Computational problems

- A computational problem specifies an input-output relationship
  - What does the input look like?
  - What should the output be for each input?
- Example:
  - Input: an integer number N
  - Output: Is the number prime?
- Example:
  - Input: A list of names of people
  - Output: The same list sorted alphabetically
- Example:
  - Input: A picture in digital format
  - Output: An English description of what the picture shows

Algorithms

- An algorithm is an exact specification of how to solve a computational problem
- An algorithm must specify every step completely, so a computer can implement it without any further “understanding”
- An algorithm must work for all possible inputs of the problem.
- Algorithms must be:
  - Correct: For each input produce an appropriate output
  - Efficient: run as quickly as possible, and use as little memory as possible – more about this later
- There can be many different algorithms for each computational problem.

Describing Algorithms

- Algorithms can be implemented in any programming language
- Usually we use “pseudo-code” to describe algorithms

```
Testing whether input N is prime:

For j = 2 .. N-1
  If j|N
    Output “N is composite” and halt
Output “N is prime”
```

Greatest Common Divisor

- The first algorithm “invented” in history was Euclid’s algorithm for finding the greatest common divisor (GCD) of two natural numbers
- **Definition:** The GCD of two natural numbers x, y is the largest integer j that divides both (without remainder). I.e. $j|x$, $j|y$ and j is the largest integer with this property.
- **The GCD Problem:**
  - Input: natural numbers x, y
  - Output: $GCD(x,y)$ – their GCD

Euclid's GCD Algorithm

```
public static int gcd(int x, int y) {
  while (y!=0) {
    int temp = x%y;
    x = y;
    y = temp;
  }
  return x;
}
```

Euclid’s GCD Algorithm – sample run

```
while (y!=0) {
  int temp = x%y;
  x = y;
  y = temp;
}
```

Example: Computing GCD(48,120)

	temp	x	y
After 0 rounds	--	72	120
After 1 round	72	120	72
After 2 rounds	48	72	48
After 3 rounds	24	48	24
After 4 rounds	0	24	0

Output: 24

Correctness of Euclid's Algorithm

- Theorem: When Euclid's GCD algorithm terminates, it returns the mathematical GCD of x and y.
- Notation: Let g be the GCD of the original values of x and y.
- Loop Invariant Lemma: For all $k \geq 0$, The values of x, y after k rounds of the loop satisfy $GCD(x,y)=g$.
- Proof of lemma: ?
- Proof of Theorem: The method returns when $y=0$. By the loop invariant lemma, at this point $GCD(x,y)=g$. But $GCD(x,0)=x$ for every integer x (since $x|0$ and $x|x$). Thus $g=x$, which is the value returned by the code.
- Still Missing: The algorithm always terminates.

Proof of Lemma

- <u>Loop Invariant Lemma:</u> For all $k \geq 0$, The values of *x, y* after *k* rounds of the loop satisfy $GCD(x,y)=g$.
- <u>Proof:</u> By induction on *k*.
  - For $k=0$, *x* and *y* are the original values so clearly $GCD(x,y)=g$.
  - Induction step: Let x, y denote that values after k rounds and *x'*, *y'* denote the values after $k+1$ rounds. We need to show that $GCD(x,y)=GCD(x',y')$. According to the code: $x'=y$ and $y'=x\%y$, so the lemma follows from the following mathematical lemma.
- <u>Lemma:</u> For all integers *x, y*: $GCD(x, y) = GCD(x\%y, y)$
- <u>Proof:</u> Let $x=ay+b$, where $y>b\geq 0$. I.e. $x\%y=b$.
  - *(1)* Since $g|y$, and $g|x$, we also have $g|(x-ay)$, I.e. $g|b$. Thus $GCD(b,y) \geq g = GCD(x,y)$.
  - *(2)* Let $g'=GCD(b,y)$, then $g'|(x-ay)$ and $g'|y$, so we also have $g'|x$. *Thus* $GCD(x,y) \geq g'=GCD(b,y)$.

