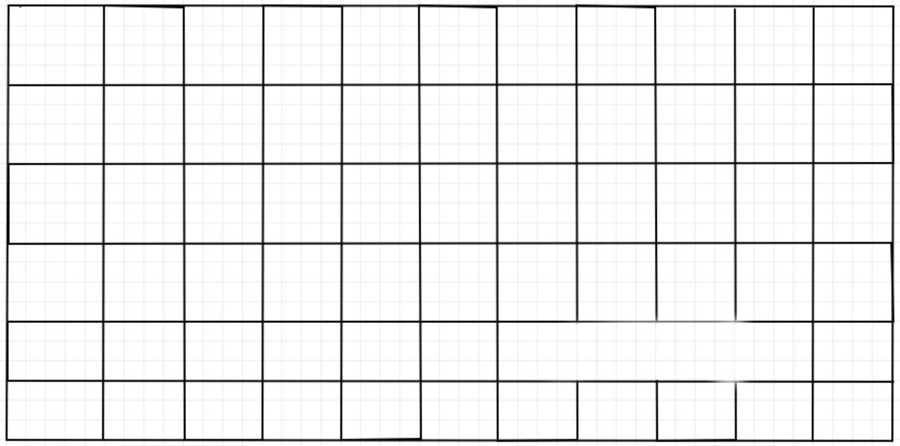
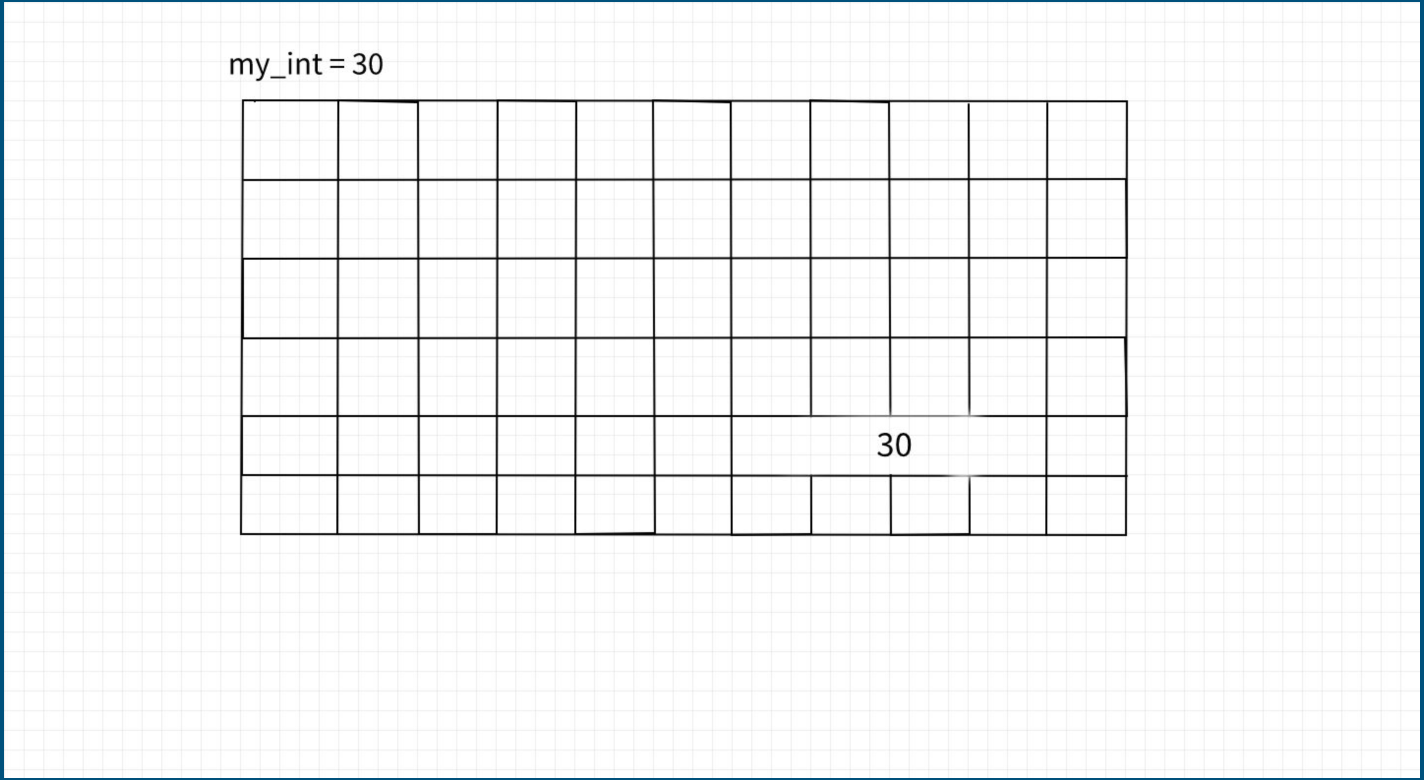
Big O Notation

