COMPSCI 311 Introduction to Algorithms

Lecture 2: Asymptotic Notation and Efficiency

Marius Minea

University of Massachusetts Amherst

slides credit: Dan Sheldon, Akshay Krishnamurthy, Andrew McGregor

28 January 2019

Algorithm Design

- Formulate the problem precisely
- Design an algorithm to solve the problem
- Prove the algorithm is correct
- Analyze the algorithm's running time

Big-O: Motivation

What is the running time of this algorithm?
How many "primitive steps" are executed for an input of size n?

```
sum = 0
for i= 1 to n do
    for j= 1 to n do
        sum += A[i]*A[j]
    end for
end for
```

The running time is

$$T(n) = ? \cdot n^2 + ? \cdot n + ? \ .$$

What are the coefficients?

For large values of n, $T(n)$ is *less* than some multiple of n^2.
We say $T(n)$ is $O(n^2)$ and typically don't care about other terms.

Big-O: Formal Definition

Definition: The function $T(n)$ is $O(f(n))$ (read: "is order $f(n)$") if there exist constants $c > 0$ and $n_0 \geq 0$ such that

$$T(n) \leq cf(n) \text{ for all } n \geq n_0$$

We say that f is an asymptotic upper bound for T.

Example:

$$\begin{aligned} T(n) &= 2n^2 + n + 2 \\ &\leq 2n^2 + n^2 + 2n \qquad \text{if } n \geq 1 \\ T(n) &\leq \underbrace{5}_{c} n^2 \qquad \text{if } n \geq \underbrace{1}_{n_0} \end{aligned}$$

So $T(n)$ is $O(n^2)$

Clicker Question 1

Claim $n^3 + 10^6 n$ is $O(n^3)$

To prove this we need to show that

$$n^3 + 10^6 n \leq cn^3 \quad \text{for all } n \geq n_0$$

Which values of c and n_0 make this inequality true?

A. $c = 2$, $n_0 = 1000$

B. $c = 101$, $n_0 = 100$

C. Both A and B

D. Neither A nor B

Big-O: Examples

- If $T(n) = n^2 + 1000000n$ then $T(n)$ is $O(n^2)$

  $c = 2, n_0 = 10^6$
- If $T(n) = n^3 + n \log n$ then $T(n)$ is $O(n^3)$

  $c = 2, n_0 = 1$, since $\log n < n$
- If $T(n) = 2^{\sqrt{\log n}}$ then $T(n)$ is $O(n)$

  $c = 1, n_0 = 1$, since $\sqrt{\log n} \leq \log n$ and $2^{\log_n} = n$

Big-O: Reviewing Definition

Big-O is a **relation** between two functions

$f(n) = O(g(n)$ means $\exists c > 0, n_0 \geq 0 : f(n) \leq cg(n)$ for $n \geq n_0$.

There is **no unique** function $g(n)$ so that $f(n) = O(g(n))$

Trivially, $f(n) = O(f(n))$: take $c = 1, n_0 = 0$

We also have $f(n) = O(\frac{1}{2}f(n))$: take $c = 2, n_0 = 0$

We also have $f(n) = O(nf(n))$; take $c = 1, n_0 = 1$, etc.

Whether $f(n) = O(g(n))$ does not depend on

- multiplying f or g by a constant (we can choose c)
- the first 2 or 5 or 1000 etc. values (we can choose n_0)

Clicker Question 2

Let $f(n) = 3n^2 + 4n \log_2 n + 5$. Which of the following are true?

A. $f(n)$ is $O(n^2)$

B. $f(n)$ is $O(n^2 \log_2 n)$

C. Both A and B

D. Neither A nor B

How to Use Big-O

Analyze pseudocode to determine running time $T(n)$ of algorithm as a function of n:

$$T(n) = 2n^2 + n + 2$$

Prove that $T(n)$ is asymptotically upper-bounded by a simpler function using definition of big-O:

$$\begin{aligned} T(n) &= 2n^2 + n + 2 \\ &\leq 2n^2 + n^2 + 2n^2 && \text{if } n \geq 1 \\ &\leq 5n^2 && \text{if } n \geq 1 \end{aligned}$$

This is right, but too much work. We will:

- **prove properties** of big-O that simplify finding big-O bounds,
- **use** these properties to **take "shortcuts"** when analyzing algorithms

(you probably learned the shortcuts without formal justification).

Properties of Big-O: Transitivity

Claim (Transitivity): If f is $O(g)$ and g is $O(h)$, then f is $O(h)$.

Example:

- $\underbrace{2n^2 + n + 1}_{f(n)}$ is $O(\underbrace{n^2}_{g(n)})$
- $\underbrace{n^2}_{g(n)}$ is $O(\underbrace{n^3}_{h(n)})$
- Therefore, $2n^2 + n + 1$ is $O(n^3)$

Transitivity Proof

Claim (Transitivity): If f is $O(g)$ and g is $O(h)$, then f is $O(h)$.

