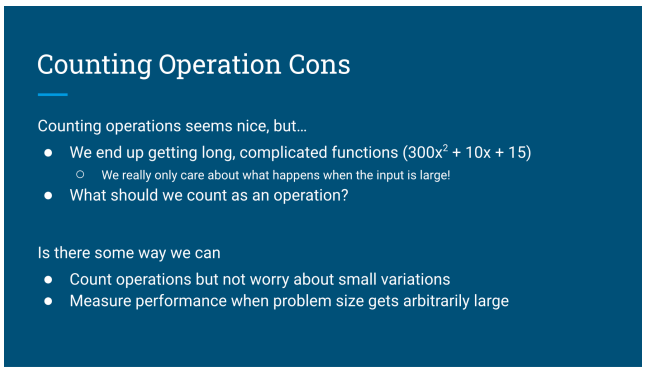
Big O

Date: Oct 28, 2024

Big O Notation



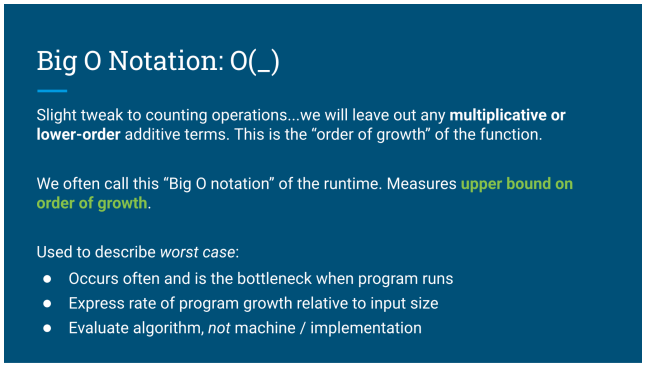
What we're going to focus on is that idea of counting operations, but we're not going to worry about small variations, whether it's three or four steps inside of the loop. We're going to show that that doesn't matter.

We're focused on what happens when the size of the problem gets arbitrarily large. We don't care about counting things from 0 up to x when x is 10 or 20. Instead, what happens when it's a million or a billion?

And we want to relate that time needed against the size of the input, so we can make that comparison.

So to do that, we have to do a couple of things.

We have to decide what we're going to measure, and then we have to think how we count without worrying about implementation details.



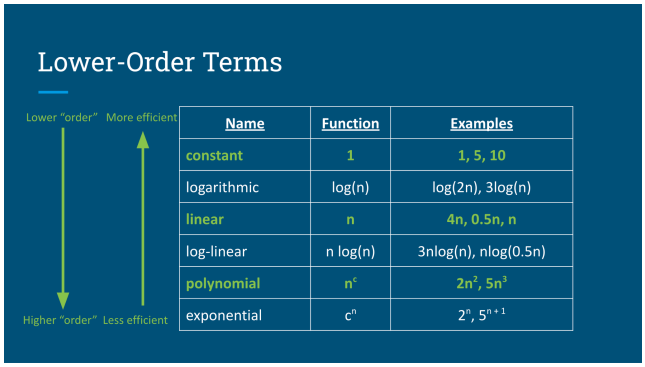
Focusing on the order of growth!
This means you're focusing on the dominant term when analyzing the time complexity of an algorithm.

Big O Notation:

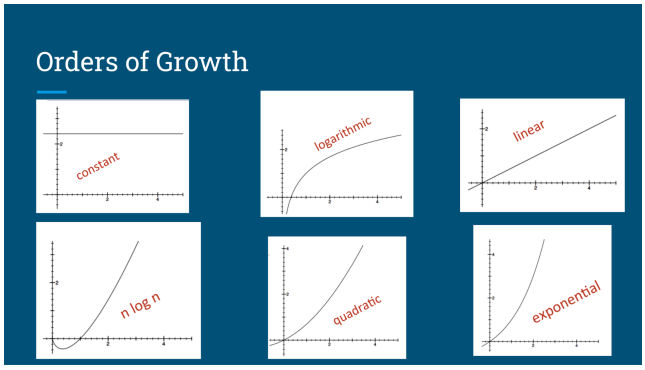
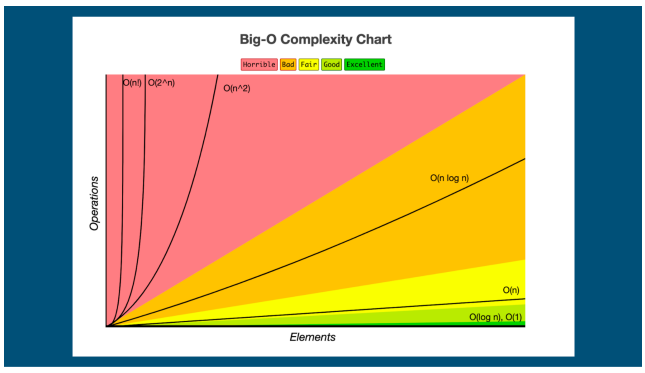
- Big O notation describes the upper bound of an algorithm's runtime as the input size grows.

