Design and Analysis of Algorithms

Algorithm Analysis

Even Semester- 2016-17

Course Details

- Lecture Notes – on AUMS
- Text Book
  - Michael T Goodrich, Roberto Tamassia, “Algorithm Design: Foundations, Analysis and Internet Examples”, John Wiley and Sons, 2001
  - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, “Introduction to Algorithms, Second Edition”, The MIT Press, 2001

Evaluation

- Grade Policy
  - Final – 50%
  - Midterm – 30%
  - Assignments/Quizzes/Tutorials – 20%
    - One after each topic
    - Quizzes/Tutorials 5 minimum
    - Programming assignments
      - Implement important algorithms and apply them for problem solving

Lecture Schedule (Tentative)

- Term I (before periodical I)
  - Algorithm Analysis
  - Sorting Algorithms
  - Greedy Algorithms
  - Recurrence Analysis
- Term II (before periodical II)
  - Divide and Conquer
  - Dynamic Programming
  - Backtracking and Branch and Bound

Lecture Schedule

- Term III
  - String Algorithms
  - Graph Algorithms
  - Introduction to NP Completeness
- May be modified over time

Problem Solving

- Identify problems in real world solvable by computers
- Understand the problem
  - Understand the inputs
  - Output requirements
  - Constraints under which the problem must operate
- Identify potential solutions
- Select best solution
  - Fastest
  - Most accurate

Pseudocode

- High level description of an algorithm
- More structured than English prose
- Less detailed than an actual program
  - Hides program design details

Algorithm ***arrayMax*(*A*, *n*)**
  Input array ***A*** of ***n*** integers
  Output maximum element of ***A***

  currentMax $\leftarrow$ *A*[0]
  for ***i*** $\leftarrow$ 1 **to** ***n*** - 1 **do**
    if *A*[*i*] > ***currentMax*** **then**
      currentMax $\leftarrow$ ***A***[***i***]
  return ***currentMax***

Pseudocode

- Expressions
  - ← assignment, like = in Java
  - = Equality testing, like == in Java
  - Superscripts and other mathematical formatting allowed
- Method Declaration
  - Algorithm *method*(arg1...)

    Input..

    Ouput..
- Indentation replaces braces

Control Flow

- if ... then .. [else...]
- while .. do ..
- repeat ... until ..
- for ... do...

Analyzing Algorithms

- Correctness
- Amount of Work done
- Space used
- Simplicity, clarity
- Optimality

Correctness

- Understand what correctness means
  - Define the characteristics of the input an algorithm is expected to work on
  - The results that each input must produce
- Prove the statement about the relationship between input and output
- Prove Correctness of algorithm

Proof of Correctness

- Simple Techniques
  - By example
  - By contrapositives and contradiction
  - Induction
  - Loop Invariants

Analysis of Amount of Work done

- Algorithm
  - Set of simple instructions to be followed to solve a problem
- Algorithm Analysis
  - Determine resources, time and space the algorithms requires
  - Helps choose among different algorithms to a solution
- Goal
  - Estimate time required to execute the algorithm
  - Reduce the running time of the program
  - Understand results of careless use of recursion

Issues in calculating running time

- Running time grows with input size
- Varies with different inputs
- Actual running time can be calculated in seconds or milliseconds
  - The system setup must be same for all inputs
    - Same hardware and software must be used
  - Actual time maybe affected by other programs running on the same machine
- A theoretical analysis is usually preferred

Average Case and Worst Case

- Running time of an algorithm is not constant
  - Depends on input
    - Can run fast for certain inputs and slow for others
    - e.g linear search
- Average Case Cost
  - Cost of the algorithm on average
  - Difficult to calculate
- Worst Case
  - Gives an upper limit for the running time
  - Easier to analyze

What we need

- Model of Computation
  - Mathematical Framework
  - Asymptotic Notation
- What to Analyze
  - Running Time Calculations
- Checking the analysis

Random Access Machine Model

- Model of Computation to analyze algorithms
- Primitive Operations
  - Assigning a value to a variable
  - Performing an arithmetic operation
  - Calling a method
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a method
- Count primitives to give high level estimate

