EE360C: Algorithms

The Basics

Summer 2019

Department of Electrical and Computer Engineering
University of Texas at Austin

Algorithms

Definition of an Algorithm

Definition 1: Algorithm

An *algorithm* is any step-by-step procedure to solve a problem.

An algorithm is:

- a sequence of computational steps that transform the input into an output
- a tool for solving a well-specified computational problem
- said to be *correct* if, for every input instance, it halts with the correct output
- said to *solve* a computational problem if it is correct

Algorithm Efficiency

Fundamental Issues in Algorithms

We'll talk about two fundamental issues in algorithms:

- analysis
- design

Algorithm Analysis Basics

Analyzing an algorithm refers to predicting the resources (both amount of time and space) that the algorithm requires, and how these resources scale with increasing input sizes.

In the next few lectures, we will make this notion concrete, and develop the mathematical machinery needed.

But What's Our Goal?

Our goal is to develop (correct) *efficient* algorithms as solutions to well-defined problems.

Let's consider some working definitions...

Algorithm Efficiency Definition 1

First try

An algorithm is efficient if, when implemented, it runs quickly on real input instances.

This is a good start, but...

- it's awfully vague
- even bad algorithms can run fast when applied to small test cases
- even good algorithms can run slow when implemented poorly
- what is a "real" input instance? (part of the problem is that we don't know the range on possible inputs *a priori*)
- this definition doesn't consider how well the algorithm's performance *scales* as the problem size grows

Algorithm Efficiency Definition 1 (cont.)

We would like a definition of algorithm efficiency that is:

- platform-independent
- instance-independent
- of predictive value with respect to increasing instance sizes

Example

The stable matching problem

- What is the size of the problem? Before we can answer that, we need to know:
- What must be input to run the algorithm? The input preference lists of all participants.

A problem instance has a "size" N, which is the total size of the input preference lists (what must be input to run the algorithm).

- there are n men and n women
- each has a preference list, so there are $2n$ lists
- each list is of size n
- $N = 2n \times n = 2n^2$

Algorithm Efficiency Definition 1 (cont.)

Input Size

The definition of **input size** depends on the particular computational problem being studied.

- As a general rule, the running time grows with the size of the input

Running Time

The **running time** of an algorithm on a particular input is the number of primitive operations or “steps” executed.

- running time should be machine independent
- we assume a constant amount of time is required to execute each line of pseudocode

Algorithm Efficiency Definition 2

Worst-Case Running Time

The *worst-case* running time of an algorithm is the worst possible running time the algorithm could have over all inputs of size N.

- Is worst-case the best measure? Generally yes.
- Average-case analysis—study of performance of an algorithm over "random" instances. How does one pick random instances?
- What is a reasonable analytical benchmark to compare against to tell us whether the running time is good or not? A simple guide is to compare against a brute-force search over the space of all possible solutions.

Algorithm Efficiency Definition 2 (contd.)

Second Try

An algorithm is efficient if it achieves qualitatively better worst-case performance, at an analytical level, than brute-force search.

Consider the stable matching problem (SMP) that uses brute-force:

- If there are n men and n women, how many possible perfect matchings exist? $n!$
- For each of these possible pairings, find if an instability exists, i.e. If $(m, w) \notin S$, find m' such that $(m', w) \in S$ and $m > m'$, and find w' such that $(m, w') \in S$ and $w > w'$. What is the worst-case running time? n^2
- Therefore, worst-case running time for a brute-force search for SMP is of the order $(n!n^2)$.
- What was the worst-case running time for G-S algorithm? n^2.

Algorithm Efficiency Definition 3

But the definition is still a little vague. What is "qualitatively better"?

A better working definition is:

Third (and Final) Try

An algorithm is efficient if it has polynomial running time.

What is polynomial time?
For any algorithm, if there exists $c > 0$ and $d > 0$, such that for every input instance of size N, the running time is bounded by cN^d computational steps.
If this running-time bound holds, then we say the algorithm has a polynomial running time.

Algorithm Efficiency Definition 3 (contd.)

Third (and Final) Try

An algorithm is efficient if it has polynomial running time.