Termination of Euclid's Algorithm

- Why does this algorithm terminate?
  - After any iteration we have that $x > y$ since the new value of y is the remainder of division by the new value of x.
  - In further iterations, we replace (x, y) with $(y, x\%y)$, and $x\%y < x$, thus the numbers decrease in each iteration.
  - Formally, the value of xy decreases each iteration (except, maybe, the first one). When it reaches 0, the algorithm must terminate.

```
public static int gcd(int x, int y) {
  while (y!=0) {
    int temp = x%y;
    x = y;
    y = temp;
  }
  return x;
}
```

Square Roots

- The problem we want to address is to compute the square root of a real number.
- When working with real numbers, we can not have complete precision.
  - The inputs will be given in finite precision
  - The outputs should only be computed approximately
- The square root problem:
  - Input: a positive real number x, and a precision requirement ε
  - Output: a real number r such that $|r-\sqrt{x}|\leq\varepsilon$

Square Root Algorithm

```
public static double sqrt(double x, double epsilon){
  double low = 0;
  double high = x>1 ? x : 1;
  while (high-low > epsilon) {
    double mid = (high+low)/2;
    if (mid*mid > x)
       high = mid;
    else
       low = mid;
  }
  return low;
}
```

Binary Search Algorithm – sample run

```
while (high-low > epsilon) {
  double mid = (high+low)/2;
  if (mid*mid > x)
    high = mid;
  else
    low = mid;
}
```

Example: Computing sqrt(2) with precision 0.05:

	mid	mid*mid	low	high
After 0 rounds	--	--	0	2
After 1 round	1	1	1	2
After 2 rounds	1.5	2.25	1	1.5
After 3 rounds	1.25	1.56..	1.25	1.5
After 4 rounds	1.37..	1.89..	1.37..	1.5
After 5 rounds	1.43..	2.06..	1.37..	1.43..
After 6 rounds	1.40..	1.97..	1.40..	1.43..

Output: 1.40…

Correctness of Binary Search Algorithm

- Theorem: When the algorithm terminates it returns a value *r* that satisfies $|r-\sqrt{x}|\leq\varepsilon$.
- Loop invariant lemma: For all $k \geq 0$, The values of *low, high* after *k* rounds of the loop satisfy: $low \leq \sqrt{x} \leq high$.
- Proof of Lemma:
  - For k=0, clearly low=0 $\leq \sqrt{x} \leq$ $high=max(x,1)$.
  - Induction step: The code only sets *low=mid* if mid $\leq \sqrt{x}$, and only sets *high=mid* if mid$>\sqrt{x}$.
- Proof of Theorem: The algorithm terminates when $high\text{-}low\leq\varepsilon$, and returns *low.* At this point, by the lemma: $low \leq \sqrt{x} \leq high \leq low+\varepsilon$. Thus $|low-\sqrt{x}|\leq\varepsilon$.
- Missing Part: Does the algorithm always terminate? How Fast? We will deal with this later.

How fast will your program run?

- The running time of your program will depend upon:
  - The algorithm
  - The input
  - Your implementation of the algorithm in a programming language
  - The compiler you use
  - The OS on your computer
  - Your computer hardware
  - Maybe other things: other programs on your computer; …
- <u>Our Motivation:</u> analyze the running time of an algorithm as a function of only simple parameters of the input.