	• It helps us understand how the algorithm scales. • We care about the long-term trends. • Simplify for analysis purposes! **What about multiplicative constants?** • Constants don't significantly affect how the runtime scales. • Whether an algorithm takes 5n or 100n operations, the growth is still linear. **What about lower-order additive terms?** • As the input size (n) gets very large, lower-order terms like n or log(n) become insignificant compared to the dominant term (e.g., n^2). Let's say you have an algorithm with the following operation count: 5n^2 + 3n + 10 The dominant term is: n^2 (it grows fastest as n increases). We drop the constants and lower-order terms, leaving us with O(n^2). **Worse-case:** Often, the worst-case scenarios happen frequently enough to significantly impact the overall performance of a program. They become the "bottleneck" — the slowest part that limits the speed of the entire process. **Express rate of program growth as a function of input size:** This refers to how the runtime of an algorithm increases as the amount of data it processes (the input size) grows. This is where Big O

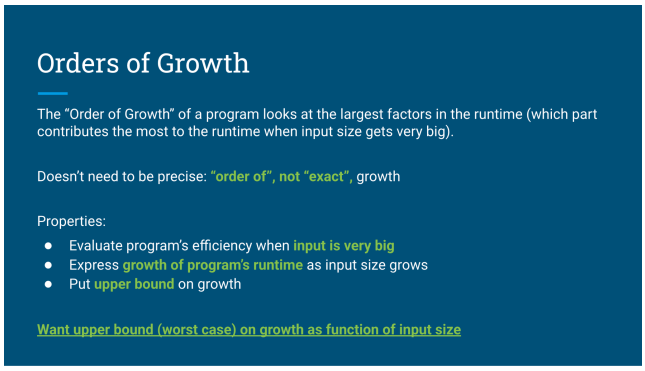
	notation comes in, providing a way to express that growth rate (e.g., O(n), O(n^2), O(log n)). **Evaluate the efficiency of the algorithm:** Different machines and implementations might make the algorithm run faster or slower, but the underlying growth rate remains the same.

	"Lower order" refers to terms in a mathematical expression that grow more slowly than other terms as the input size increases. Growth Rates: Different mathematical functions have different growth rates. For example: Constant: O(1) - Doesn't grow at all. Logarithmic: O(log n) - Grows slowly. Linear: O(n) - Grows proportionally to the input size. Quadratic: O(n^2) - Grows much faster as the input size increases. Focus on Scalability: When analyzing algorithms, we primarily care about how the runtime scales with larger inputs. Lower order terms have less impact on this scalability compared to the dominant term. Simplification: Big O notation often ignores lower order terms and constants to provide a clearer picture of the algorithm's overall growth rate. This simplification makes it easier to compare algorithms and understand their efficiency. Polynomial n (linear) n^2 (quadratic)

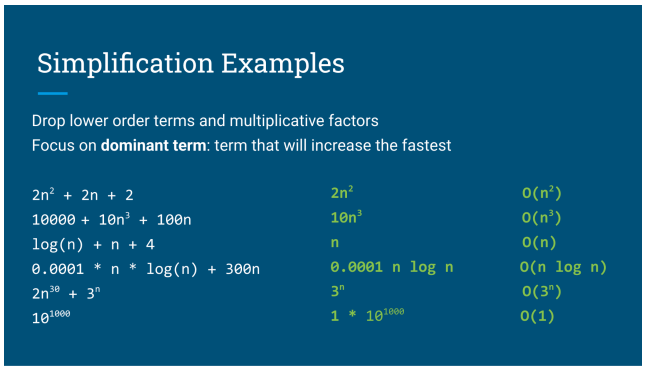
	n^3 (cubic) 2n^2 + 5n + 1 Key Characteristic: The growth rate is determined by the highest power of n. As n increases, the term with the highest power dominates the overall value of the function. Exponential The variable n appears in the exponent. This causes the function value to increase dramatically as n grows.

 	Let's take a look at different orders of growth. The x axis on these charts is the size of input and the y axis is the amount of time taken for that input size. First here we have constant growth, which means that the time needed for the program to run is constant and the amount of time doesn't change as the size of the input gets changed. Next here we have linear growth, which grows in a straight line. Quadratic starts to grow more quickly. Here we also have logarithmic, which is always better than linear because it slows down as we increase the size. Then n log n or log linear is kind of interesting, but it's a very common complexity for really valuable algorithms in computer science, notably ones for sorting. And it has a nice behavior, sort of between the linear and the quadratic.

	And lastly here we have exponential, which very quickly escalates in time as our inputs get bigger.

	We want to evaluate efficiency, particularly when the input is very large. What happens when we really scale this up? We want to express the growth of the program's runtime as that input grows. Not the exact runtime, but that notion of it. If we doubled the input, how much longer does it take? What's the relationship between increasing the size of the input and the increase in the amount of time it takes to solve it? We're going to put an upper bound on that growth. An upper bound means that it's at least as big as or bigger than the actual amount of time it's going to take. And we're going to not worry about being precise. We're going to talk about the order of rather than the exact growth. We don't need to know to the femtosecond how long this is going to take, or to the exact number of operations this is going to take. But, we do want to say things like this is going to grow linearly. We double the size of the input, it doubles the amount of time. Or this is going to grow quadratically. We double the size of the input, it's going to take four times as much time to solve it. (2x) ^ 2 = (2^2)(x^2) = 4x^2 Or if we're really lucky, this is going to have constant growth, meaning no matter how the input changes, it's not going to take any more time.