This definition is precise, but is also *negatable*.

Of course, running time of n^{100} is clearly not great, and a running time of $n^{1+.02(\log n)}$ is not clearly bad. But in practice, polynomial time is generally good.

Why it Matters

Running times for algorithms on inputs of increasing size on a processor executing a million instructions a second. (*Note: very long means running time exceeded* 10^{25} *years.*)

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Asymptotic Notation

Asymptotic Notation

We study the asymptotic efficiency of algorithms; i.e., how the running time scales with increasing input size in the limit.

An algorithm with the best asymptotic performance will be the best choice for all but very small inputs.

The goal is to identify broad classes of algorithms with similar behavior.

We measure running times in the number of primitive "steps" an algorithm must perform. We will be counting steps in pseudocode.

Asymptotic Bounds

We don't need to be overly precise about running times, since we care about the rates of growth.

Asymptotic Bounds

We want to represent the asymptotic running time of an algorithm (its growth rate relative to the input size) independent of any constant factors.

Asymptotic Upper Bounds: O-Notation

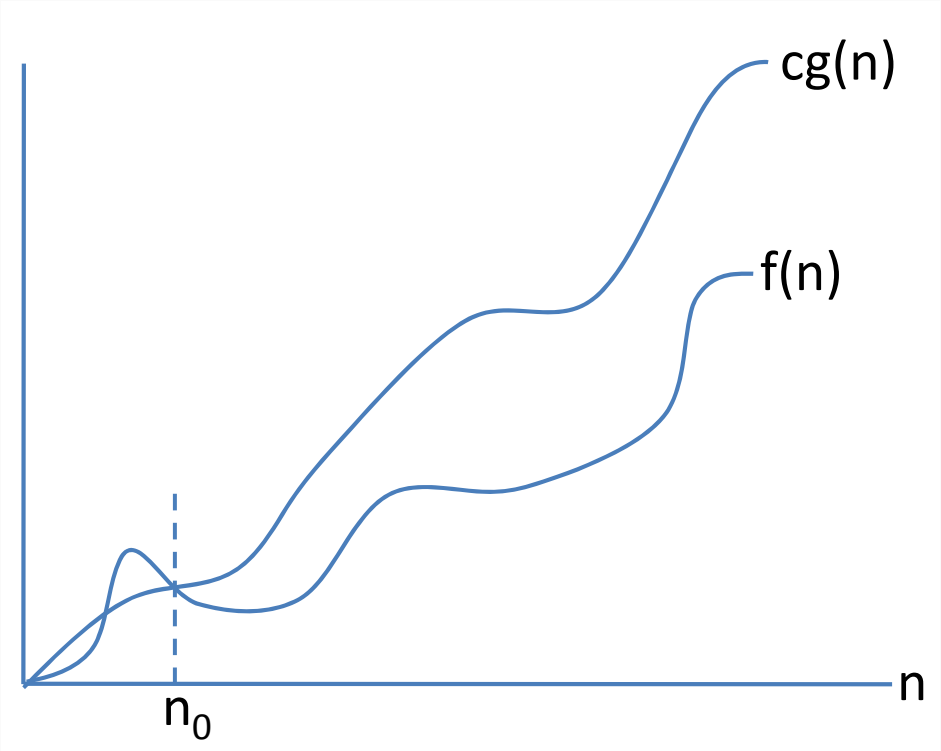
Let $f(n)$ be the worst-case running time of a certain algorithm on an input size n.
Given another function $g(n)$, we say $f(n)$ is $O(g(n))$ ("$f(n)$ is order $g(n)$") if, for sufficiently large n, $f(n)$ is bounded above by a constant multiple of $g(n)$.

Definition 2

$f(n)$ is $O(g(n))$ if
there exist positive constants $c > 0$ and $n_0 \geq 0$ such that
$0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

- f is *asymptotically upper-bounded by* g.
- $O(g(n))$ is a set, but we usually abuse notation and write: $f(n) = O(g(n))$.

O-Notation (cont.)