Basic idea: counting operations

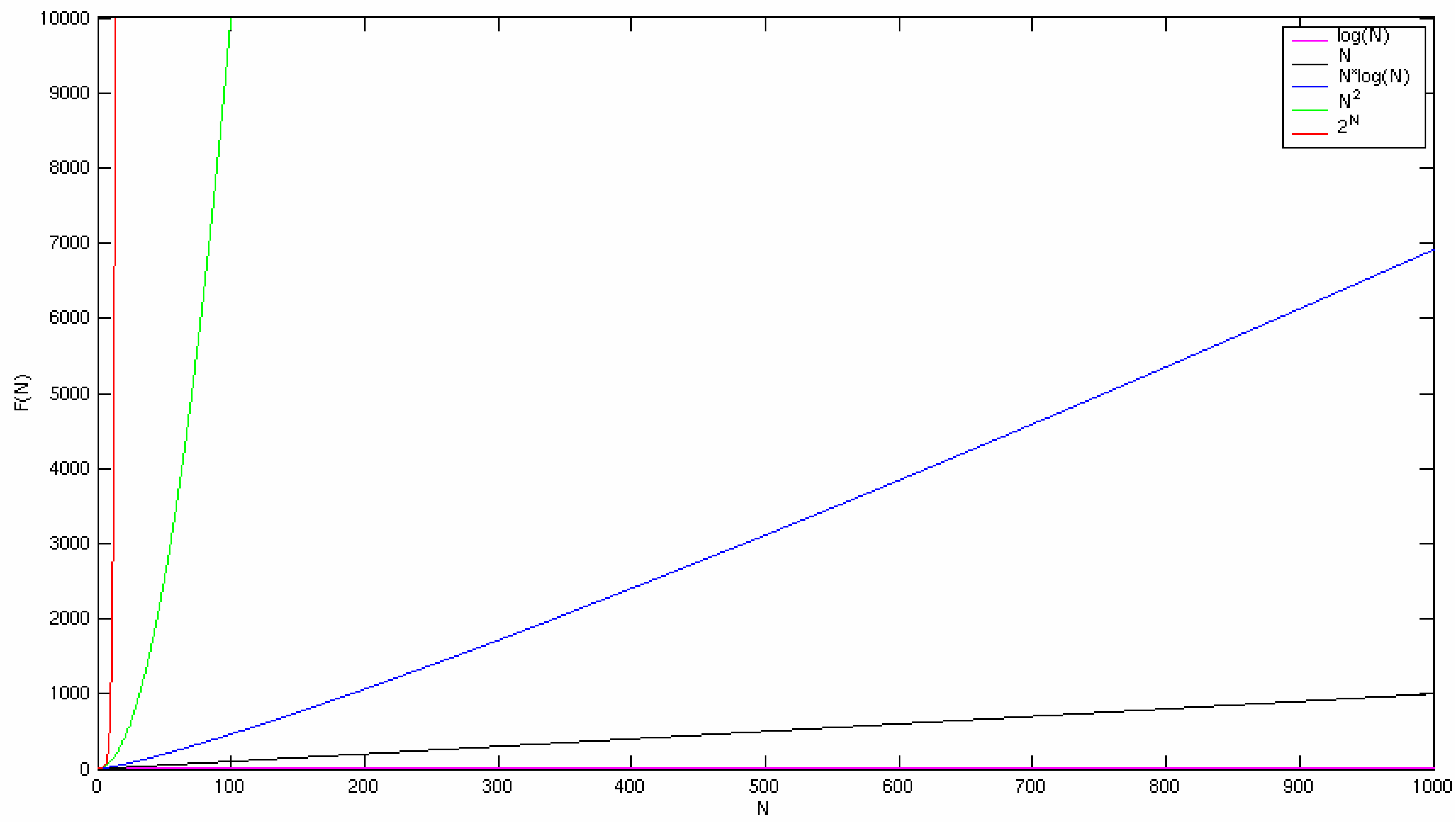
- Each algorithm performs a sequence of basic operations:
  - Arithmetic: (low + high)/2
  - Comparison: if (x > 0) …
  - Assignment: temp = x
  - Branching: while (true) { … }
  - …
- Idea: count the number of basic operations performed on the input.
- Difficulties:
  - Which operations are basic?
  - Not all operations take the same amount of time.
  - Operations take different times with different hardware or compilers

Asymptotic running times

- Operation counts are only problematic in terms of constant factors.
- The general form of the function describing the running time is invariant over hardware, languages or compilers!
- Running time is "about" *.*
- We use "Big-O" notation, and say that the running time is $O(N^2)$

```java
public static int myMethod(int N){
  int sq = 0;
  for(int j=0; j<N ; j++)
    for(int k=0; k<N ; k++)
      sq++;
  return sq;
}
```

Asymptotic behavior of functions



Mathematical Formalization

- Definition: Let *f* and *g* be functions from the natural numbers to the natural numbers. We write $f=O(g)$ if there exists a constant c such that for all n: $f(n) \leq cg(n)$.

$$f=O(g) \quad \Leftrightarrow \quad \exists c \forall n: f(n) \leq cg(n)$$

- This is a mathematically formal way of ignoring constant factors, and looking only at the “shape” of the function.
- $f=O(g)$ should be considered as saying that “*f* is at most *g*, up to constant factors”.
- We usually will have *f* be the running time of an algorithm and *g* a nicely written function. E.g. The running time of the previous algorithm was $O(N^2)$.

Asymptotic analysis of algorithms

- We usually embark on an *asymptotic worst case* analysis of the running time of the algorithm.
- Asymptotic:
  - Formal, exact, depends only on the algorithm
  - Ignores constants
  - Applicable mostly for large input sizes
- Worst Case:
  - Bounds on running time must hold for *all* inputs.
  - Thus the analysis considers the worst-case input.
  - Sometimes the “average” performance can be much better
  - Real-life inputs are rarely “average” in any formal sense

The running time of Euclid's GCD Algorithm

- How fast does Euclid's algorithm terminate?
  - After the first iteration we have that $x > y$. In each iteration, we replace (x, y) with $(y, x\%y)$.
  - In an iteration where $x>1.5y$ then $x\%y < y < 2x/3$.
  - In an iteration where $x \leq 1.5y$ then $x\%y \leq y/2 < 2x/3$.
  - Thus, the value of xy decreases by a factor of at least $2/3$ each iteration (except, maybe, the first one).

```
public static int gcd(int x, int y) {
  while (y!=0) {
    int temp = x%y;
    x = y;
    y = temp;
  }
  return x;
}
```

