CMPT 225: Data Structures & Programming – Unit 07 – Analysis

Dr. Jack Thomas
Simon Fraser University

Spring 2021

Today’s Topics

- Analyzing Like a Programmer
- Seven Important Functions
- The Algorithm Analysis Toolbox
  - It’s Big-Oh

What Does It Mean To Be A "Good Programmer"?

- Not a philosophy question.
- Essentially, or … he raw knowledge of … tools, and yes, structures an … **How**.
- The other hal … **hen**, that is, when to appl … hich situations to produce th …
- To do this, we … w programmers … ance.

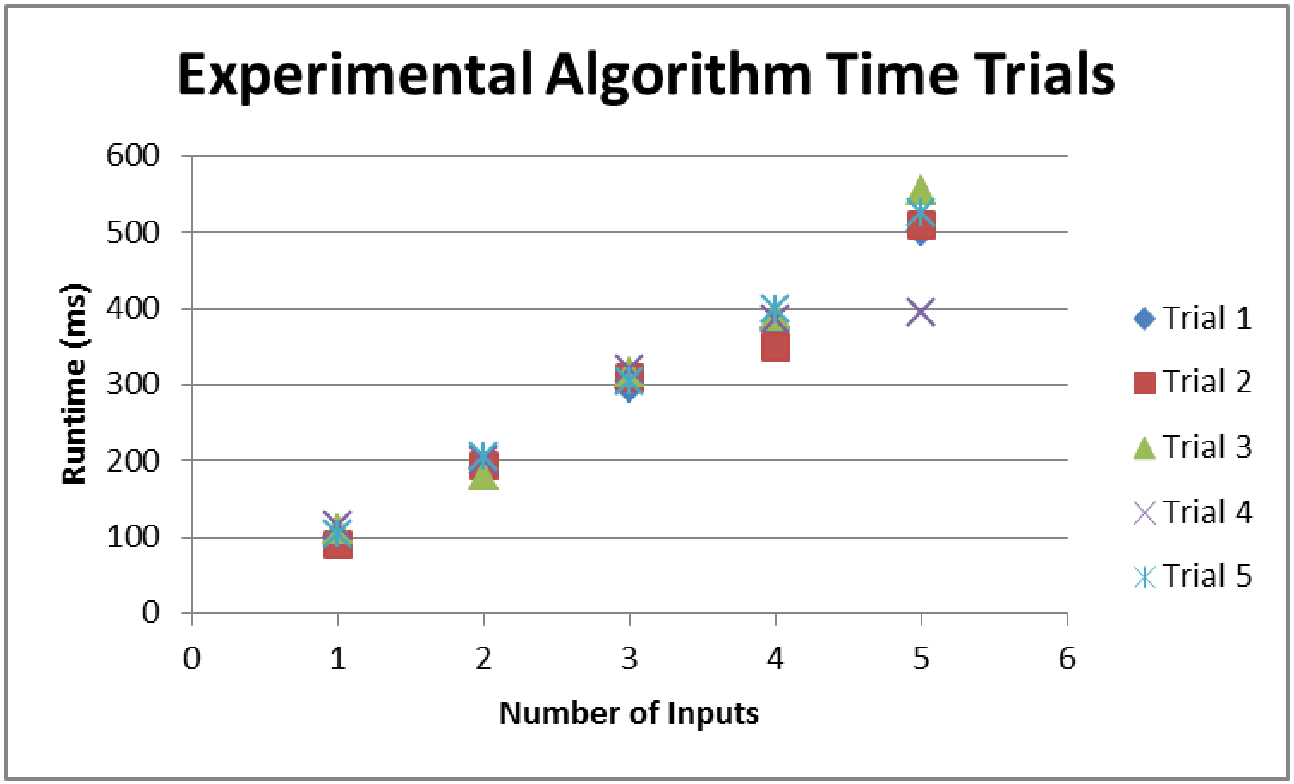
Image credit: https://en.wikipedia.org/wiki/The_Thinker

Time And Space: The Enemy

- It is not enough that your code should work, it should also be **optimal** in terms of time it takes to run and space it takes to store.
- Of the two, **optimizing for time** is usually the bigger challenge.
- Therefore, we'll focus on **measuring the speed of the algorithms** we base our functions off of.