- For all values of n to the right of n_0, the value of $f(n)$ is on or below $cg(n)$
- We say that $g(n)$ is an **upper bound** for $f(n)$

O-Notation: An Example

Claim

$T(n) = pn^2 + qn + r$ is in $O(n^2)$, for positive constants p, q, r.

$\forall n \geq 1$, we have $qn \leq qn^2$, and $r \leq rn^2$, so we can write
$T(n) = pn^2 + qn + r \leq pn^2 + qn^2 + rn^2 = (p + q + r)n^2$
for all $n \geq 1$.

This is the required definition of $O(\cdot)$:
$T(n) \leq cn^2$, where $c = p + q + r$.

O-Notation: Another Example

Claim

$T(n) = pn^2 + qn + r$ is in $O(n^3)$, for positive constants p, q, r.

- $T(n) \leq (p + q + r)n^2$
- $n^2 \leq n^3$
- $T(n) \leq (p + q + r)n^3$

Some Notes on O-Notation

- both $O(n^3)$ and $O(n^2)$ are vaild upper bounds for $T(n)$. It's just that $O(n^3)$ was not the "tightest" possible upper bound.
- Some texts use O to informally describe asymptotically tight bounds (i.e., when we use Θ)
- O-notation can be useful in quickly and easily bounding the running time of an algorithm by inspection

Asymptotic Lower Bounds: Ω-Notation

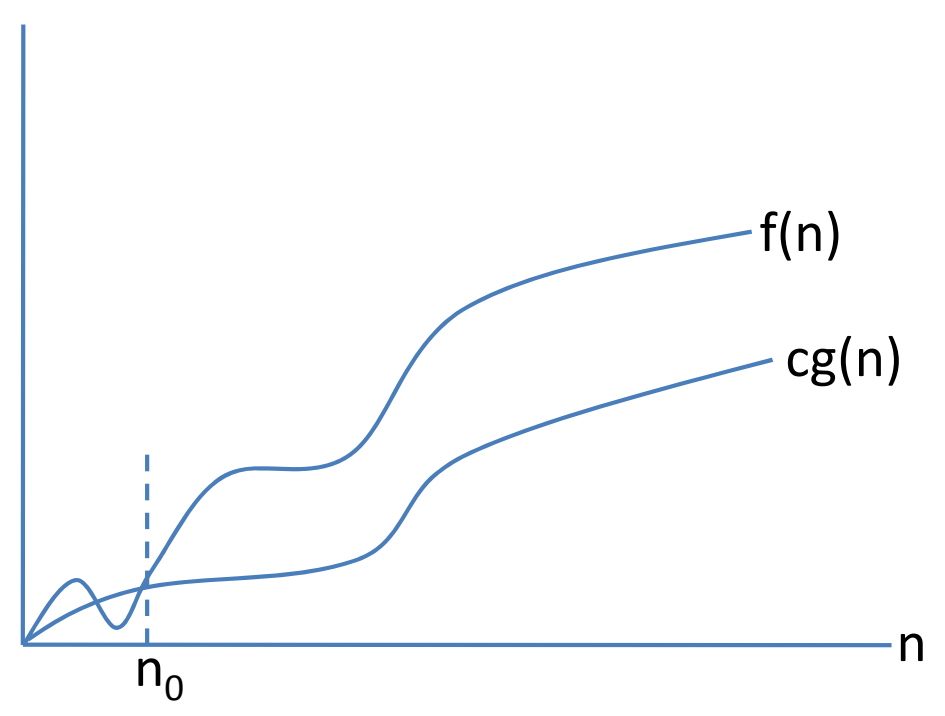
Lower bounds can be useful for stating that an algorithm's running time is at least some magnitude

Definition 3

Given $f(n)$, we denote by $\Omega(g(n))$ the set of functions:

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

Ω-Notation (cont.)



- For all values of n to the right of n_0, the value of $f(n)$ is on or above $cg(n)$
- We say that $g(n)$ is a **lower bound** for $f(n)$

Ω-Notation: An Example

Claim

$T(n) = pn^2 + qn + r$ is in $\Omega(n^2)$.

$T(n) = pn^2 + qn + r \geq pn^2$ for all $n \geq 0$.

