CSE332: Data Structures & Parallelism

Lecture 2: Algorithm Analysis

Ruth Anderson

Autumn 2025

Administrative

- Survey – Due Monday 9/29
- EX0 – Due next Friday 10/03
- “Meet the Staff” activity
  - Sometime during the first 4 weeks of class, visit a CSE 332 office hour (in person or on zoom)
  - Tell the staff member you want to get checked off
  - You do not have to have a question about course content
  - We just want to meet you!
- Lecture MegaThread in Ed Lessons
  - We will have one of these for each lecture
  - Feel free to ask questions there during or after lecture!

Today – Algorithm Analysis

- What do we care about?
- How to compare two algorithms
- Analyzing Code
- Asymptotic Analysis
- Big-Oh Definition

What do we care about?

- Correctness:
  - Does the algorithm do what is intended.
- Performance:
  - Speed time complexity
  - Memory space complexity
- Why analyze?
  - To make good design decisions
  - Enable you to look at an algorithm (or code) and identify the bottlenecks, etc.

Q: How should we compare two algorithms?

A: How should we compare two algorithms?

- Uh, why NOT just run the program and time it??
  - Too much *variability*, not reliable or *portable*:
    - Hardware: processor(s), memory, etc.
    - OS, Java version, libraries, drivers
    - Other programs running
    - Implementation dependent
  - Choice of input
    - Testing (inexhaustive) may *miss* worst-case input
    - Timing does not *explain* relative timing among inputs (what happens when n doubles in size)

- Often want to evaluate an *algorithm*, not an implementation
  - Even *before* creating the implementation ("coding it up")

Comparing algorithms

When is one *algorithm* (not *implementation*) better than another?

- Various possible answers (clarity, security, …)
- But a big one is *performance*: for sufficiently large inputs, runs in less time (our focus) or less space

Large inputs (n) because probably any algorithm is “plenty good” for small inputs (if *n* is 10, probably anything is fast enough)

Answer will be *independent* of CPU speed, programming language, coding tricks, etc.

Answer is general and rigorous, complementary to “coding it up and timing it on some test cases”

- Can do analysis before coding!

Today – Algorithm Analysis

- What do we care about?
- How to compare two algorithms
- Analyzing Code
  - **How to count different code constructs**
  - Best Case vs. Worst Case
  - Ignoring Constant Factors
- Asymptotic Analysis
- Big-Oh Definition

Algorithm Analysis

- Usually, define a function $f: \mathbb{N} \rightarrow \mathbb{N}$
- Domain: size of the input to the code (e.g., number of elements in our array, number of characters in our string)
- Co-Domain: Counts of resources used (e.g., number of basic operations [time], number of bytes of memory used, etc.)

- Be sure you're clear on the units of your domain and co-domain
  – It won't make a big difference for this class, but in complexity theory (e.g. CSE 431, some of 421) bits of input vs. number of elements as input can make a big difference.

What Are We Counting?

- Worst case analysis
  - What's the $f(N)$ [running time, memory, etc.] for the **worst** state our data structure can be in or the **worst** input we can give of size N? (i.e. the biggest $f(N)$ could be on an input size N)
- Best case analysis
  - What is $f(N)$ for the **best** state of our structure and the best question of size N? (the smallest $f(N)$ could be)
- Average case analysis
  - What is the value of $f(N)$ on average over all possible inputs of size N?
  - Have to ask this question very carefully to get a meaningful answer
- **We usually do worst case analysis.**

Analyzing code ("worst case")

Basic operations take "some amount of" constant time

- Arithmetic
- Assignment
- Access one Java field **or array index**
- Etc.

(This is an *approximation of reality*: a very useful "lie".)