The running time of Euclid's Algorithm

- <u>Theorem:</u> Euclid's GCD algorithm runs it time *O(N),* where N is the input length *($N=\log_2 x + \log_2 y$).*
- <u>Proof:</u>
  - Every iteration of the loop (except maybe the first) the value of xy decreases by a factor of at least 2/3. Thus after *k+1* iterations the value of *xy* is at most $(2/3)^k$ the original value.
  - Thus the algorithm must terminate when k satisfies: $xy(2/3)^k < 1$
  - (for the original values of x, y).
  - Thus the algorithm runs for at most $1+\log_{3/2} xy$ iterations.
  - Each iteration has only a constant ***L*** number of operations, thus the total number of operations is at most $(1+\log_{3/2} xy)L$
  - Formally,
  - Thus the running time is *O(N).*

$$(1+\log_{3/2} xy)L \leq L(1+2\log_2 x + 2\log_2 y) \leq 3LN$$

Algorithms and Problems

Algorithm: a method or a process followed to solve a problem.

– A recipe.

A problem is a mapping of input to output.

An algorithm takes the input to a problem (function) and transforms it to the output.

A problem can have many algorithms.

Algorithm Properties

An algorithm possesses the following properties:

- It must be correct.
- It must be composed of a series of concrete steps.
- There can be no ambiguity as to which step will be performed next.
- It must be composed of a finite number of steps.
- It must terminate.

A computer program is an instance, or concrete representation, for an algorithm in some programming language.

How fast is an algorithm?

- To compare two sorting algorithms, should we talk about how fast the algorithms can sort 10 numbers, 100 numbers or 1000 numbers?

- We need a way to talk about how fast the algorithm *grows* or *scales* with the input size.

  – Input size is usually called n
  – An algorithm can take $100n$ steps, or $2n^2$ steps, which one is better?

Introduction to Asymptotic Notation

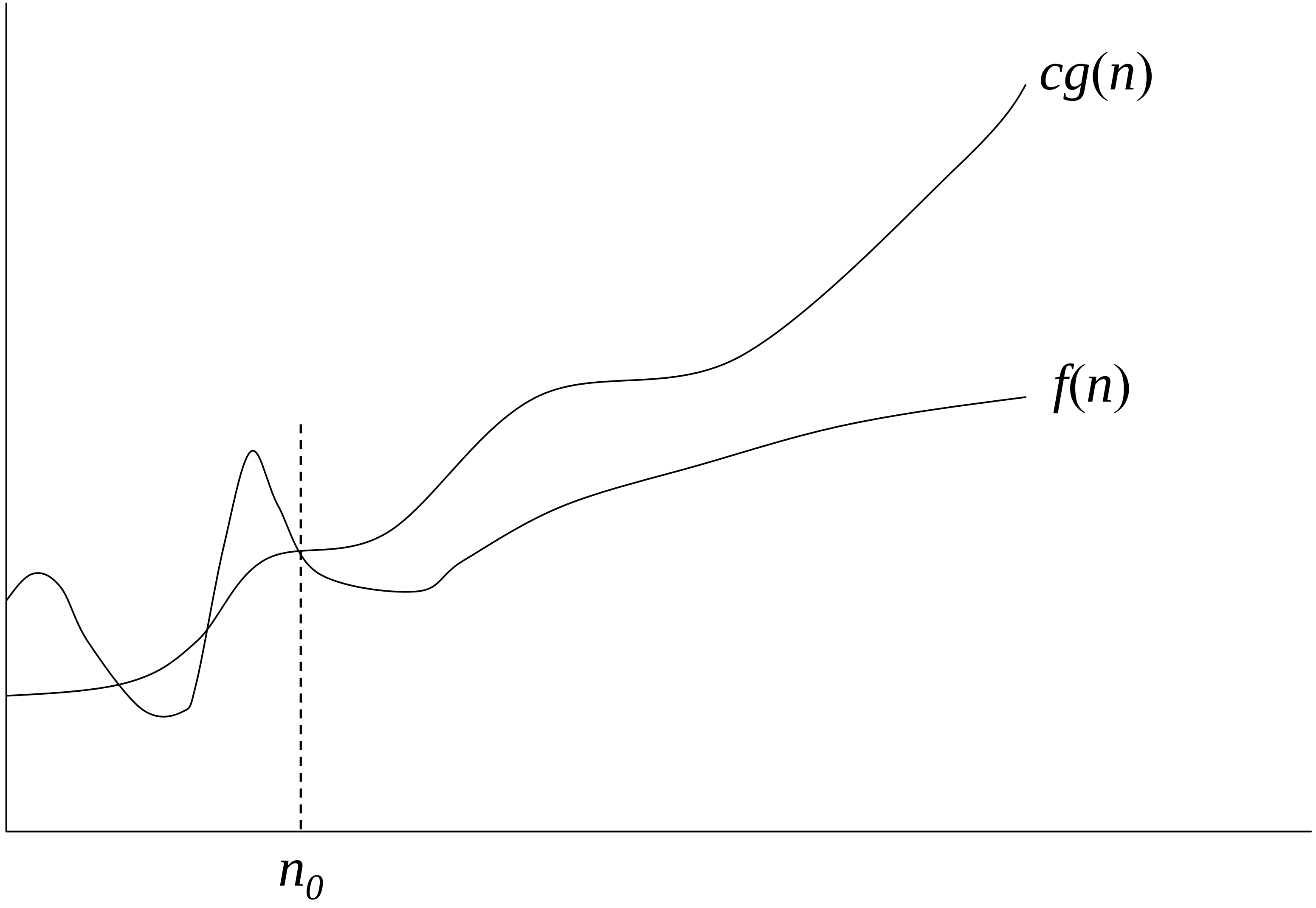
- We want to express the concept of "about", but in a mathematically rigorous way
- Limits are useful in proofs and performance analyses
- Θ notation: $\Theta(n^2)$ = "this function grows similarly to n^2".
- Big-O notation: O (n^2) = "this function grows at least as *slowly* as n^2".
  - Describes an *upper bound.*