Measuring Performance: The Experimental Approach

- One way to test the performance of a system is to literally test it – **run trials with different inputs**, measuring time to completion.



Measuring Performance: The Experimental Approach

- Generally, the goal is to determine the **dependence of running time on input size** through plotting the different trials and searching for a trend.
- Sometimes the effect of certain **input features** (e.g. sorted vs. unsorted, the colour of an image, etc) can also be discovered this way.

Experimental Drawbacks

- While experiments give us real results, there are also significant **limitations**:
  - Can normally **only test a sample** of all possible inputs.
  - **Hard to compare two algorithms** generally with all the details of their specific implementations making noise.
  - Hard to predict if performance will be similar for **different hardware or software** environments.
  - Can only reliably study **fully implemented systems**, which makes design a lot more difficult!
- We need a way to predict performance *without* having to run the test first…

Theoretical Algorithm Analysis

- The process of **analyzing the high-level pseudocode for an algorithm** to predict its time-efficiency, within some bounds of uncertainty.
- Has a concrete **procedure**, including a **notation** it's written in and standard measurements for comparison between algorithms.
- But first, let's introduce some basic elements.

Algorithm Pseudocode

- What I’ve been doing when I post Algorithms.
- A **high-level description** of the algorithm, distinct from any one programming language.
- The **syntax isn’t entirely official**, though we’ll be using the version from the recommended textbook.

Algorithm Pseudocode

- **Control flow**
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation instead of brackets
- **Method declaration**
  - **Algorithm** *method* (arg [, arg…])

    Input…

    Output …
- **Method call**
  - *var.method*(arg [, arg…])
- **Return value**
  - return *expression*
- **Expressions**
  - <- assignment (like = in Java)
  - = equals (like ==)
  - n^2 Math formatting allowed

Primitive Operations

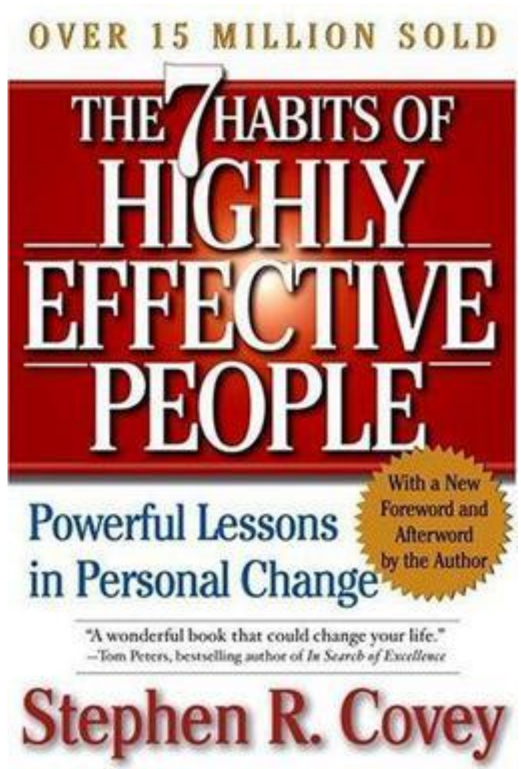
- A variety of basic actions a program can take are abstracted together as **primitive operations**.
- These include:
  - Assigning a value to a variable
  - Calling a method
  - Performing an arithmetic operation (e.g. adding)
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a method.
- These operations may take different amounts of actual time to execute, but at the speed and scale computer systems operate at, **these differences can be ignored**.

Counting Primitive Operations

- If we treat all primitive operations as **costing some constant amount of time**, our basis for measuring an algorithm's efficiency can be simply counting the number of primitive operations.
- The **actual time these operations will take will vary** a bit from each other, may vary depending on the inputs they operate on, and will certainly vary depending on the hardware or software environment, but there's still a strong correlation.

The Seven Functions of Highly Effective Programmers

- Nobody remembers this book?



- I am so old.