Coding Temple 2021

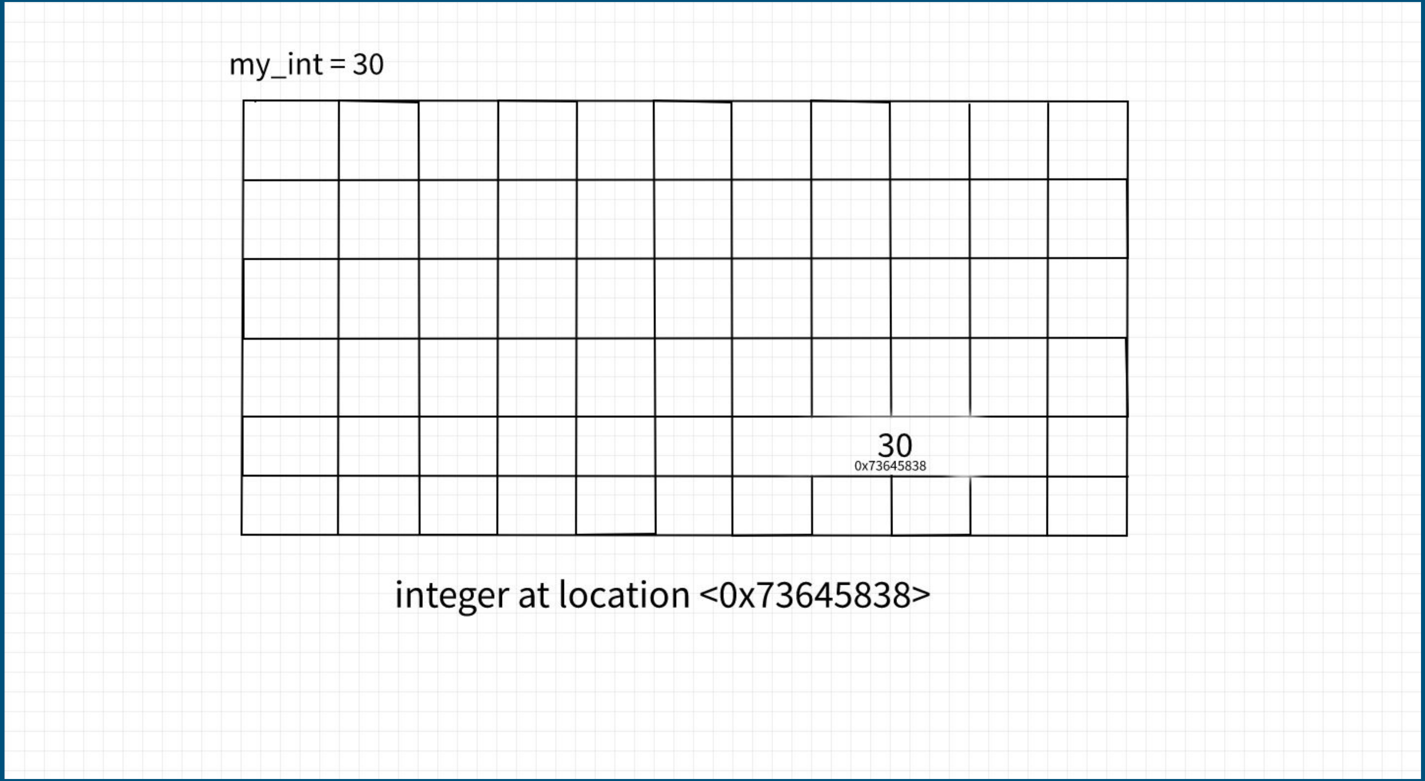
A little bit about memory...



A little bit about memory...



A little bit about memory...



Big O Notation

Big O is the way computer engineers (and super smart math nerds who do this stuff without computers) notate how much time and memory a function will take. Generally, speed is prioritized, so finding the fastest way of solving a problem is what big O is oriented around. Usually, a problem will have multiple ways of approaching and solving it; optimizing the solution is an important part of writing algorithms.