Counting Primitives: Recap

Algorithm FindMax(S, n)

Input : An array S storing n numbers, n>=1

Output: Max Element in S

curMax <-- S[0] (2 operations)

i ← 0 (1 operations)

while i< n-1 do (n comparison operations)

    if curMax < A[i] **then** (2(n-1) operations)

        curMax <-- A[i] (2(n-1) operations)

    i ← i+1; (2 (n-1) operations)

return curmax (1 operations)

Complexity between 5n and 7n-2

Problems

- Calculate running time:
- sum = 0;

```
for( i=1; i<n; i*=2 )
        sum++;
```

- sum = 0;

```
for( i=0; i<n; i++ )
        for( j=1; j<n; j*=2 )
                sum++;
```

- sum = 0;

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

Problem continued

- sum = 0;

```
for( i=1; i<2n; i++ )
    for( j=1; j<=i; j++ )
        sum++;
```

- Consider the following code segment:

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j += i)
        x = x + 1;
}
```

Problems

- Consider the task of finding the missing element in a sequence of n elements
- Consider the task of finding the frequency of occurrence of each element in a set
- Prefix averages
  - The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:
  - $A[i] \leftarrow (X[0] + X[1] + \ldots + X[i])/(i+1)$
  - Two algorithms

Problems

- ***Algorithm*** prefixAverage1(X,n)

  Input array X of integers

  Output array A of prefix averages of X

  $A \leftarrow$ new array of n integers

  for $i \leftarrow 0$ to n-1 **do**

      $s \leftarrow X[0]$

      for $j \leftarrow 1$ to i **do**

          $s \leftarrow s + X[j]$

      $A[i] \leftarrow s/(i+1)$

  return A

Problems

- ***Algorithm*** prefixAverage2(X,n)

  Input array X of integers

  Output array A of prefix averages of X

  $A \leftarrow$ new array of n integers

  for $i \leftarrow 0$ to n-1 **do**