Proof: we know from the definition that

- $f(n) \leq cg(n)$ for all $n \geq n_0$
- $g(n) \leq c'h(n)$ for all $n \geq n'_0$

Therefore

$$\begin{aligned} f(n) &\leq cg(n) && \text{if } n \geq n_0 \\ &\leq c(c'h(n)) && \text{if } n \geq n_0 \text{ and } n \geq n'_0 \\ &= \underbrace{cc'}_{c''} h(n) && \text{if } n \geq \underbrace{\max\{n_0, n'_0\}}_{n''_0} \\ f(n) &\leq c''h(n) && \text{if } n \geq n''_0 \end{aligned}$$

Know how to do proofs using Big-O definition.

Properties of Big-O: Additivity

Claims (Additivity):

- If f is $O(h)$ and g is $O(h)$, then $f + g$ is $O(h)$.

$$\underbrace{3n^2}_{O(n^5)} + \underbrace{n^4}_{O(n^5)} \text{ is } O(n^5)$$

- If f is $O(g)$, then $f + g$ is $O(g)$

$$\underbrace{n^3}_{g(n)} + \underbrace{23n + n \log n}_{f(n)} \text{ is } O(n^3)$$

Using Additivity

- OK to drop lower order terms:

$$2n^5 + 10n^3 + 4n\log n + 1000n \text{ is } O(n^5)$$

- Polynomials: Only highest-degree term matters. If $a_d > 0$ then:

$$a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d \text{ is } O(n^d)$$

- You are using additivity when you ignore the running time of statements outside for loops!

Other Useful Facts: Log vs. Poly vs. Exp

Fact: $\log_b(n)$ is $O(n^d)$ for all $b, d > 0$

All polynomials grow faster than logarithm of any base

Fact: n^d is $O(r^n)$ when $r > 1$

Exponential functions grow faster than polynomials

Logarithm review

Definition: $\log_b(a)$ is the unique number c such that $b^c = a$

Informally: the number of times you can divide a into b parts until each part has size one

Properties:

- Log of product $\rightarrow$ sum of logs
  - $\log(xy) = \log x + \log y$
  - $\log(x^k) = k \log x$
- $\log_b(\cdot)$ is inverse of $b^{(\cdot)}$
  - $\log_b(b^n) = n$
  - $b^{\log_b(n)} = n$
- $\log_a n = log_a b \cdot log_b n$ (logs in any two bases are proportional)

When using big-O, it's OK not to specify base.
Assume $\log_2$ if not specified.

Big-O comparison

Which grows faster?

$$n(\log n)^3 \quad \text{vs.} \quad n^{4/3}$$

divide by common factor n, simplifies to:

$$(\log n)^3 \quad \text{vs.} \quad n^{1/3}$$

take cubic root, simplifies to:

$$\log n \quad \text{vs.} \quad n^{1/9}$$

- We know $\log n$ is $O(n^d)$ for all $d > 0$
  - $\Rightarrow \log n$ is $O(n^{1/9})$
  - $\Rightarrow n(\log n)^3$ is $O(n^{4/3})$

Apply transformations (monotone, invertible) to both functions.
Try taking log.

Big-O: Correct Usage

Big-O: a way to categorize growth rate of functions relative to other functions.

Not: "***the*** running time of my algorithm".

Correct Usage:

- Worst-case running time of algorithm in input of size n is $T(n)$.
- $T(n)$ is $O(n^3)$.
- The running time of the algorithm is $O(n^3)$.

Incorrect Usage:

- $O(n^3)$ is ***the*** running time of the algorithm.
  (There are many different asymptotic upper bounds to the running time of the algorithm.)

Big-Ω Motivation

Algorithm **foo**

```
for i = 1 to n do
    for j = 1 to n do
        do something...
    end for
end for
```

Fact: run time is $O(n^3)$

Algorithm **bar**

```
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
            do something else..
        end for
    end for
end for
```

Fact: run time is $O(n^3)$

Conclusion: **foo** and **bar** have the same asymptotic running time.
What is wrong?

More Big-Ω Motivation

Algorithm **sum-product**

```
sum = 0
for i= 1 to n do
    for j= i to n do
        sum += A[i]*A[j]
    end for
end for
```

What is the running time of **sum-product**?

Easy to see it is $O(n^2)$. Could it be better? $O(n)$?

Big-Ω

Informally: T grows at least as fast as f

Definition: The function $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that

$$T(n) \geq cf(n) \text{ for all } n \geq n_0$$

f is an asymptotic lower bound for T

Clicker Question 3

Which is an equivalent definition of big Omega notation?

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

B. $f(n)$ is $\Omega(g(n))$ if for any $n \geq 0$ there exists a constant $c > 0$ such that $f(n) \geq c \cdot g(n)$

C. Both A and B

D. Neither A nor B

Big-Ω Exercise

Let $T(n)$ be the running time of **sum-product**.
Show that $T(n)$ is $\Omega(n^2)$

Algorithm **sum-product**

```
sum = 0
for i= 1 to n do
    for j= i to n do
        sum += A[i]*A[j]
    end for
end for
```

Big-Ω: Solution

Hard way

- Count exactly how many times the loop executes

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2} = \Omega(n^2)$$

Easy way

- Ignore all loop executions where $i > n/2$ or $j < n/2$
- The inner statement executes at least $(n/2)^2 = \Omega(n^2)$ times

For Big-O, we can approximate upwards
For Ω, we can approximate downwards (ignore some computation)