Image credit: https://en.wikipedia.org/wiki/The_7_Habits_of_Highly_Effective_People

Okay Seriously, the Seven Functions

- Another basic element we'll need is knowing how to **represent different growth rates** as different kinds of mathematical functions.
- When comparing algorithms, we don't usually need to narrow down their runtime to a precise amount, just a **general order of magnitude**.
- If you plotted the trial data from running the implemented system, this would be the kind of **trendline** that would best fit the graph of trial results.
- This will require some

M A T H   R E V I E W

1. Constant

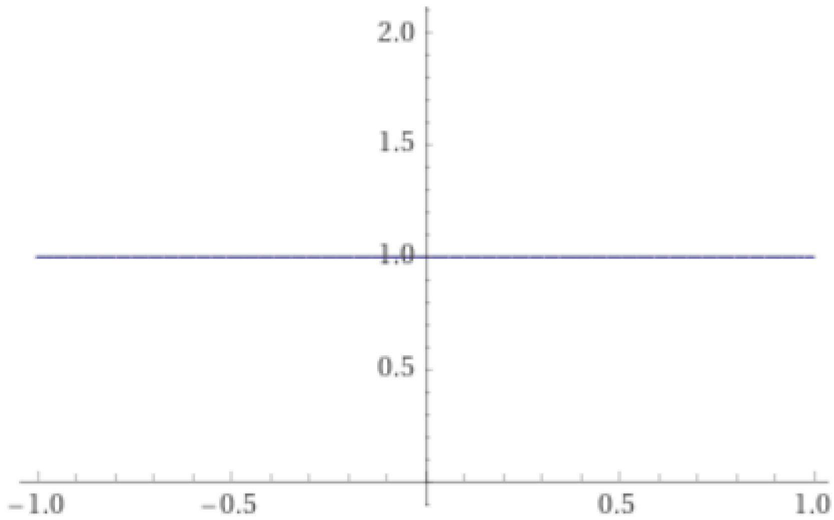


Image credit: https://www.wolframalpha.com/

1. Constant

- f(n) = c
- c is some **constant value**, meaning that no matter what value n is, the result will be c.
- In analysis terms, this usually means the function doesn’t care how big the input is, it’ll **always take the same amount of time** – say, checking if an array is empty or not.

2. Logarithmic

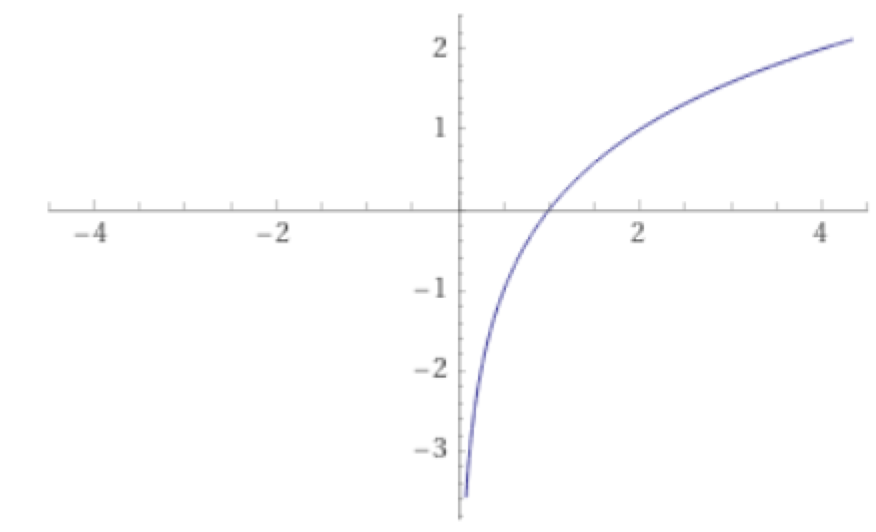


Image credit: https://www.wolframalpha.com/

2. Logarithmic

- $f(n) = \log_b n$
- b is some constant, the **base**. The rule of thumb is the result will be equal to the number of times that b can divide n.
- In computer science, **base 2 is the most common log**, to the point that it's sometimes just written as log n (some other fields do base 10 as log n, so watch out!).
- In analysis, common for functions that navigate smartly through data – a **binary search**, for example.