Big-O

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

- What does it mean?
  - If $f(n) = O(n^2)$, then:
    - $f(n)$ can be larger than n^2 sometimes, **but…**
    - I can choose some constant c and some value n_0 such that for **every** value of n larger than n_0 : $f(n) < cn^2$
    - That is, for values larger than n_0, $f(n)$ is never more than a constant multiplier greater than n^2
    - Or, in other words, $f(n)$ does not grow more than a constant factor faster than n^2.

Visualization of $O(g(n))$



Big-O

$$2n^2 = O\left(n^2\right)$$

$$1{,}000{,}000\,n^2 + 150{,}000 = O\left(n^2\right)$$

$$5n^2 + 7n + 20 = O\left(n^2\right)$$

$$2n^3 + 2 \neq O\left(n^2\right)$$

$$n^{2.1} \neq O\left(n^2\right)$$

More Big-O

$$20n^2 + 2n + 5 = O\left(n^2\right)$$

- Prove that:
- Let $c = 21$ and $n_0 = 4$
- $21n^2 > 20n^2 + 2n + 5$ for all $n > 4$

  $n^2 > 2n + 5$ for all $n > 4$

  TRUE

Tight bounds

- We generally want the tightest bound we can find.
- While it is true that $n^2 + 7n$ is in O(n^3), it is more interesting to say that it is in O(n^2)

