CS 114 Math Preliminaries (Contd.)

Proof techniques

1. Direct proofs
2. Contra-positive proofs
3. Proof by contradiction
4. Proof by induction

Contra-positive proof

- To prove A → B, we prove ¬ B → ¬ A as (A → B) ↔ (¬ B → ¬ A) (they are logically equivalent)
- **Example** : For $x \in \mathbb{Z}$, if x^2 - 6 x + 5 is even then x must be odd.
- Easier to prove x is even → x^2 - 6 x + 5 is odd

Proof by Contradiction

- Suppose we need to prove $A \rightarrow B$.
- We prove $A \wedge \neg B \rightarrow$ False

i.e. Assuming $A \wedge \neg B$ we prove some absurdity.

This implies what we assumed is false.

- Note it is NOT contra-positive proof !!

Proof by contradiction - example

- Prove that $\sqrt{2}$ is an irrational number.

--Assume it is a rational number $\frac{a}{b}$ where a and b have no common factors.

$\rightarrow \frac{a}{b} = \sqrt{2} \rightarrow a^2 = 2\,b^2 \rightarrow a^2$ is even $\rightarrow$ a is even

$\rightarrow$ b is odd (a and b have no common factors) – (1)

a is even $\rightarrow 2\,b^2$ is a multiple of 4 $\rightarrow b^2$ is even $\rightarrow$ b is even -- (2)

(1) & (2) – absurdity.

Hence $\sqrt{2}$ cannot be a rational number.

Proof by induction

- We want to prove a predicate p(x) for all x ∈ S where S is a countable (typically infinite) set with some total order ≤ on elements of S
- **Idea** : Enumerate elements of S with respect to total order $a_1 \leq a_2 \leq \ldots\ldots$

1. Prove P(a_1) where a_1 is least element of S – **Basis step**
2. **Induction step** : P(a_m) → P(a_{m+1}) for all m ≥ 1

Examples of induction

Prove $\sum_{i=1}^{n} i^2 = n\frac{(n+1)(2n+1)}{6}$ for all n ≥ 1

Basis step (n= 1) : $\sum_{i=1}^{1} i^2 = 1$.

RHS = 1(1 + 1)(2* 1 + 1)/6 = 1

Induction step : Assume true for m ≥ 1. Prove for m+1.

i.e. Assume $\sum_{i=1}^{m} i^2 = m\frac{(m+1)(2m+1)}{6}$

Then $\sum_{i=1}^{m+1} i^2 = m\frac{(m+1)(2m+1)}{6} + (m+1)^2$

$= (m+1)\,[m\,(2m+1)/6 + (m+1)]$

Can be shown to be $(m+1)\frac{(m+2)(2m+3)}{6}$

Examples of induction

- Consider the recurrence

  T (0) = 0

  T(n) = 2n− 1 +T(n− 1) for n > 0.

Show by induction that T (n) = n^2 for n ≥ 0.

Basis : n = 0, T(0) = 0 = 0^2 , done

Induction step : Assume true for n ≥ 0.

Show for n+ 1.

T(n+ 1) = 2 (n+ 1) – 1 +T(n) = 2n+ 1 + n^2 = $(n + 1)^2$

Strong induction

- For induction step we prove :

$P(a_1) \wedge P(a_2) \wedge P(a_3) \wedge .. P(a_m) \rightarrow P(a_{m+1})$

Example: Every integer > 1 can be written as a product of primes.

Basis step : n = 2. It is obvious

Induction step : Assume true for all integers $\leq$ m, prove for m+ 1.

Case a : m+ 1 is prime, we are done

Case b : m+ 1 is not prime. Then m+ 1 = a * b where a > 1 and b > 1 and a, b $\leq$ m . Then by induction hypothesis, a and b can be written as products of primes.

$\rightarrow$ m+ 1 can be written as product of primes.

