to give you an idea of what common Big-Os are for algorithms, and how "good" they are. We first start with what are considered tractable runtimes—any algorithm in these categories is generally considered reasonable for any size input you would typically want to use.

- **O(1) constant time:** This runtime is the best you can do. The runtime does not change as input size changes. Whether you have 5 elements or 50 million elements, the algorithm takes the same amount of time.
- **O(lg(N)) logarithmic time:** This runtime is really good. Every time the input size doubles, the runtime increases by a constant amount. For example, the algorithm might take 2 seconds to run on 1 million elements, 2.1 seconds on 2 million elements, 2.2 seconds on 4 million elements, 2.3 seconds on 8 million elements, and so on. Algorithms in this category generally work by eliminating half of the possibilities at each step.
- **O(N) linear time:** This runtime is still pretty good, but for very large data is likely to take noticeable amounts of time. These algorithms are quite common. In general when you are iterating "for each element of (data set)" and doing some "simple" operations inside the loop, you have an O(N) algorithm.
- **O(N*lg(N)) linerathmic time:** This runtime often comes up when a logarithmic time algorithm has to be applied to each element of the input set. Algorithms in this class are still generally quite efficient. This class of runtimes is where the best possible sorting algorithms reside.
- **O(N^2), O(N^3), ...other polynomial time algorithms:** Algorithms in other polynomial categories are still generally considered tractable for most problem sizes. However, as the exponent on the N increases, the runtime may become quite long for large input sizes.

This graph briefly summarizes the general shapes of the runtimes up to :



Note that O(1) and O(lg(N)) are both plotted, it is just hard to see them both because O(lg(N)) grows so slowly compared to O(N). In fact, we didn't put O(N2) on the graph because when we do, you cannot really see the difference in the other functions:



Note that this second graph has all the series of data that the first does, they are just so small compared to the new O(N^2) line, that they only appear at the very bottom of the graph. If we continued this trend and added O(N^3), it would make O(N^2) appear tiny.

However, even for moderate degree polynomials, we can still generally compute things in some feasible (even if not pleasant) amount of time, especially if we can throw significant hardware resources and parallelism at the problem. There are, however, runtimes that are intractable. If your program has such a runtime, then even modestly sized data becomes impossible.

The two major classes of intractable runtimes are:

- **O(2^n) exponential time:** Adding one to the size of the inputs doubles the runtime. So if 10 elements take 1 second, then 11 elements take 2 seconds, and 12 elements takes 4 seconds. This may not sound so bad, until you consider the fact that for N=60, 2^N is one quintillion (one billion billion). As one quintillion is one billion billion, we can note that an O(N^2) algorithm with the same constant factors operating on a one billion element input set would take the same time as an O(2^n) algorithm operating on a 60 element input set.
- **O(N!) factorial time:** This runtime comes up when you try all possible combinations for N elements, and is even worse than exponential time.

To underscore just how impossible exponential and factorial runtimes are, we show you roughly how long it would take for a few input sizes on three different hardware configurations. We note that the exact times/numbers are all dependent on a lot of things (such as the constant factors, how well the algorithm can be split across multiple computers etc). However, this shows the general trend, and underscores how bad these runtimes are. We consider three hardware configurations. First, a "typical" desktop computer: what you might buy at the store and at home or school. Second, running it on the world's fastest supercomputer. Third, using the top 500 supercomputers in the world, put together to solve your problem. In this table, we have colored everything less than a month in green, and everything more than 1,000 years in red.

Problem size for...		Time on...		
O(2^n)	O(N!)	A typical desktop computer	The world's top supercomputer	The combination of the top 500 supercomputers
50	20	1 week	0.01 seconds	1 millisecond
60	22	20 years	10 seconds	1 second
70	24	20,000 years	3 hours	20 minutes
80	26	20 million years	20 weeks	2 weeks
90	28	20 billion years	400 years	40 years
100	30	20 trillion years	400,000 years	40,000 years

We don't want you to focus on the specific numbers here (these are all rough approximations for a hypothetical problem anyways). What we want you to see here is that for an O(2^N) problem inputs of size ~60 require massive hardware and parallelism, and for inputs of size ~100, the problem becomes so computationally intensive that you just cannot solve it no matter what your hardware budget is: you would need hundreds of thousands of copies of the world's fastest supercomputer working together perfectly to solve it under a year. The same general trends hold for O(N!) algorithms with inputs of size ~22 requiring massive hardware, and those of size ~30 becoming impossible.

Accordingly, if you ever are contemplating an algorithm that has these sorts of runtimes, be aware that you just can't use it for data of non-trivial size.

Day 8, Part 1

Now it is time to do **16_closest_point** on the Mastery Learning Platform. Login to the course server, go to **16_closest_point** in your git repository, and read the README for directions. When you have passed that assignment, return to Sakai and continue with the content here.