  $s \leftarrow s + X[i]$

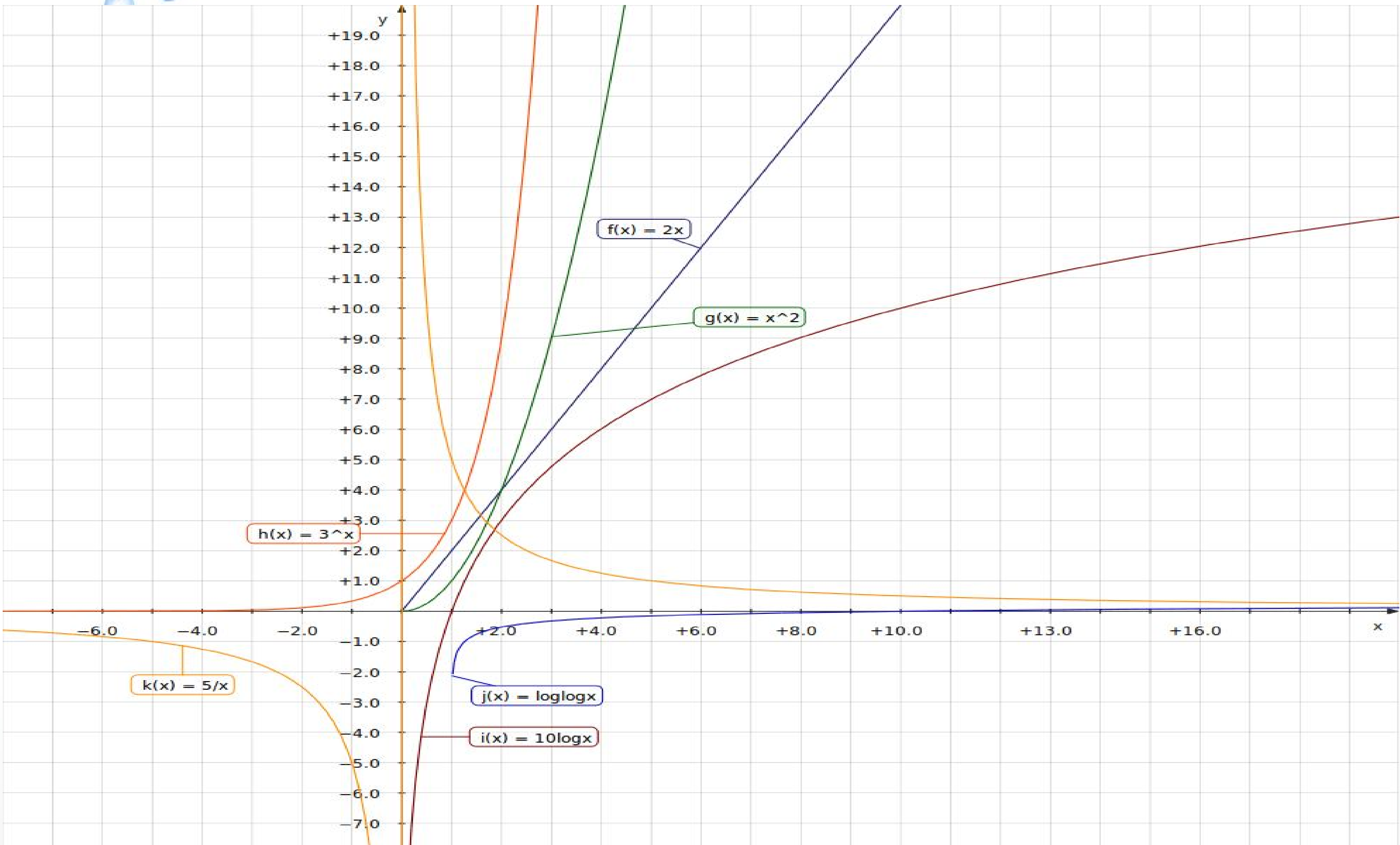
  $A[i] \leftarrow s/(i+1)$

  return A

Growth Rates of Running Time

- Important factor to be considered when estimating running time
- When experimental setup (hardware/software) changes
  - Running time is affected by a constant factor
    - 2n or 3n or 100n is still linear
  - Growth rate of the running time is not affected
- Growth rates of functions
  - Linear
  - Quadratic
  - Exponential

Some Function Plots



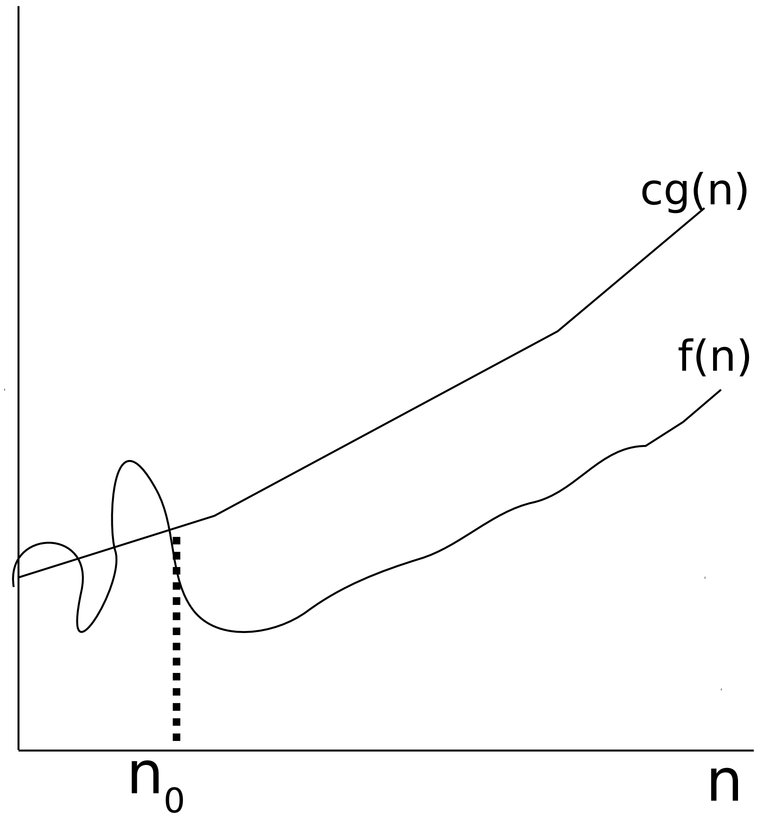
Asymptotic Analysis

- Can be defined as a method of describing limiting behavior
- Used for determining the computational complexity of algorithms
- A way of expressing the main component of the cost of an algorithm using the most determining factor
    - e.g if the running time is $5n^2+5n+3$, the most dominating factor is $5n^2+5n+3$
    - Capturing this dominating factor is the purpose of asymptotic notations

Big Oh Notation

- Given a function f(n) we say, $f(n) = O(g(n))$ if there are positive constants c and n_0 such that $f(n) <= cg(n)$ when $n >= n_0$
- Example
  - $2n + 8$ is $O(n)$
  - $2n+8 <= cn$
  - $(c-2)n >= 8$
  - $n >= 8/(c-2)$
  - Choose c = 3, and n_0 as 8, then the rule holds

O(n) – growth function



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

Example

- Example: the function n^2 is not $O(n)$
    - Must prove $n^2 <= cn$
    - $n <= c$
    - The above inequality cannot be satisfied since c must be a constant
    - Hence proof by contradiction

Example

- Example: the function n^2 is not $O(n)$
    - Must prove $n^2 <= cn$
    - $n <= c$
    - The above inequality cannot be satisfied since c must be a constant
    - Hence proof by contradiction

More Examples