Random Access Machine

- Abstract computational model for time/space complexities
- Infinite memory and unlimited word size
- Uniform (independent of word size) vs logarithmic time complexity (proportional to logarithm of word size) models
- We will use uniform complexity model in algorithm analysis (visit logarithmic model later during TM discussion)
- Unit times for primitive operations and unit space for a word

RAM time complexity

- Primitive operations
  1. variable assignment
  2. method call/return statement
  3. arithmetic and comparison operations
  4. array indexing
  5. deference an object pointer (follow object reference)
- Total operation count proportional to running time

RAM space complexity

- Unit value for each primitive data item like numbers regardless of its size
- Space complexity is measured in terms of maximum space required at any stage of execution of the algorithm including the inputs and outputs.

BinarySearch Example

```
public static int binarySearch(int [] A, int e)
{
    int low = 0, high = A.length - 1;
    while (low < high) {
        int mid = (int) Math.floor((low + high) /
2);
        if (e == A[mid]) {
            return mid;
        } else if (e < A[mid]) {
            high = mid;
        } else {
            low = mid + 1;
        }
    }
    return -1;
}
```

BinarySearch Example

1. Initialization – 2 units
2. In each while loop:

   a. loop condition – 1 unit

   b. mid value calc (addition + floor func), assignment – 3 units

   c. 1 unit for indexing mid

   d. if element not found – 2 units for comparisons + 1 unit for low/high assignment

   e. if element found – 1 unit for comparison

3. 1 unit for return

- Total time (best case) – 2 + 1 + 3 + 1 + 1 + 1 = 9
- Total time (worst case – element not found) – 2 + (m+1) + m *(3+1+3) + 1 = 3 + 8m

  (where m is number of iterations in the worst case)

Later we will show m <= [log n]

Asymptotic Growth Rates

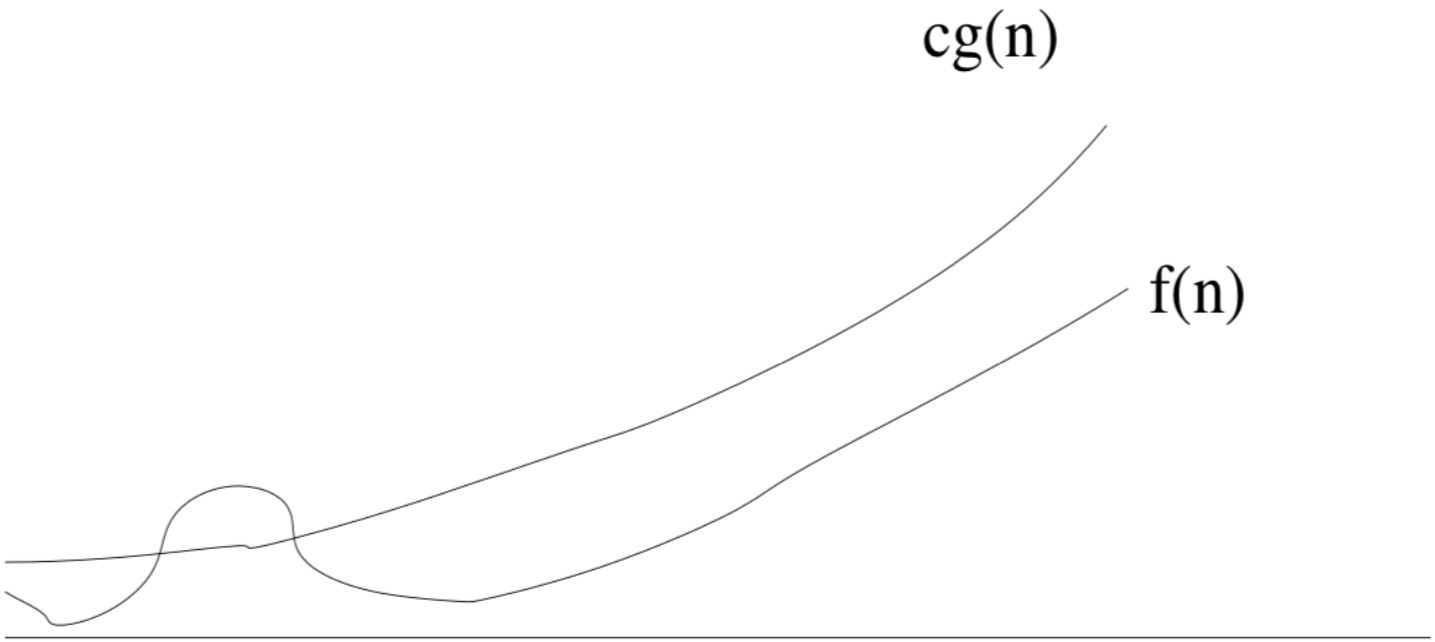
- An algorithm may perform well for small input sizes but may not scale up for large input sizes.
- We need a time-complexity function definition that ignores constant factors and measures how well it grows for large input sizes
- Asymptotic growth functional notation provides mathematical framework to capture these ideas.