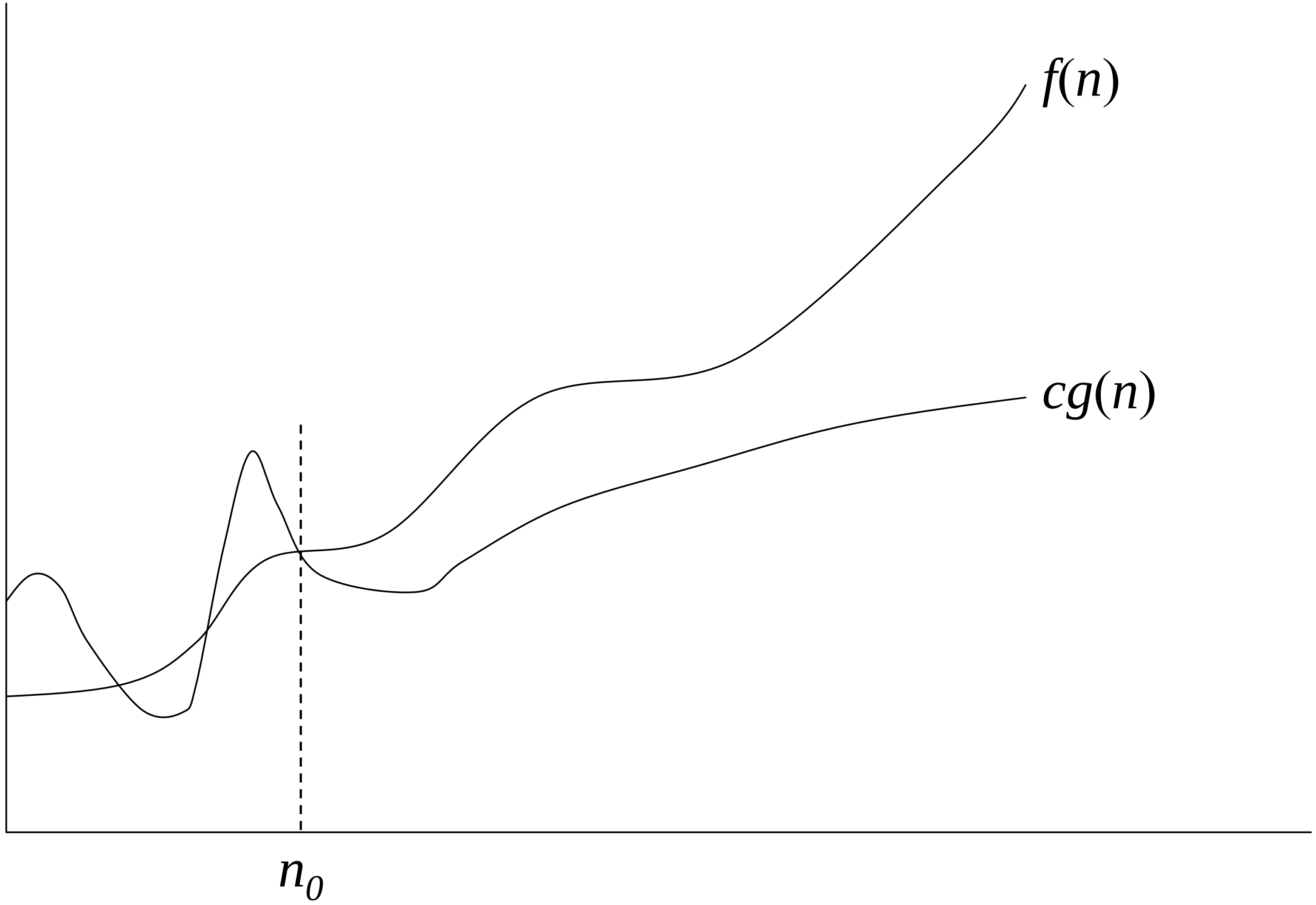
Big Omega – Notation

- $\Omega()$ – A **lower** bound

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

- $n^2 = \Omega(n)$
- Let $c = 1$, $n_0 = 2$
- For all $n \ge 2$, $n^2 > 1 \times n$