- Show 7n-2 is O(n)
  - need $c > 0$ and $n_0 >= 1$ s.t $7n-2 <= cn$ for $n >= n_0$
  - this is true for $c = 7$ and $n_0 = 1$
- Show $3n^3 + 20n^2 + 5$ is $O(n^3)$
  - find c, n_0 s.t $3n^3 + 20n^2 + 5 <= cn^3$ for $n >= n_0$
  - this is true for $c = 4$ and $n_0 = 21$
- Show 3 log n + log log n is O(log n)
  - need $c > 0$ and $n_0 >= 1$ such that $3 \log n + \log \log n <= c \log n$ for $n >= n_0$
  - this is true for $c = 4$ and $n_0 = 2$

Problems

Show that $6n^2 + 20n$ is $O(n^3)$

When does the time taken to calculate the circumference of a circle run faster than the time taken to find the area of a circle, considering n to be the radius.

Problems

- Graph the following expressions. For each expression, state for which values of n that expression is the most efficient.

For the following functions: $4n^2$, $\log_3 n$, $20n$, $\log_2 n$, $n^{2/3}$

  - Graph all functions on a single plot
  - Arrange the functions by asymptotic growth rate from slowest to fastest.

Some problems

- Order the following functions by the big-Oh notation

$6n\log n$, 2^{100}, $\log^2 n$, $1/n$, n^3, $n^2\log n$

- What is the total running time of counting from 1 to n in binary if the time needed to add 1 to the current number i is proportional to the number of bits in the binary expansion of i that must change in going from i to i+1

- Is 2^{n+1} $O(2^n)$?

- Is 2^{2n} $O(2^n)$?

Big Oh Significance

- The big-Oh notation gives an upper bound on the growth rate of a function
- "f(n) is O(g(n))" means that the growth rate of f(n) is no more than the growth rate of g(n)
  - Both can grow at the same rate
- Though 1000n is larger than n^2, n^2 grows at a faster rate
  - n^2 will be larger function after n = 1000
  - Hence 1000n = O(n^2)
