Big-O notation explained by a self-taught programmer

Big-O notation used to be a really scary concept for me. I thought this is how "real" programmers talked about their code. It was all the more scary because the academic descriptions (such as Wikipedia) made very little sense to me. This is frustrating because the underlying concepts aren't actually that hard.

Simply put, Big-O notation is how programmers talk about algorithms. Algorithms are another scary topic which I'll cover in another post, but for our purposes, let's say that "algorithm" means a function in your program (which isn't too far off). A function's Big-O notation is determined by how it responds to different inputs. How much slower is it if we give it a list of 1000 things to work on instead of a list of 1 thing?

Consider this code:

```
def item_in_list(to_check, the_list):
    for item in the_list:
        if to_check == item:
          return True
    return False
```

So if we call this function like `item_in_list(2, [1,2,3])`, it should be pretty quick. We loop over each thing in the list and if we find the first argument to our function, return True. If we get to the end and we didn't find it, return False.

The "complexity" of this function is `O(n)`. I'll explain what this means in just a second, but let's break down this mathematical syntax. `O(n)` is read "Order of N" because the `O` function is also known as the Order function. I think this is because we're doing approximation, which deals in "orders of magnitude".

"Orders of magnitude" is **YET ANOTHER** mathematical term which basically tells the difference between classes of numbers. Think the difference between 10 and 100. If you picture 10 of your closest friends and 100 people, that's a really big difference. Similarly, the difference between 1,000 and 10,000 is pretty big (in fact, its the difference between a junker car and a lightly used one). It turns out that in approximation, as long as you're within an order of magnitude, you're pretty close. If you were to guess the number of gumballs in a machine, you'd be within an order of magnitude if you said 200 gumballs. 10,000 gumballs would be *WAY* off.

Figure 1: A gumball machine whose number of gumballs is probably within an order of magnitude of 200.

Back to dissecting `O(n)`, this says that if we were to graph the time it takes to run this function with different sized inputs (e.g. an array of 1 item, 2 items, 3 items, etc), we'd see that it approximately corresponds to the number of items in the array. This is called a linear graph. This means that the line is basically straight if you were to graph it.

Some of you may have caught that if, in the code sample above, our item was always the first item in the list, our code would be really fast! This is true, but

Big-O is all about the approximate worst-case performance of doing something. The worst case for the code above is that the thing we're searching for isn't in the list at all. (Note: The math term for this is "upper bound", which means its talking about the mathematic limit of awfulness).

If you wanted to see a graph for these functions, you ignore the `O()` function and change the variable `n` for `x`. You can then type that into Wolfram Alpha as "plot x" which will show you a diagonal line. The reason you swap out n for x is that their graphing program wants `x` as its variable name because it corresponds to the x axis. The x-axis getting bigger from left to right corresponds to giving bigger and bigger arrays to your function. The y-axis represents time, so the higher the line, the slower it is.