This is the required definition of $\Omega(\cdot)$:
$T(n) \geq cn^2$, where $c = p$.

Asymptotically Tight Bounds: Θ-Notation

If a running time $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$, then we've found a "tight" bound.

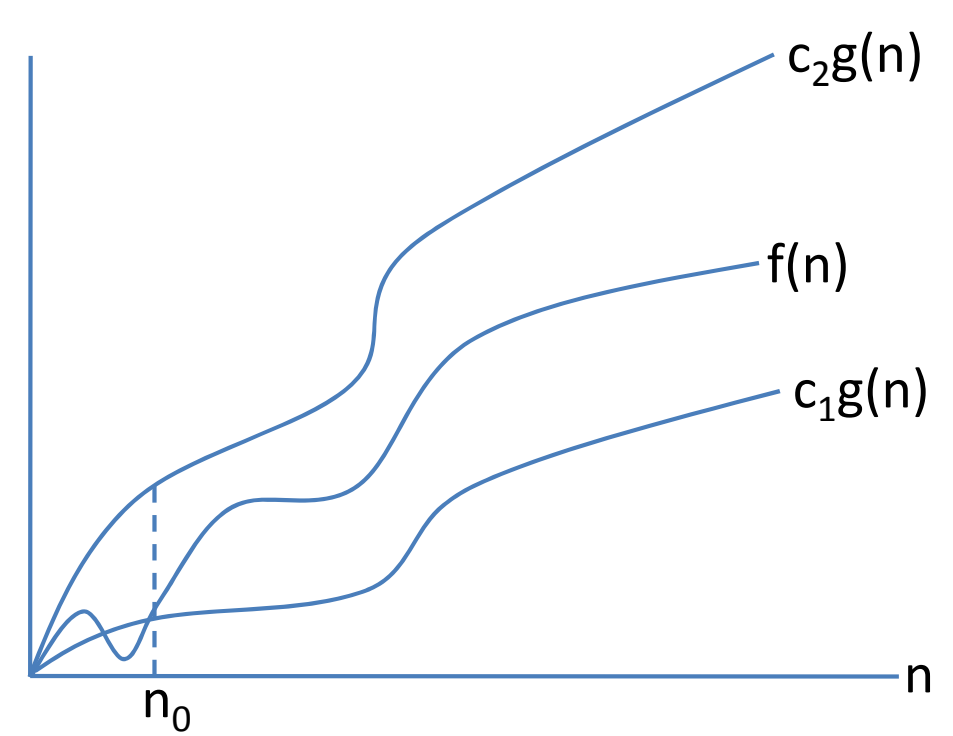
Definition 4

Given $g(n)$, we denote by $\Theta(g(n))$ the set of functions:

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

- Effectively, f is "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$ for large n

Θ-Notation (cont.)



- For all values of n to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$.
- We say that $g(n)$ is an **asymptotically tight bound** for $f(n)$

Θ-Notation: An Example

Claim

Consider $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

- We must show that there exists c_1, c_2, and n_0 such that
$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$
for all $n \geq n_0$
- This is equivalent to showing
$$c1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$
for all $n \geq n_0$
- Take $c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$ and $n_0 = 7$

There are many other viable choices for c_1, c_2, and n_0. And for other functions in $\Theta(n^2)$ we may need different c_1, c_2, and n_0.

Θ-Notation: Another Example

Claim

$6n^3 \neq \Theta(n^2)$

- Were this the case, then there exist c_1, c_2, and n_0 such that $6n^3 \leq c_2 n^2$ for all $n \geq n_0$.
- Equivalently, this means that, for all $n \geq n_0$, $n \leq \frac{c_2}{6}$.
- Since c_2 is a constant, and n is not, this is impossible!

In English...

Upper Bound

$f(n) = O(g(n))$:

- $f(n)$ is bounded above by a constant multiple of $g(n)$.
- $f(n)$ is asymptotically upper bounded by $g(n)$.

Lower Bound

$f(n) = \Omega(g(n))$

- $f(n)$ is at least a constant multiple of $g(n)$.
- $f(n)$ is asymptotically lower bounded by $g(n)$.

Questions?