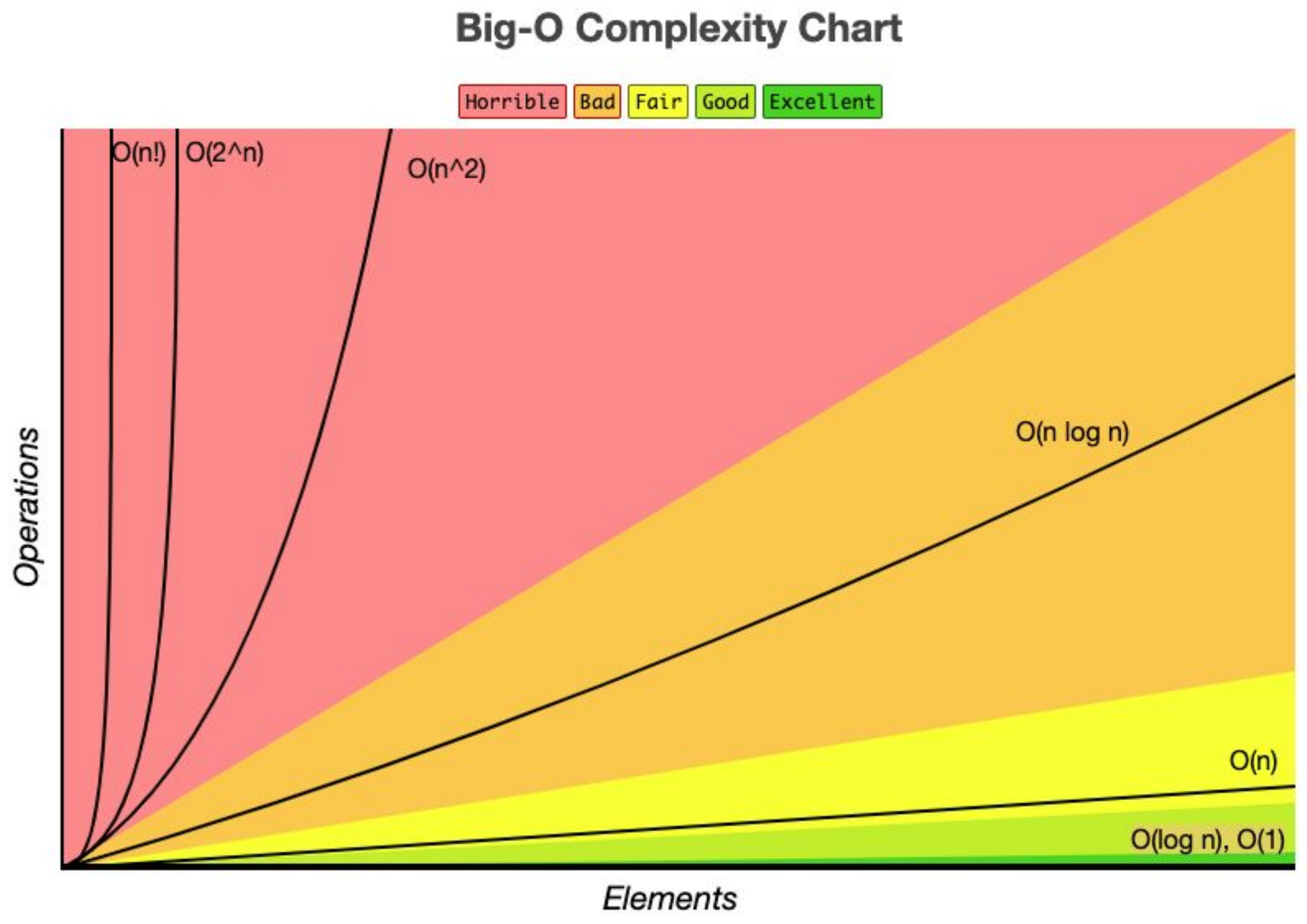
Computer brainz

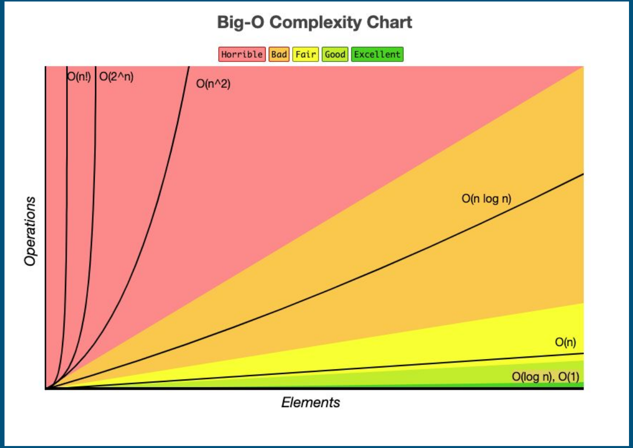
An example: When we give a computer a list, it can’t look at multiple items in the list at once. So it holds them in memory while it works. But it works on one item at a time, and telling it how to do that work is hugely important. **This is the focus of algorithmic choices.**

The phone book

Imagine looking for a specific name in a phone book. There are a few ways to get there. One would be starting on page one and working through each item on the page until you find the letter. Is this efficient?