Big-Oh notation

- A function f(n) : $\mathbb{N}$ -> R^+ is O(g(n)) if there exists a real constant c > 0 and integer $n_0 > 0$ such that for all $n \geq n_0$, $f(n) \leq c\ g(n)$
- Note that O(g(n)) denotes a set of functions (here we mean f(n) ϵ O(g(n))
- f(n) bounded from above by cg(n) asymptotically but may not be strictly.
- What is O(1) ?

  Constant value function -- f(n) = d
- Examples:
- BinarySearch algorithm has time-complexity O(log n)

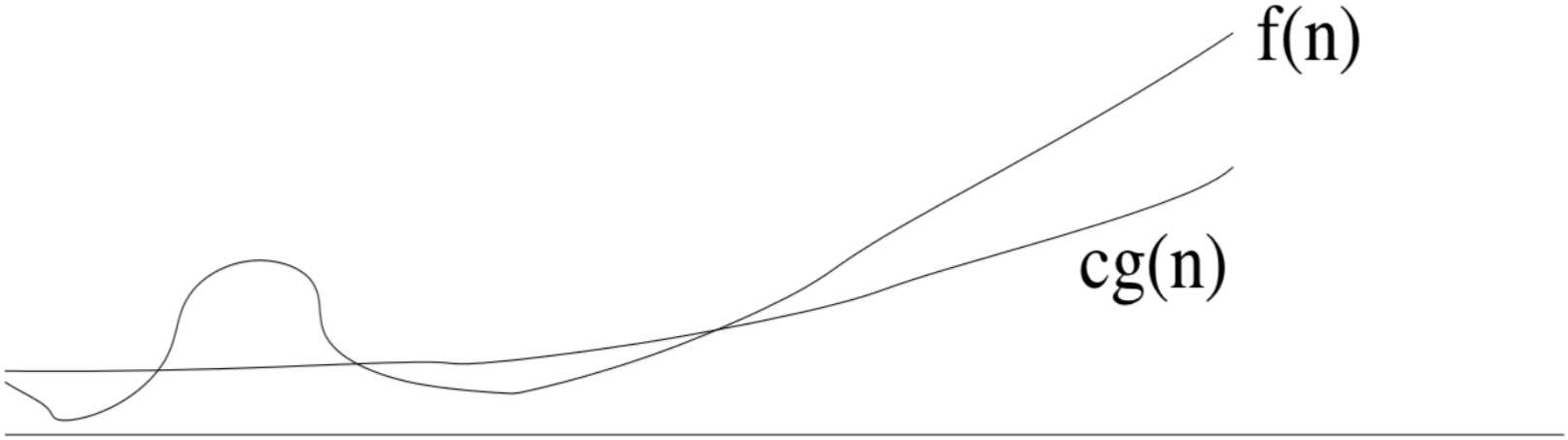
f(n) is O(g(n))



Big-Omega notation

- A function f(n) : $\mathbb{N}$ -> R^+ is $\Omega(g(n))$ if there exists a real constant c > 0 and integer $n_0 > 0$ such that for all $n \geq n_0$, $f(n) \geq c\ g(n)$
- Note that $\Omega(g(n))$ denotes a set of functions
- f(n) bounded from below by c.g(n) asymptotically but may not be strictly.
- f(n) is $\Omega(g(n))$ iff g(n) is O(f(n))
- Examples:
- BinarySearch algorithm has $\Omega(1)$ time-complexity