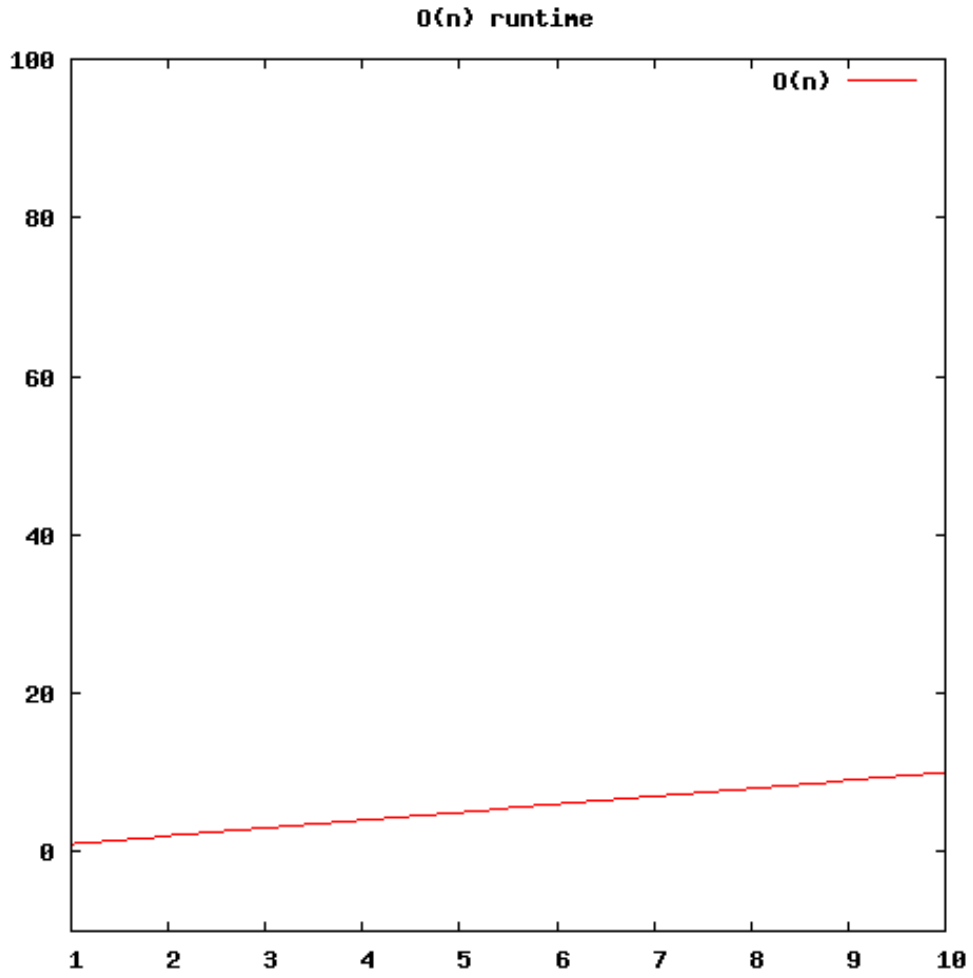

Figure 2: Runtime characteristics of an O(n) function

So what are some other examples of this?

Consider this function:

```
def is_none(item):
    return item is None
```

This is a bit of a silly example, but bear with me. This function is called `O(1)` which is called "constant time". What this means is no matter how big our input is, it always takes the same amount of time to compute things. If you go back to Wolfram and `plot 1`, you'll see that it always stays the same, no matter how far right we go. If you pass in a list of 1 million integers, it'll take about the same time as if you were going to pass in a list of 1 integer. Constant time is considered the best case scenario for a function.

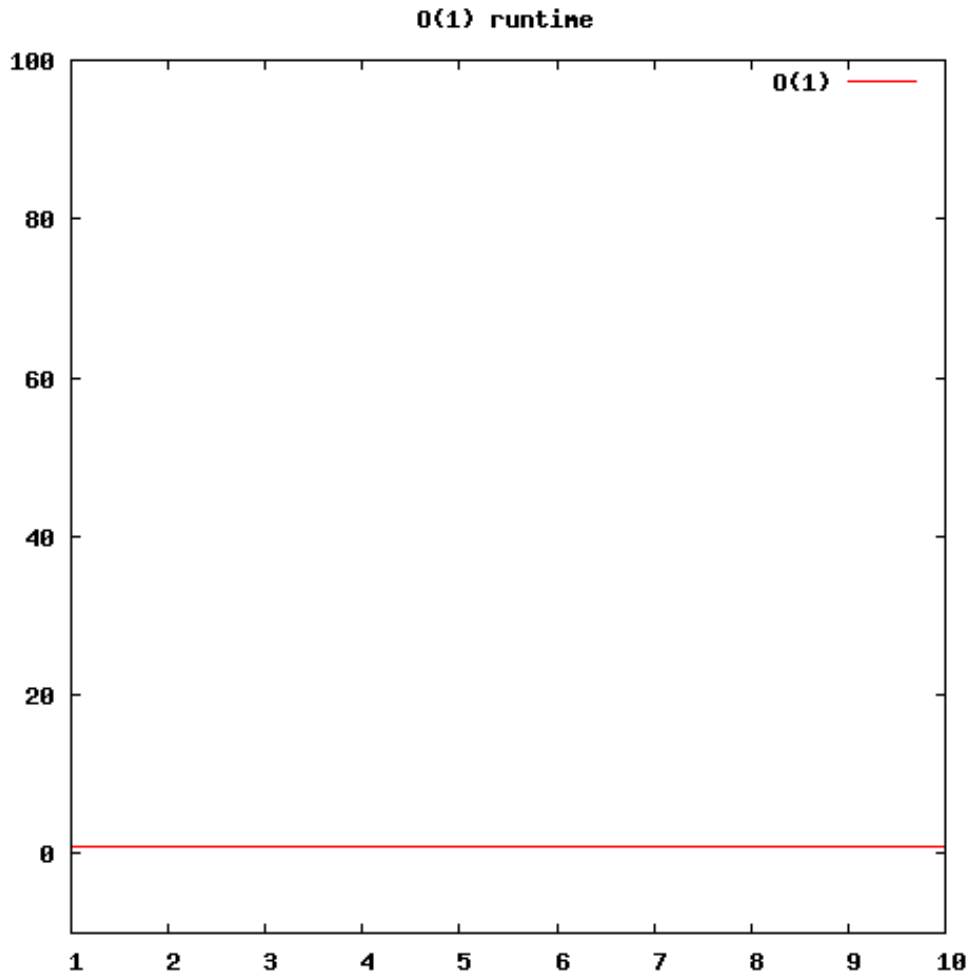

Figure 3: Runtime characteristics of an O(1) function

Consider this function:

```
def all_combinations(the_list):
    results = []
    for item in the_list:
        for inner_item in the_list:
            results.append((item, inner_item))
    return results
```