To do that, we're going to look at the largest factors in the runtime. Which piece of the program takes the most time? And so in order of growth, we are going to look at an upper bound on the growth as a function of the size of the input in the worst case. So here's the notation we're going to use. It's called Big O notation.

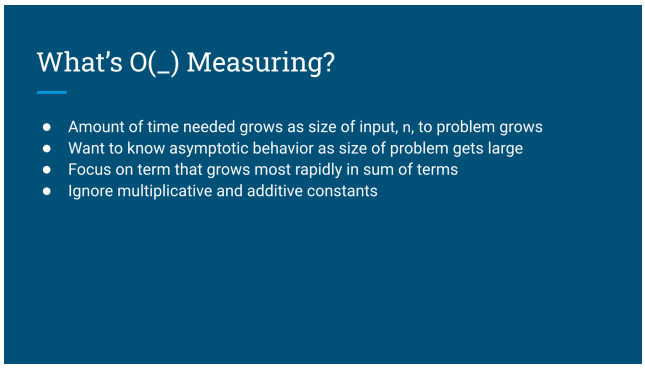


So here are some examples. If we're counting operations and we come up with an expression that has n squared plus 2n plus 2 operations, that expression is order n squared. The 2 and the 2n don't matter. Let's think about what happens if we made n really big. n squared is much more dominant than the other terms. We say that's order n squared.

Even this expression we say is order n squared. So in this case, for lower values of n, this term is going to be the big one in terms of number of steps. I have no idea how I wrote such an inefficient algorithm that it took 100 steps to do something. But if we had that expression for smaller values of n, this matters a lot. This is a really big number. But when we're interested in the growth, then that's the term that dominates. When we have expressions, if it's a polynomial expression, it's the highest order term. It's the term that captures the complexity. Both of these are quadratic. This term is order n, because n grows faster than log of n.

This funky looking term, even though that looks like the big number there and it is a big number, that expression we see is order n log n. Because again, if I plot out as how this changes as I make n really large, this term eventually takes over as

	the dominant term. What about that one? What's the big term there?

	So what does O(n) measure? Well, we're just summarizing here. We want to describe how much time is needed to compute or how does the amount of time, rather, needed to compute problem growth as the size of the problem itself grows. So we want an expression that counts that asymptotic behavior. And we're going to focus as a consequence on the term that grows most rapidly. As you give an algorithm more data to process (a larger input size), it generally takes longer to complete its task. Asymptotic behavior refers to how the runtime of an algorithm changes as the input size (n) approaches infinity. We're interested in the long-term trends, not minor fluctuations for small inputs. What is important is the dominant term in an expression. For example, in 2n^2 + 5n + 10, the n^2 term grows much faster than the others as n increases. This dominant term dictates the overall growth rate of the algorithm. Constants don't significantly affect how the runtime scales with input size. Whether it takes 5n or 100n operations, the growth is still linear O(n). Similarly, additive constants become insignificant as n grows very large.

	In essence, this statement is saying that when analyzing algorithm efficiency, we care about the long-term growth rate as the input size increases. We focus on the dominant term and use Big O notation to express this growth rate in a simplified way, ignoring less important constants and lower-order terms. This approach allows us to: ● Compare algorithms: Easily compare the efficiency of different algorithms regardless of the specific hardware or implementation. ● Predict scalability: Understand how an algorithm's performance will change as the amount of data increases. ● Make informed decisions: Choose the most suitable algorithm for a given task, especially for large datasets.

Storing Data in Computer Memory (Python)

Strings

- Immutable: Strings in Python are immutable. This means that once a string is created, its contents cannot be changed. Any operation that appears to modify a string actually creates a new string in memory.
- Interning: Python uses a technique called "string interning" to optimize memory usage. When you create a string, Python checks if an identical string already exists in memory. If it does, it reuses that existing string object instead of creating a new one. This is particularly common for short strings and identifiers.
- Contiguous Memory: The characters of a string are stored in contiguous memory locations. This allows for efficient access to individual characters or substrings.

Lists

- Mutable: Lists, on the other hand, are mutable. You can modify their contents (add, remove, or change elements) after they are created.

- Dynamic Arrays: Lists are implemented as dynamic arrays. This means that they can grow or shrink in size as needed. When a list runs out of space, a new, larger array is allocated in memory, and the elements are copied over.
- References: Lists don't store the actual objects they contain directly. Instead, they store references (or pointers) to the memory locations where those objects reside. This allows lists to hold objects of different types and sizes.

Note: The exact details of how strings and lists are stored in memory can vary slightly depending on the Python implementation and version you're using.

TODOs

- ☐ Project 2 (pt. 2) - Search Engine
    - Due next Monday at 11:59pm!
- ☐ HW 9 - Big O
    - Releases on Tuesday!
- ☐ Quiz 8 - Big O
    - This Friday - first 15 minutes of class!