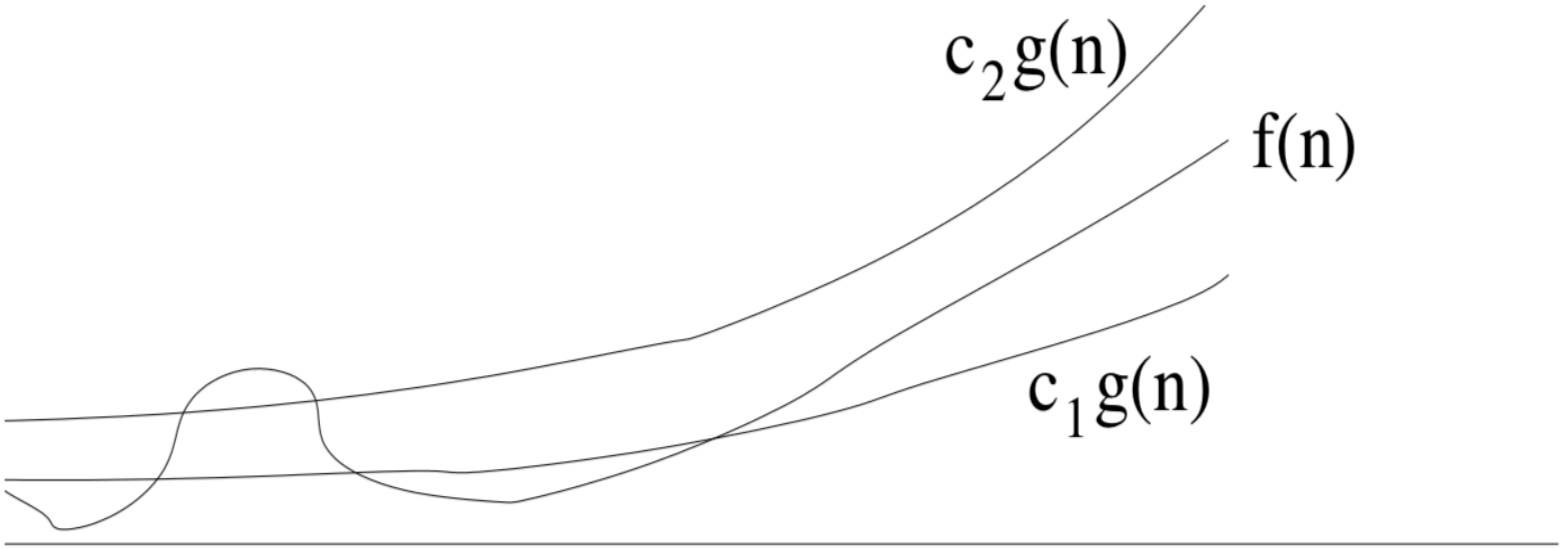
f(n) is Ω(g(n))



Big-Theta notation

- A function f(n) : $\mathbb{N}$ -> R^+ is $\Theta(g(n))$ if there exist +ve real constants c 1 and c2 and integer $n_0 > 0$ such that for all $n \geq n_0$, c1 g(n) ≤ f(n) ≤ c2 g(n)
- Note that $\Theta(g(n))$ denotes a set of functions
- Theta bound is said to be a tight bound
- For algorithm complexity, we always attempt to get a tight bound though we use big-oh.
- f(n) is $\Theta(g(n))$ iff g(n) is $\Theta(f(n))$
- Examples:
- BinarySearch algorithm has $\Theta(\log n)$ worst-case time-complexity

f(n) is Θ(g(n))



Significance of big-oh

Big-oh allows algorithm comparisons as input sizes scale-up

Running Time	Maximum Problem Size (n)		
	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$20n\lceil \log n \rceil$	4,096	166,666	7,826,087
$2n^2$	707	5,477	42,426
n^4	31	88	244
2^n	19	25	31

Significance of big-oh (contd.)