Consecutive statements	Sum of time of each statement
Loops	Num iterations * time for loop body
Conditionals	Time of condition plus time of slower branch
Function Calls	Time of function's body
Recursion	Solve *recurrence equation*

Examples

```
b = b + 5
c = b / a
b = c + 100

for (i = 0; i < n; i++) {
     sum++;
 }

if (j < 5) {
   sum++;
} else {
  for (i = 0; i < n; i++) {
     sum++;
   }
}
```

Another Example

```
int coolFunction(int n, int sum) {
   int i, j;
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
        sum++;
      }
   }
   print "This program is great!"
   for (i = n; i > 1; i = i / 2) {
        sum++;
   }
   return sum
}
```

Today – Algorithm Analysis

- What do we care about?
- How to compare two algorithms
- Analyzing Code
  - How to count different code constructs
  - **Best Case vs. Worst Case**
- Asymptotic Analysis
- Big-Oh Definition

Linear search – Best Case & Worst Case

2	3	5	16	37	50	73	75	126

Find an integer in a *sorted* array

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
   for(int i=0; i < arr.length; ++i)
      if(arr[i] == k)
        return true;
   return false;
}
```

Best case:

Worst case:

Asymptotic Notation

- That's a nice formula. But does everything in it matter?
- Multiplying by constant factors doesn't matter – let's just ignore them.
- Lower order terms don't matter – delete them.
- Gives us a "simplified big-O"

- $10n \log n + 3n$
- $5n^2 \log n + 13n^3$
- $20n \log \log n + 2\, n \log n$
- 2^{3n}

Asymptotic Notation

- That's a nice formula. But does everything in it matter?
- Multiplying by constant factors doesn't matter – let's just ignore them.
- Lower order terms don't matter – delete them.
- Gives us a "simplified big-O"

- $10n\log n + 3n$ — $O(n\log n)$
- $5n^2\log n + 13n^3$ — $O(n^3)$
- $20n\log\log n + 2\,n\log n$ — $O(n\log n)$
- 2^{3n} — $O(8^n)$

Today – Algorithm Analysis

- What do we care about?
- How to compare two algorithms
- Analyzing Code
  - How to count different code constructs
  - Best Case vs. Worst Case, and more
- Asymptotic Analysis
- Big-Oh Definition

Formally Big-O

- We wanted to find an upper bound on our algorithm's running time, but
  - We don't want to care about constant factors.
  - We only care about what happens as n gets large.
- The formal, mathematical definition is Big-O.

Big-O

$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$$f(n) \leq c \cdot g(n)$$

We also say that $g(n)$ "dominates" $f(n)$.

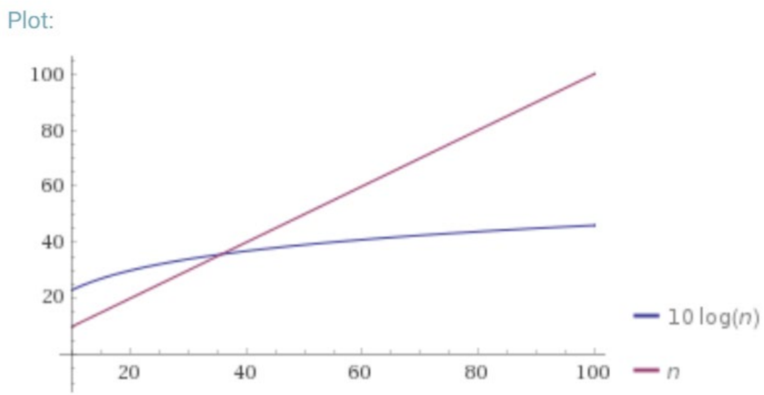
Why is that the definition?

Big-O

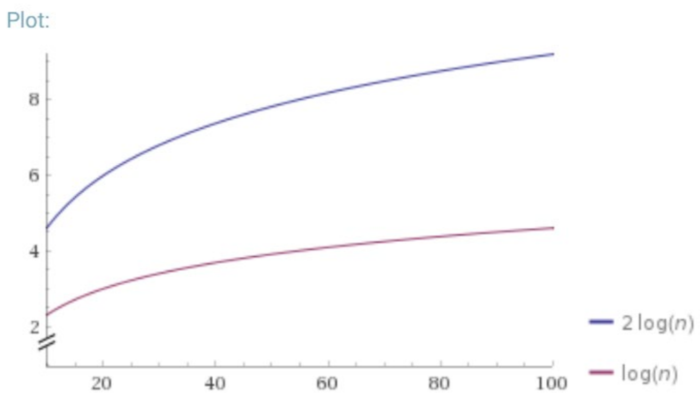
$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$$f(n) \leq c \cdot g(n)$$

- Why n_0?



Why c?



Why Are We Doing This?

- You already intuitively understand what big-O means.
- Who needs a formal definition anyway?
  - We will.
- Your intuitive definition and my intuitive definition might be different.
- We're going to be making more subtle big-O statements in this class.
  - We need a mathematical definition to be sure we're on the same page.
- Once we have a mathematical definition, we can go back to intuitive thinking.
  - But when a weird edge case, or subtle statement appears, we can figure out what's correct.