- The big-Oh notation can be used to rank functions according to their growth rate

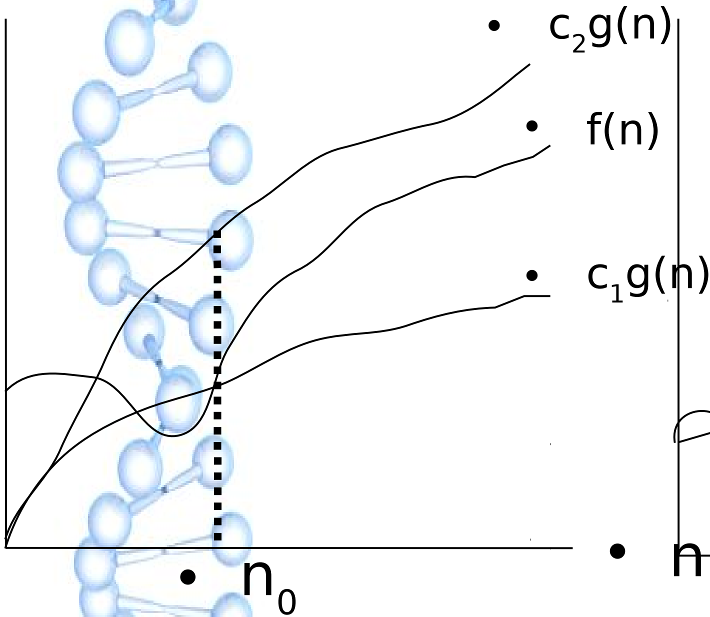
Some common rules

- If is f(n) a polynomial of degree d, then f(n) is $O(n^d)$, i.e.,
  - Drop lower-order terms
  - Drop constant factors
- Use the smallest possible class of functions to represent in big Oh
  - "2n is O(n)" instead of "2n is $O(n^2)$"
- Use the simplest expression of the class
  - "3n+ 5 is O(n)" instead of "3n + 5 is O(3n)"

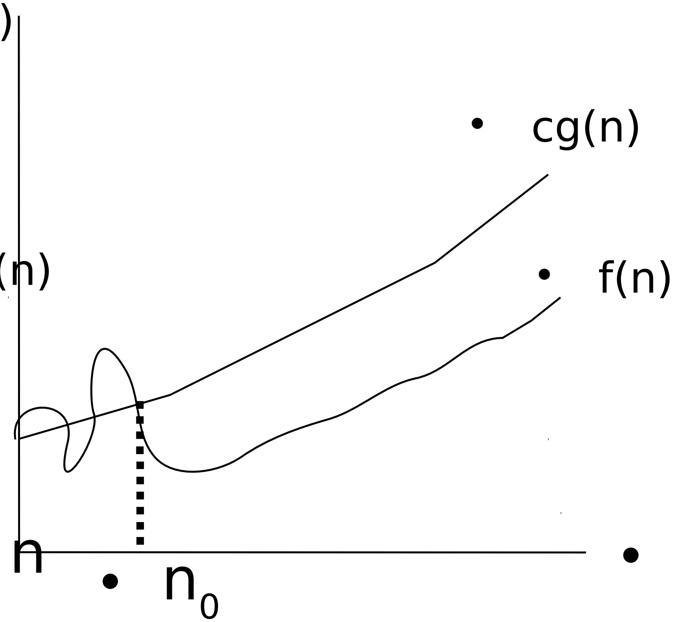
Asymptotic Notations

- f(n) = O(g(n)) if there are constants c and n_0 such that f(n)<= cg (n) when n>= n_0
- f(n) = Ω(g(n)) if there are constants c and n_0 such that f(n)>= cg(n) when n>=n_0
- f(n) = Θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).
- f(n) = o(g(n)) if f(n) = O(g(n)) and $f(n) \neq \Theta(g(n))$
  - Goal
    - Establish relative order among functions!!

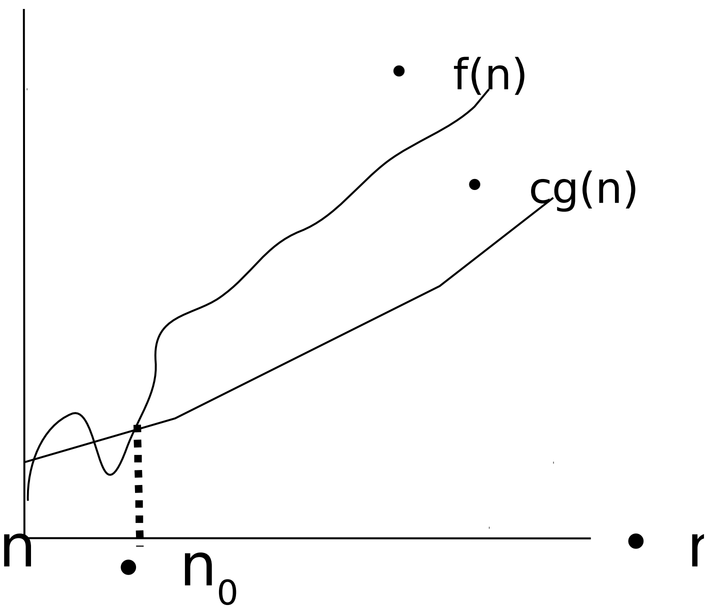
Growth of Functions



- $f(n) = \Theta(g(n))$



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



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

Problems

- $n^3 - 3n^2 - n + 1 = \Theta(n^3)$.
- For each of the following pairs of functions, either f(n) is in O(g(n)), f(n) is in Ω(g(n)), or f(n) = Θ (g(n)). Determine which relationship is correct and briefly explain why.
  - $f(n) = \log n^2$; $g(n)= \log n + 5$
  - $f(n) = (n^2 - n)/2$, $g(n) = 6n$

Importance of Asymptotic Notation

- Though 1000n is larger than n^2, n^2 grows at a faster rate
  - n^2 will be larger function after $n = 1000$
  - $1000n = O(n^2)$
- If f(n) is O(g(n)), we are guaranteeing that f(n) grows at a rate no faster than g(n)
- f(n) is Ω(g(n)), then g(n) is lower bound

Importance of Asymptotics