3. Linear

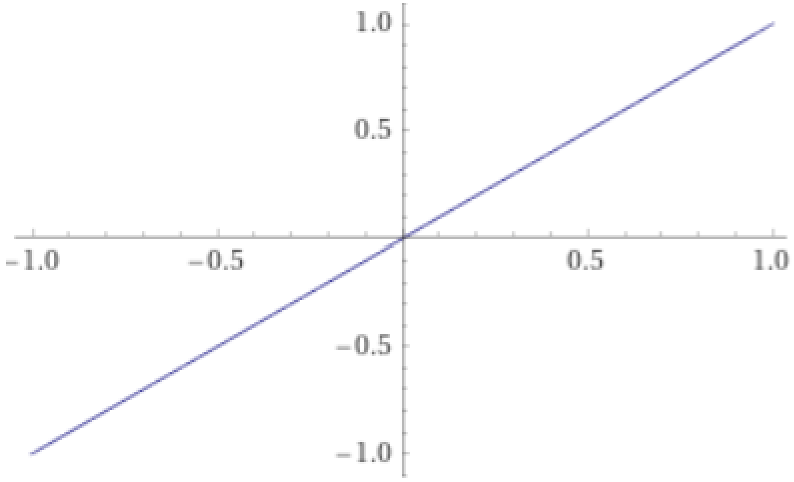


Image credit: https://www.wolframalpha.com/

3. Linear

- $f(n) = n$
- As n increases, the result increases proportionately with it.
- Typically true of functions which need to **perform some constant task for every input**, like printing every name in an array of names.

4. N-Log-N

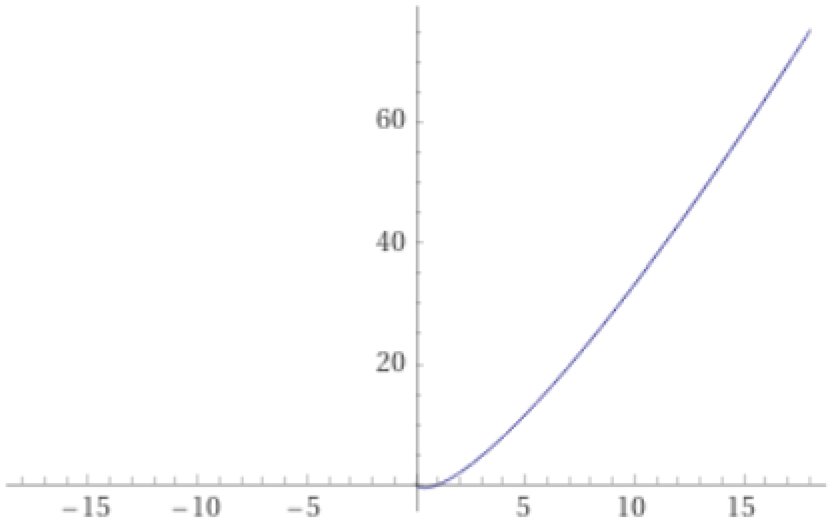


Image credit: https://www.wolframalpha.com/

4. N-Log-N

- f(n) = n log n
- As n increases, the result increases by the **product of n and log n**.
- In analysis terms, a little slower than linear, but a lot faster than n*n (quadratic), so often the result **when a function has a clever way of avoiding a quadratic outcome**. A lot of sorting cases end up here.