Edge Cases

- True or False: $10n^2 + 15n$ is $O(n^3)$
- [this is an edge case]

Edge Cases

- True or False: $10n^2 + 15n$ is $O(n^3)$
- [this is an edge case]
- **It's true!** – it fits the definition.
- Big-O is just an **upper bound**. It doesn't have to be a "good" upper bound.
- If we want the "best" upper bound, we'll ask you for a **tight** big-O bound.
- $O(n^2)$ is the tight bound for this example.
- It is (usually) technically correct to say your code runs in time $O(n^{n!})$.
  – DO NOT TRY TO PULL THIS TRICK ON AN EXAM. Or in an interview.

O, Omega, Theta [oh my?]

- Big-O is an **upper bound**
  - My code uses at most this many resources (e.g. runs in at most this much time)
- Big-Omega is a lower bound

Big-Omega

$f(n)$ is $\Omega(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$$f(n) \geq c \cdot g(n)$$

- Big Theta is "equal to"

Big-Theta

$f(n)$ is $\Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Viewing O as a class

- Sometimes you'll see big-O defined as a family or set of functions.

Big-O (alternative definition)

$O(g(n))$ **is the set of all functions** $f(n)$ **such that there exist positive constants** c, n_0 **such that for all** $n \geq n_0$**,** $f(n) \leq c \cdot g(n)$

For that reason, some people write $f(n) \in O\left(g(n)\right)$ **where we wrote "**$f(n)$ **is** $O(g(n))$**".**

Other people write "$f(n) = O\left(g(n)\right)$**" to mean the same thing.**

We never write $O(5n)$ **instead of** $O(n)$ **– they're the same thing!**

It's like writing $\frac{6}{2}$ **instead of** 3**. It just looks weird.**

Common Categories

- The most common running times all have fancy names:
- $O(1)$ constant
- $O(\log n)$ logarithmic
- $O(n)$ linear
- $O(n \log n)$ "n log n"
- $O(n^2)$ quadratic
- $O(n^3)$ cubic
- $O(n^c)$ polynomial (where c is a constant)
- $O(c^n)$ exponential (where c is a constant)

What's the base of the log?

- If I write $\log n$, without specifying a base, I mean $\log_2 n$.

- But does it matter for big-O?
- Suppose we found an algorithm with running time $\log_5 n$ instead.
- Is that different from $O(\log_2 n)$?
- No!
- $\log_c n = \frac{\log_2 n}{\log_2 c}$ If c is a constant, then $\log_2 c$ is just a constant, and we can hide it inside the $O()$.

Review: Properties of logarithms

- **`log(A*B) = log A + log B`**
  - So $\log(N^k) = k \log N$
- **`log(A/B) = log A – log B`**
- $x = \log_2 2^x$
- **`log(log x)`** is written **`log log x`**
  - Grows as slowly as 2^{2^y} grows fast
  - Ex:
$$\log_2 \log_2 4billion \sim \log_2 \log_2 2^{32} = \log_2 32 = 5$$
- **`(log x)(log x)`** is written $\log^2 x$
  - It is greater than **`log x`** for all **`x > 2`**

O, Ω, Θ *vs. Best, Worst, Average*

- It's a common misconception that $\Omega()$ is "best-case" and $O()$ is "worst-case". This is a misconception!!
- $O()$ says "the complexity of this algorithm is at most" (think $\leq$)
- $\Omega()$ says "the complexity of this algorithm is at least" (think $\geq$)
- You can use $\leq$ on worst-case or best case; you can use $\geq$ on worst-case or best-case.
- Best/Worst/Average say "what function f am I analyzing?"
- O, Ω, Θ say "let me summarize what I know about f, it's $\leq, \geq, = \ldots$"