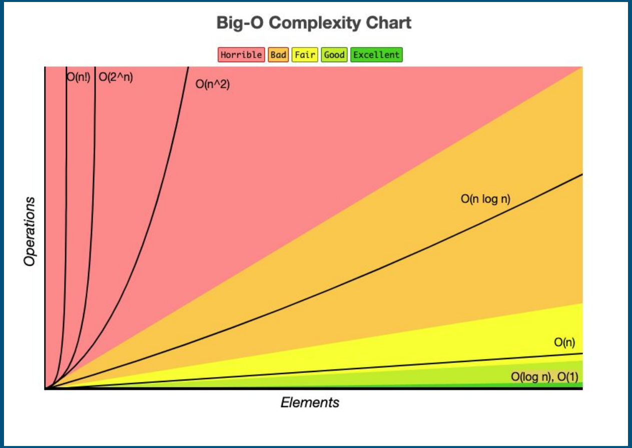
What if instead, we opened the book to the middle and asked if the name we were looking for came before or after. We then ripped the phone book in half because we are so strong and capable, chucked the half we didn’t need away, and then flipped to the middle of the half that we kept. Then we did this again. Is this efficient?


Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements



Credit to eric at
https://www.bigocheatsheet.com/

Look at this graph



Why would some of these be better than others?

If a computer can only really do one thing at a time, how would that affect what we choose to do with our chart?

Some problems cannot be solved with the green or yellow or sometimes even orange lines.

Library books

In the phone book example, it was a little harder to see where the ‘memory’ heavy side of things came in to play. Let’s try another.

Imagine you are sorting library books. You have 1000 of them and you need to alphabetize them and put them away. One way would be to take each book and put it on the shelf, and if you need to, insert the book ahead of what you’ve already placed into the correct location. OR

You could make piles of books that start with A, B, C etc and then sort those independently and put those on the shelf, which requires less running back and forth from your book pile to the shelf.

Memory

In both examples, we end up holding bigger pieces of information in memory. In the phone book, we compare and hold the entire half of the phonebook instead of just one name at a time. In the library case, we hold piles of books based on their letter and wait with them until it’s time to use them. In both cases, how we use the memory affects how we use the time.

Sometimes memory gets ignored for time, but super fast functions take a lot of electricity as well.

Rules of Big O

- Drop constants! O(2N) is still just O(N)
- ANY kind of looping function (even if it doesn't LOOK like one) is a loop! Think of things like map, while, even some things like sorted() is a secret loop!
- Think mathematically… if you have a loop inside a loop, it is like n * n which results in an exponent - n^2!
- Think mathematically… if you have two loops together, it like n and then another n which means it's not an exponent! Don't just count for loops!
- Imagine the average performance of your algorithm. If we used 10,000 different data inputs (if that's part of the function) imagine what the average performance would be.

Why do we just ignore some numbers???????? It’s not precise to say ‘I don’t care if we add numbers at the end it’s still just O(n^2) I don’t like this Brandon why did the nerds do this to us’

Imagine a mathematical equation with a huuuuuuge exponent.

For example:

$5n^3 + 7n^2 + 5$

Now if n = 1, this is only 5 + 7 + 5, which is 17. The 5 at the end is important now.

But.

If n = 20, this is 40,000 + 2,800 + 5. The 5 seems much less important in the light of 42,805.

If n = 100, we have 5,000,000 + 70,000 + 5… and the 70,000 is also prrrrretty useless in the grand scheme. So we don’t really care about that either.

Imagine a mathematical equation with a huuuuuuge exponent.

For example:

$5n^3 + 7n^2 + 5$

If n = 100, we have 5,000,000 + 70,000 + 5… and the 70,000 is also prrrrretty useless in the grand scheme. So we don’t really care about that either.

But if we didn’t have the cubed n at the beginning, that 70,000 is important compared to the 5. So we then care about the n^2. Also, we are more concerned with the scaling - 10,000 (if you don’t multiply n^2 with 7) went up a lot faster than 5 because 5 doesn’t change at all. It’s a **constant**.

Scaling

Last little bit from here… The scaling part applies when you don't know the length of your list. If I said:

```
def someFunc():
    a_list = [1]
    for i in a_list:
        print(i)
```

Our function will not change no matter what data happens around it. It'll always loop just once because there's only one item in the list, so it'd be O(1) because 1 is a constant. BUT

Scaling

```
def someFunc(a_list):
    for i in a_list:
        print(i)
```

This would not be constant; it would be linear. If the length of our list = 100, it will take longer than if our list is only 5 items. That's why our choices for how to solve our problems are important - we don't always know what our algorithms will be fed, so we want to pick algorithms that respond well to whatever gets thrown at them. Because our algorithm changes on that input, we'd just have O(n) where n is the length of the input.

Lab time

Do a laboratory activity, thank you for coming to my ted talk