5. Quadratic

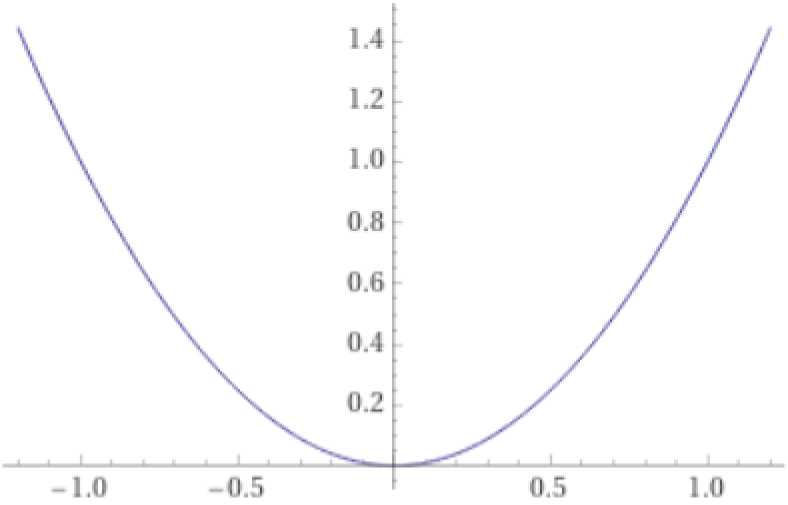


Image credit: https://www.wolframalpha.com/

5. Quadratic

- $f(n) = n^2$
- As n increases, the result is the product of **n multiplied with itself** (as in, n squared).
- Generally true of functions where **every input will have to do something with every other input** – say, applying insertion sort to an array of numbers in reverse order.

6. Cubic (and other Polynomials)

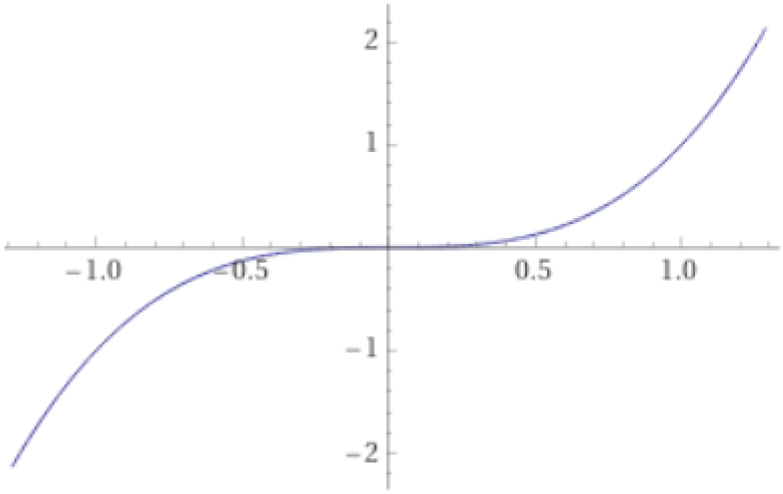


Image credit: https://www.wolframalpha.com/

6. Cubic (and other Polynomials)

- $f(n) = n^x$
- Just like quadratic, **except more acute**.
- While there's a material difference between different degrees of polynomials, in a practical sense, it's usually more important that you've ended up in this range at all.

7. Exponential

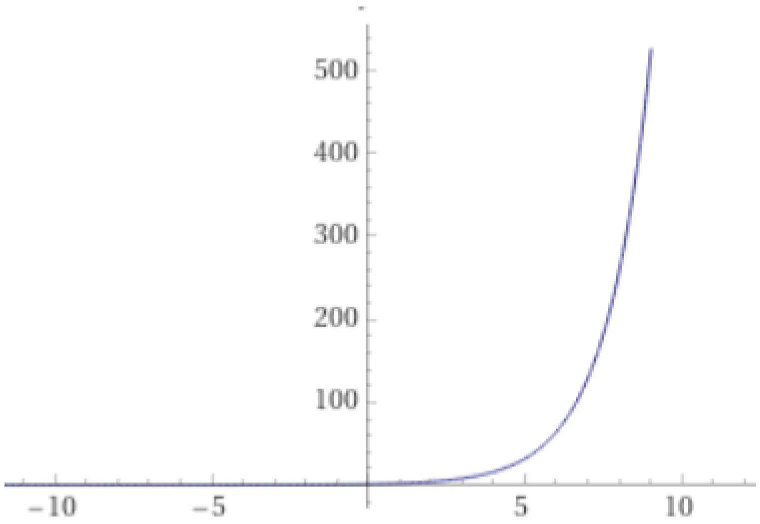


Image credit: https://www.wolframalpha.com/

7. Exponential

- $f(n) = b^n$
- b is some constant base, and every increase in n increases the result… well, **exponentially**.
- Typically **the worst-case** in analysis terms. Large values of n will make the value of b irrelevant, and become intractable even for powerful processors. To be avoided.