Visualization of $\Omega(g(n))$



Θ-notation

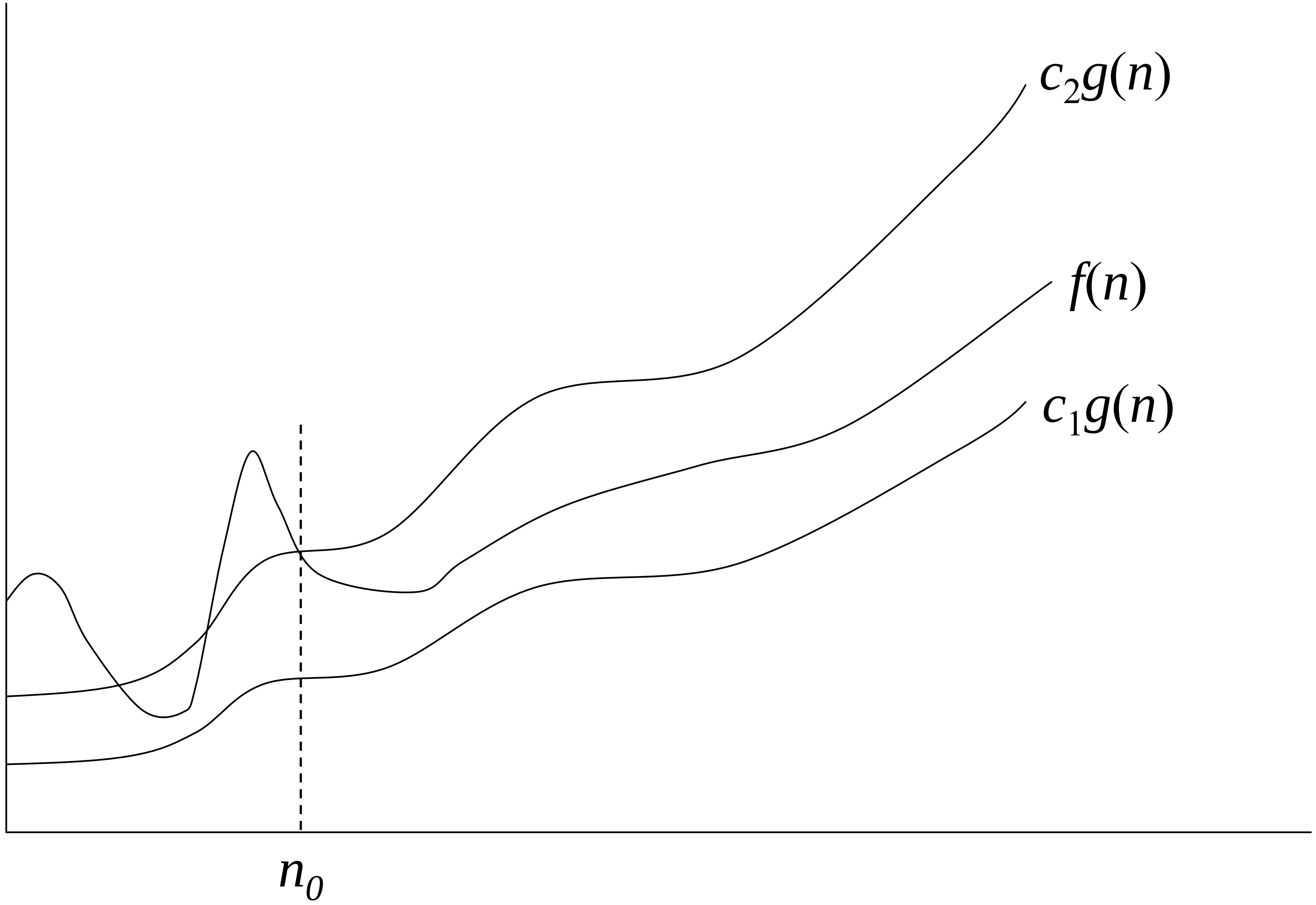
- Big-*O* is not a tight upper bound. In other words $n = O(n^2)$
- Θ provides a tight bound

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

- In other words,

$$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n)) \text{ AND } f(n) = \Omega(g(n))$$

Visualization of Θ(*g*(*n*))



A Few More Examples

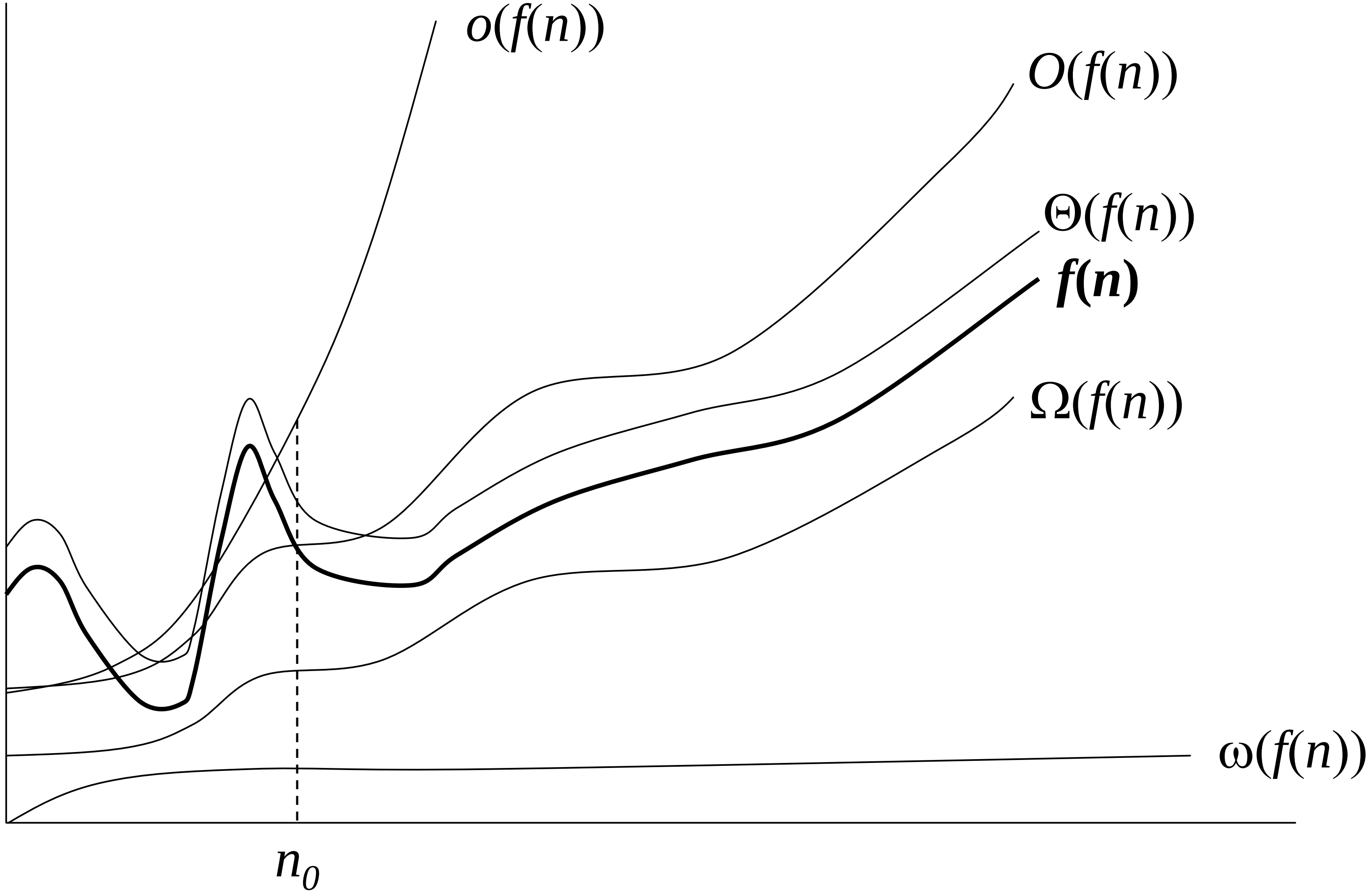
- $n = O(n^2) \neq \Theta(n^2)$
- $200n^2 = O(n^2) = \Theta(n^2)$
- $n^{2.5} \neq O(n^2) \neq \Theta(n^2)$