This matches every item in the list with every other item in the list. If we gave it an array `[1,2,3]`, we'd get back `[(1,1) (1,2), (1,3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]`. This is part of the field of combinatorics (*warning: scary math terms!*), which is the mathematical field which studies combinations of things. This function (or algorithm, if you want to sound fancy) is considered `O(n^2)`. This is because for every item in the list (aka `n` for the input size), we have to do `n` more operations. So `n * n == n^2`.

Below is a comparison of each of these graphs, for reference. You can see that an `O(n^2)` function will get slow very quickly where as something that operates in constant time will be much better. This is particularly useful when it comes to data structures, which I'll post about soon.

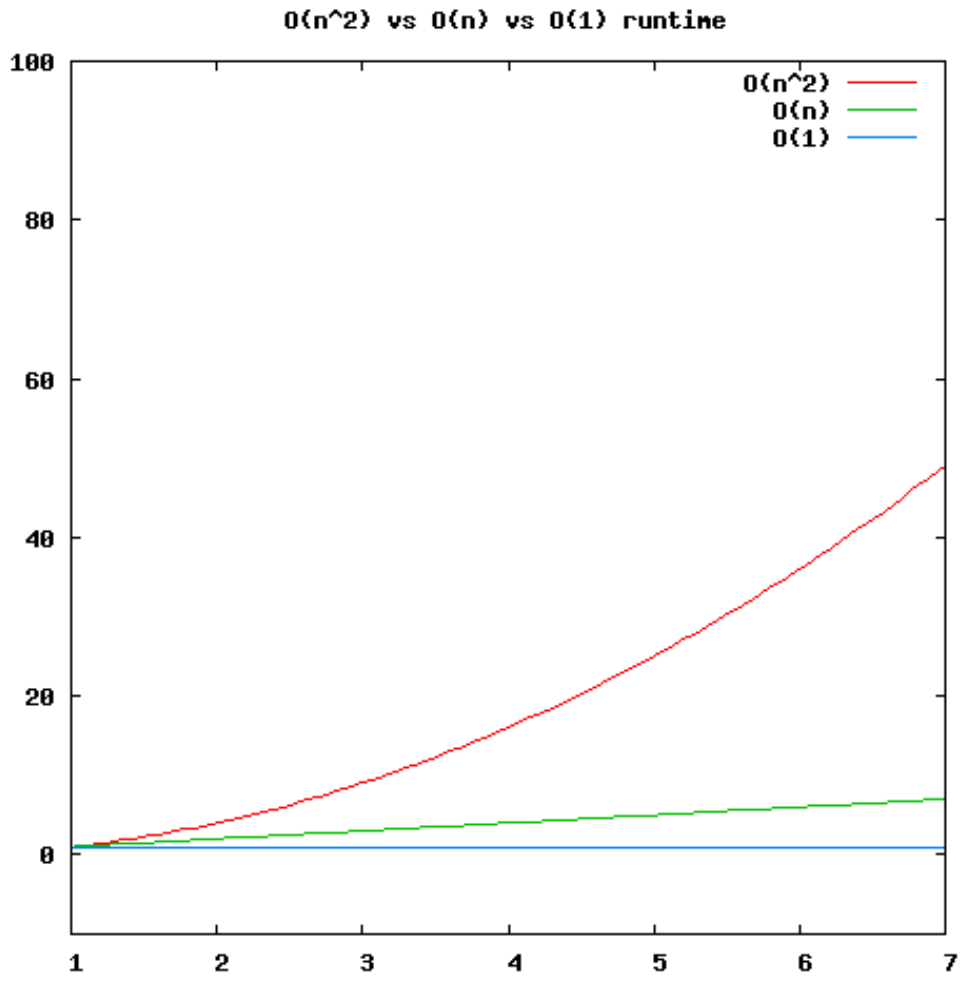

Figure 4: Comparison of $O(n^2)$ vs $O(n)$ vs $O(1)$ functions

This is a pretty high level overview of Big-O notation, but hopefully gets you acquainted with the topic. There's a coursera course which can give you a more in depth view into this topic, but be warned that it will hop into mathematic notation very quickly. If anything here doesn't make sense, send me an email.

Update: I've also written about how to calculate Big-O.

I'm thinking of writing a book on this topic. If this is something you'd like to see, please express your interest in it here.

Get notified when new articles are published

email address

Subscribe