The Worst Case Scenario

- Which function best matches the time performance for a given algorithm **may vary depending on the inputs**.
- Knowing the **average** might be useful, but it's **very hard to predict** without knowing the nature of the inputs each implementation of the algorithm will run on.
- Easier (and often more useful) to establish an **upper bound** – the performance for the most challenging possible set of inputs.

You Knew It Was Coming: Asymptotic Analysis & Big-Oh Notation

- The process of finding the function that bounds the worst-case time performance of an algorithm is called **Asymptotic Analysis**.
- By studying the pseudocode description of an algorithm, we identify **where the running time will increase the fastest with every new input** (a loop that compares every value in an array with every other value, for example).
- We typically don't need to work out the entire function, we just need **the part that grows the fastest**.
- The way we write this function is **Big-Oh Notation**.

Defining Big-Oh

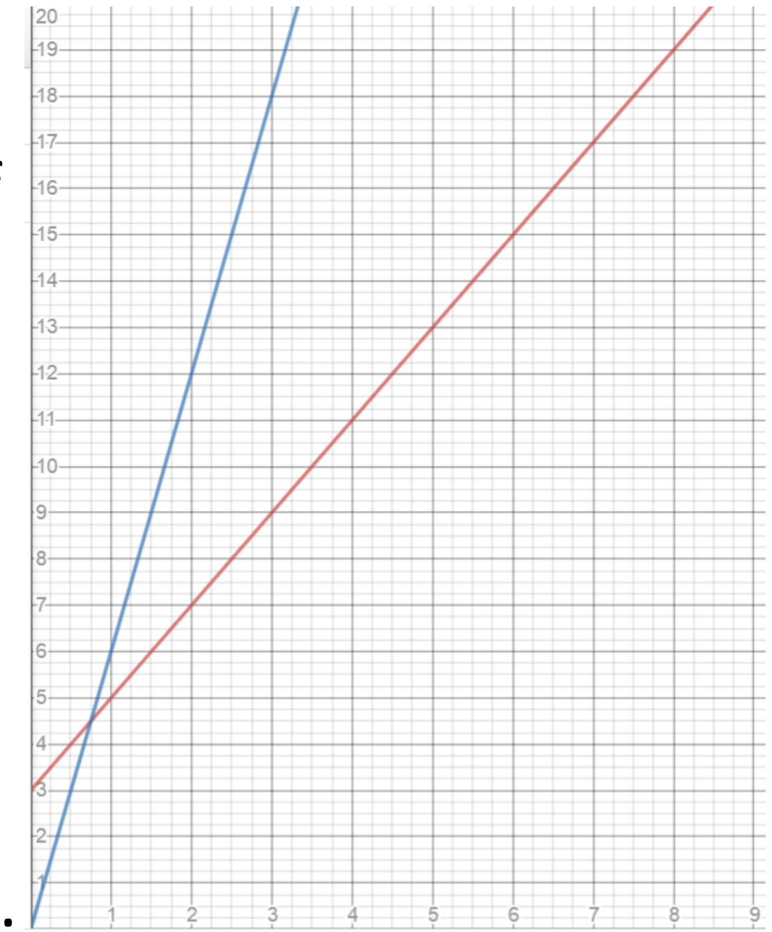
- Let **f(n)** and **g(n)** be functions mapping nonnegative integers to real numbers.
- We say that **f(n) is O(g(n))** if there is a real constant c > 0 and an integer constant n_0 >= 1 such that:

  $$f(n) <= cg(n), \text{ for } n >= n_0$$

- Therefore, we can say **f(n) is big-Oh of g(n)**, or **f(n) is order of g(n)**, or just **f(n) is O(n)**.

Defining Big-Oh

- What does that mean?
- It means that for any number of inputs, f(n) (the running time of our actual function for some n number of inputs) will be less than some constant multiplied by n.
- So f(n) will approach g(n), but never pass it, meaning g(n) bounds f(n), or f(n) **asymptotically approaches** g(n).



If f(n) were 2x+3, we can set c to 6, and make g(n) 6x, and for n>= 1, g(n) > f(n), so f(n) is O(n).