Hardware speed-up does not improve algorithm's

asymptotic time complexity

(256 times faster processor)

Running Time	New Maximum Problem Size
$400n$	$256m$
$20n\lceil \log n \rceil$	approx. $256((\log m)/(7+\log m))m$
$2n^2$	$16m$
n^4	$4m$
2^n	$m+8$

Showing bounds for f(n)

When comparing the order of growth of functions, there are a few tricks that may come in handy. If the ratio $f(n)/g(n)$ converges to a limit as $n \to \infty$ (this will almost always be the case in our examples) then we can use a simpler test: If

$$\frac{f(n)}{g(n)} \to c \in [0, \infty]$$

as $n \to \infty$, then:

- if $c < \infty$, then $f(n) = O(g(n))$.
- if $c > 0$, then $f(n) = \Omega(g(n))$.
- if $0 < c < \infty$, then $f(n) = \Theta(g(n))$.

Applying this idea, let's compare $n(3/2)^n$ and 2^n.

Taking the ratio, we have

$$\frac{n(3/2)^n}{2^n} = n\left(\frac{3}{4}\right)^n \to 0,$$

and so we conclude that $n(3/2)^n = O(2^n)$.

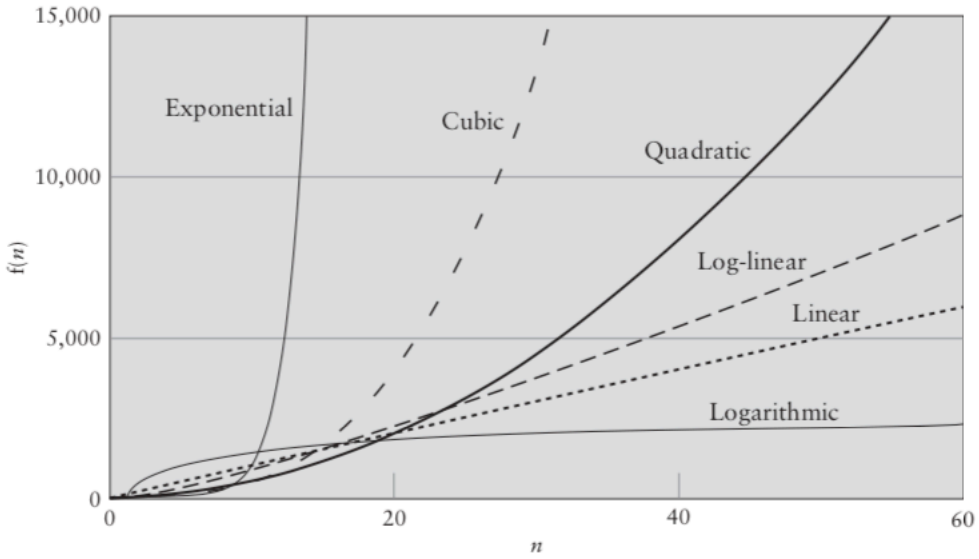
Another trick is to apply an increasing function to the ratio; often the logarithm function is useful. Since the logarithm is positive, increasing, and $\lg(x) \to \infty$ as $x \to \infty$, we know that if $\lg(f(n)/g(n))$ goes to ∞, then so does $f(n)/g(n)$ (and if the $\lg$ of the ratio converges to a finite constant, then so does the ratio).

Asymptotic grow rates

TABLE 2.2
Common Growth Rates

Big-O	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

FIGURE 2.3
Different Growth Rates



Simplifying Rules

- If $f(n)$ is in $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.
- If $f(n)$ is $O(kg(n))$ for some constant $k > 0$, then $f(n)$ is $O(g(n))$.
- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $(f_1 + f_2)(n)$ is $O(\max(g_1(n), g_2(n)))$.
- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g2(n))$ then $f_1(n)f_2(n)$ is $O(g_1(n)g_2(n))$.