- Table of max-size of a problem that can be solved in one second, one minute and one hour for various running times measures in microseconds [Goodrich]

Running Time	Maximum Problem Size (n)		
	1sec	1 min	1 hour
400n	2500	150000	9000000
20nlogn	4096	166666	7826087
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

Asymptotic Rules

- If d(n) is O(f(n)), ad(n) is O(f(n)), for any a>0
  - $d(n) <= c f(n)$
  - $a d(n) <= ac f(n)$ // ac is still a constant, hence proved
- If d(n) is O(f(n)), and e(n) is O(g(n)), then d(n)+e(n) is O(f(n)+g(n))
  - $d(n) <= c_1 f(n)$ and $e(n) <= c_2 g(n)$
  - $d(n)+e(n) <= c_1 f(n)+c_2 g(n)$
  - Choose a constant c_3 which is max of (c_1,c_2). Then $d(n)+e(n) <= c_3(f(n)+g(n))$

Asymptotic Rules

- 3. If d(n) is O(f(n)), and e(n) is O(g(n)), then d(n)e(n) is O(f(n)g(n))
  - $d(n) \le c_1 f(n)$ and $e(n) \le c_2 g(n)$
  - $d(n)e(n) \le c_1 f(n) c_2 g(n)$
  - $d(n)+e(n) \le c_3(f(n)+g(n))$ // $c_3 = c_1 c_2$
- 4. If d(n) is O(f(n)), and f(n) is O(g(n)), then d(n) is O(g(n))
  - $d(n) \le c_1 f(n)$ and $f(n) \le c_2 g(n)$
  - $\Rightarrow d(n) \le c_1 c_2 g(n) \le c_3 g(n)$ // $c_3 = c_1 c_2$

Asymptotic Rules

- 5.If d(n) is O(f(n)), and e(n) is O(g(n)), then d(n)+e(n) is Max(O(f(n)), O(g(n)))
- 6. n^x is $O(a^n)$ for any fixed x>0, a>1
  - $n^x <= ca^n => \log n^x <= c \log a^n$
  - $x \log n <= cn \log a$
- 7. $\log n^x$ is O(log n) for any fixed x>0
  - $\log n^x <= c \log n => x\log n <= c\log n$
- 8. $\log^x n$ is $O(n^y)$ for some constant x>0, y>0
  - $(\log n)^x <= cn^y$

Example

- Show $2n^3 + 4n^2 \log n$ is $O(n^3)$
  - $\log n$ is $O(n)$ (rule 8)
  - $4n^2 \log n$ is $O(4n^3)$ (rule 3)
  - $2n^3 + 4n^2 \log n$ is $O(2n^3 + 4n^3)$ (rule 2)
  - $2n^3 + 4n^3$ is $O(n^3)$ (rule 1 or polynomial rule)
  - $2n^3 + 4n^2 \log n$ is $O(n^3)$ (rule 4)