Image credit: https://www.desmos.com/calculator

Big-Omega and Big-Theta

- If Big-Oh is “less-than or equal-to”, **Big-Omega** is “greater-than or equal-to” – the lower bound, or best possible time performance.
- There’s also **Big-Theta**, which is the function that maps to the exact growth rate of our function (at least for some stretch of inputs), and will be between the two other bounds.
- Sometimes all three are the same function!

Asymptotic Analysis

- When deciding between two algorithms to solve a problem, the one with the lower O(x) will be **asymptotically better**.
- For low input values, or for non-worst-case inputs (say, a series of numbers that happens to be sorted or nearly-sorted), an asymptotically worse function **could perform better**.
- As the number of inputs increases, the asymptotically superior function **will always outperform** the competition.

Let’s Use Big-Oh!

Algorithm prefixAverages1(X):

Input: An n-element array X of numbers.

Output: An n-element array A of numbers such that A[i] is the average of elements X[0],…,X[i].

```
Let A be an array of n numbers.
for i <- 0 to n-1 do
        a <- 0
        for j <- 0 to i do
                a <- a + X[j]
        A[i] <- a/(i+1)
return array A
```

- **Initializing and returning A** takes a constant number of primitive operations per element, so **O(n)**.
- **Two nested for loops controlled by counters**, both of which are linearly dependent on n (that is, as n goes up, both counters go up proportionately), making them take n * n, or $\mathbf{O(n^2)}$

Let’s Use Big-Oh!

Algorithm prefixAverages2(X):

Input: An n-element array X of numbers.

Output: An n-element array A of numbers such that A[i] is the average of elements X[0],…,X[i].

```
Let A be an array of n numbers.
for i <- 0 to n-1 do
    s <- s + X[i]
    A[i] <- s/(i+1)
return array A
```

- **Initializing and returning** an array takes **O(n)** again.
- **Initializing the variable s** takes **O(1)**.
- There’s **just one for loop**, whose counter is controlled by n. Thus **O(n)**.
- Since $O(n) < O(n^2)$, **prefixAverages2 is asymptotically better**.

Understanding the Comparison

- As the number of inputs (n) goes up, the fastest-growing part of each method's run-time function **will come to dominate the other parts**.
- Even if prefixAverage2's full runtime ended up being 100 + n, while prefixAverage1's was just 5 + n^2, once n > 10, **prefixAverage2 would quickly overtake the competition**.
- That's why the overall O(x) for an algorithm is **the highest of the seven mathematical functions** we reviewed, rather than including all the primitive operations and lesser terms.

Tips for Analyzing Algorithms

- **Credit to Tom Shermer** for these rules of thumb.
- When analyzing an algorithm's run-time, **start by determining what n will be** – what is the input whose growth controls the run time of the function?
- If there's an array or list involved, it's probably their **size**.

Calls

- Simple assignment calls, like x = 6, are constant.
- Calls to functions, like x = array.length, take **as long as that function call takes**.
  - X = array.length would take O(1)
  - X = max(array, array.length) would take O(n)

Recursion, Conditionals, Loops

- **Recursive** functions take the time of the rest of the function, **multiplied by some value n**, depending on how the recursion is defined.
- For **conditionals** (if/else), assume the **worst condition** triggers (in terms of time), and don’t forget to measure the time the **comparison** takes!
- **Loops multiply their body** by the number of times their conditional will run.

Work Inside-Out

- Look for the **inner-most loops** (check the indentation) and start counting primitive operations.
- As you move to the **outer loops**, remember that they'll do everything in the inner loop for every term of the loop.
- A lot of loops end up **adding n** to the runtime, if they run for all inputs, unless they run in a smart way to **only have to run log n**.

Recap – Analyzing the Lecture

- Good programming means writing **optimal** (typically, **time-efficient**) code.
- We measure time by analyzing **algorithm pseudocode**, counting **primitive operations**, and matching a **growth function** to the **worst case scenario**.
- This function is described by **Big-Oh notation**, along with **Big-Omega** for best-case and **Big-Theta** for the actual growth rate.
- By comparing Big-Oh measurements for different algorithms, we can determine which one is **asymptotically better**, which is our normal standard for the optimal approach.
- Going forward, you can start analyzing the algorithms we discuss to make smart decisions about which ones to use!