CMPSCI 311: Introduction to Algorithms

Lecture 2: Asymptotic Notation and Efficiency

Akshay Krishnamurthy

University of Massachusetts

Last Compiled: January 24, 2018

Announcements:

- Homework 1 released (website, Moodle, Gradescope)
- No discussion on Friday
- Quiz 1 out on Friday

Recap: Stable Matching

- Given n students and n colleges, each with preferences over the other. Can we find a stable matching?
  - *Stability:* Don't want to match c with s and c' with s' if c and s' would prefer to switch to being matched with each other.
- Yes! Propose and Reject Algorithm.
  - Algorithm terminates in n^2 iterations
  - Everyone gets matched
  - Resulting matching is stable!

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

$$T(n) = 3n^2 + n + 1 \ .$$

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

Big-O: Formal Definition

Definition: The function $T(n)$ is $O(f(n))$ if there exist constants $c \geq 0$ 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.

Examples:

- If $T(n) = n^2 + 1000000n$ then $T(n)$ is $O(n^2)$
- If $T(n) = n^3 + n \log n$ then $T(n)$ is $O(n^3)$
- If $T(n) = 2^{\sqrt{\log n}}$ then $T(n)$ is $O(n)$
- If $T(n) = n^3$ then $T(n)$ is $O(n^4)$ but it's also $O(n^3)$, $O(n^5)$ etc.

Properties of Big-O Notation

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

Claims (Additivity):

- If f is $O(h)$ and g is $O(h)$, then $f + g$ is $O(h)$.
- If $f_1, f_2, \ldots, f_k$ are each $O(h)$, then $f_1 + f_2 + \ldots + f_k$ is $O(h)$.
- If f is $O(g)$, then $f + g$ is $O(g)$.

We'll go through a couple of examples...

Consequences of Additivity

- OK to drop lower order terms. E.g., if

$$f(n) = 4.1n^3 + 23n + n \log n$$

then $f(n)$ is $O(n^3)$

- Polynomials: Only highest degree term matters. E.g., if

$$f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d, \quad a_d > 0$$

then $f(n)$ is $O(n^d)$

Other Useful Facts: Log vs. Poly vs. Exp

Fact: $\log_b(n)$ is $O(n^d)$ for all b and d

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

Challenge problem: Prove these facts!

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

- $\log(xy) = \log x + \log y$
- $\log(x^k) = k \log x$
- $\log_b(b^n) = n$
- $b^{\log_b(n)} = n$
- $\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$

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 $c \geq 0$ 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

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
```

Do on board: easy way and hard way

Exercise review

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

Big-Θ

Definition: the function $T(n)$ is $\Theta(f(n))$ if it is both $O(f(n))$ and $\Omega(f(n))$.

f is an asymptotically tight bound of T

Big-Θ example

How do we correctly compare the running time of these algorithms?

Algorithm **foo**

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

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
```

Answer: **foo** is $\Theta(n^2)$ and **bar** is $\Theta(n^3)$. They do not have the same asymptotic running time.

Additivity Revisited

Suppose f and g are two (non-negative) functions and f is $O(g)$

Old version: Then $f + g$ is $O(g)$

New version: Then $f + g$ is $\Theta(g)$

Example:

$$\underbrace{n^2}_{g} + \underbrace{42n + n \log n}_{f} \text{ is } \Theta(n^2)$$

Algorithm design

- Formulate the problem precisely
- Design an algorithm to solve the problem
- Prove the algorithm is correct
- Analyze the algorithm's running time # Running Time Analysis (K&T, Ch. 2)
- What is efficiency?
- Mathematical foundations: asymptotic growth of functions, big-O and friends
- Skills: analyze big-O running time of algorithms

Approach

Mathematical analysis of worst-case running time of an algorithm as function of input size. Why these choices?

- Mathematical: describes the *algorithm*. Avoids hard-to-control experimental factors (CPU, programming language, quality of implementation)
- Worst-case: just works. ("average case" appealing, but hard to analyze)
- Function of input size: allows predictions. What will happen on a new input?

Notions of Efficiency

When is an algorithm efficient? Consider stable matching...

Brute force: $O(n!)$
Gale-Shapley?: $O(n^2)$

We must have done something clever

Question: Is it $\Omega(n^2)$?

Polynomial Time

Working definition of efficient

Definition: an algorithm runs in polynomial time if the number of primitive execution steps is at most cn^d, where n is the input size and c and d are constants.

Polynomial Time

Examples of polynomial time:

$f_1(n) = n$
$f_2(n) = 4n + 100$
$f_3(n) = n\log(n) + 2n + 20$
$f_4(n) = 0.01n^2$
$f_5(n) = n^2$
$f_6(n) = 20n^2 + 2n + 3$

Not polynomial time:

$f_7(n) = 2^n$
$f_8(n) = 3^n$
$f_9(n) = n!$

Polynomial Time

Why is this a good definition of efficiency?

- Matches practice: almost all practically efficient algorithms have this property
- Usually distinguishes a clever algorithm from a "brute force" approach
- Refutable: gives us a way of saying an algorithm is not efficient, or that no efficient algorithm exists.