Some Other Asymptotic Functions

- Little *o* – A **non-tight** asymptotic upper bound
  - $n = o(n^2)$, $n = O(n^2)$
  - $3n^2 \neq o(n^2)$, $3n^2 = O(n^2)$
- $\Omega()$ – A **lower** bound
  - Similar definition to Big-O
  - $n^2 = \Omega(n)$
- $\omega()$ – A **non-tight** asymptotic lower bound

- $f(n) = \Theta(n) \Leftrightarrow f(n) = O(n)$ **and** $f(n) = \Omega(n)$

Visualization of Asymptotic Growth



Analogy to Arithmetic Operators

$$f(n) = O(g(n)) \quad \approx \quad a \le b$$

$$f(n) = \Omega(g(n)) \quad \approx \quad a \ge b$$

$$f(n) = \Theta(g(n)) \quad \approx \quad a = b$$

$$f(n) = o(g(n)) \quad \approx \quad a < b$$

$$f(n) = \omega(g(n)) \quad \approx \quad a > b$$

Example

$$20n^3 + 7n + 1000 = \Theta\left(n^3\right)$$

- Prove that:
- Let $c = 21$ and $n_0 = 10$
- $21n^3 > 20n^3 + 7n + 1000$ for all $n > 10$

  $n^3 > 7n + 5$ for all $n > 10$

  TRUE, but we also need…
- Let $c = 20$ and $n_0 = 1$
- $20n^3 < 20n^3 + 7n + 1000$ for all $n \geq 1$

  TRUE

Looking at Algorithms

- Asymptotic notation gives us a language to talk about the run time of algorithms.
- Not for just one case, but how an algorithm performs as the size of the input, *n*, grows.
- Tools:
  - Series sums
  - Recurrence relations

Running Time Example

Example 1: `a = b;`

This assignment takes constant time, so it is $\Theta(1)$.

Example 2:

```
sum = 0;
for (i=1; i<=n; i++)
  sum += n;
```

Space Bounds

Space bounds can also be analyzed with asymptotic complexity analysis.

Time: Algorithm

Space: Data Structure

Space/Time Tradeoff Principle

One can often reduce time if one is willing to sacrifice space, or vice versa.

- Encoding or packing information
  - Boolean flags
- Table lookup
  - Factorials

Disk-based Space/Time Tradeoff Principle: The smaller you make the disk storage requirements, the faster your program will run.

Growth of Functions

n	1	lgn	n	nlgn	n^2	n^3	2^n
1	1	0.00	1	0	1	1	2
10	1	3.32	10	33	100	1,000	1024
100	1	6.64	100	664	10,000	1,000,000	1.2×10^{30}
1000	1	9.97	1000	9970	1,000,000	10^9	1.1×10^{301}

Is there a "real" difference?

-