Data Structures and Algorithms

University of Antwerp

Data Structures and Algorithms

Lecturer: Dr. Ali Anwar

IDLab
INTERNET & DATA LAB

imec

Study Material

- Slides of the course (Lectures)
- Practical Lessons (Labs)
- Textbook:
  - Data Structures and Algorithms in C++ (Second Edition)
    Authors: Michael T. Goodrich, Roberto Tamassia, David Mount
    ISBN: 0470383275
    Pdf available on the blackboard
- Reference Book:
  - Data Structures and Algorithm Analysis
    Author: Clifford A. Shaffer (Edition 3.2 C++ Version)
    Pdf available on the blackboard

Course Organization and Evaluation

- Lectures:
  - 6 sessions (two hours each)
  - OOP, Abstract Data Types, Analysis Tools, Lists, Linked lists, Stacks, Queues, Sorting, Iterators, Trees, Hash tables, Graphs

- Evaluations:
  - Exams in January / June
  - Theory Exam: closed book, written (50 %)
  - Practical Exam: Project with an oral presentation (50%)

Course Objectives

- Awareness of cost and benefits attached to each design choice.
- Get to know and apply the most commonly used data structures and algorithms:
  - Essential requirement for a software engineer.
- Understand how the costs of a data structure or program are measured and how to apply this:
  - With these techniques you can also assess the efficiency of new data structures.

Lecture 1

Learning Outcomes

Questions

- Why do we need data structures?
- What is a data structure?
- What is an algorithm?
- What is an abstract data type (ADT)?
- How to measure an algorithmic efficiency?

Why Do We Need Data Structures?

- Computer Program = Instructions + Data
  - Need of efficient structures to organize the data;
  - Need of efficient algorithms to process / retrieve the data.
- For large scale applications, efficiency is the key:
  - Processing of large amount of data;
  - In time processing and acquisition of results require optimization;
  - Data structures and algorithms are used to ensure optimality.

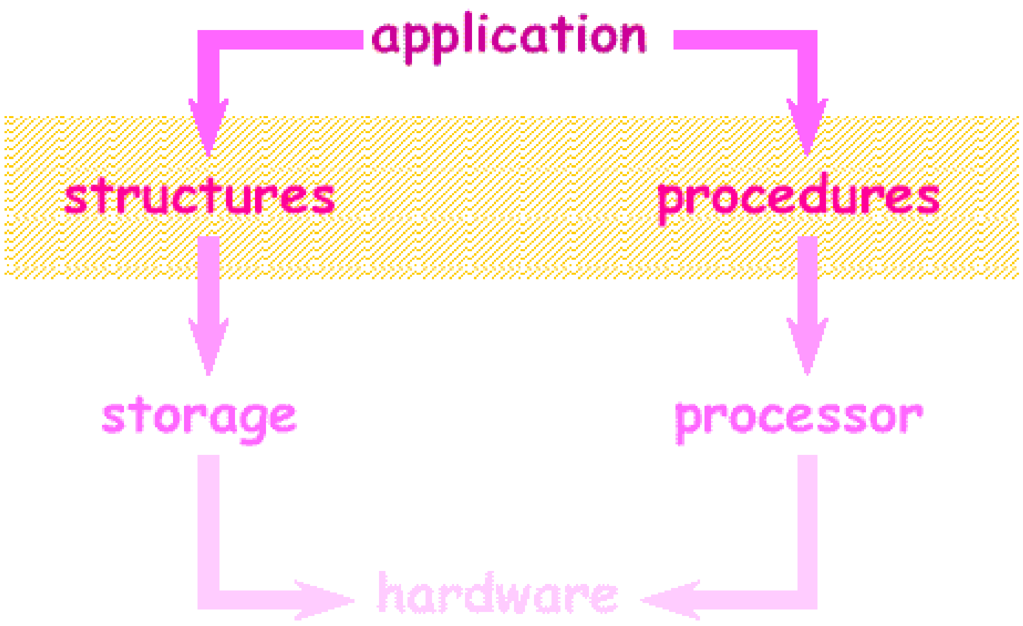
Main Components of a Software System

Abstract Data

Types

Concrete Data

Structures



Algorithms

Instructions

What Is a Data Structure?

- Organization of the data for efficient usage
- Best example is an array!
- Can you define an array?
  - Collection of data types / items stored at contagious memory locations
- Benefits:
  - Easier to locate the objects (due to indexing)
- Weakness:
  - Contagious block of memory should be reserved
  - Difficult to update the size based on runtime requirements
  - Insertion and deletion at arbitrary locations is costly

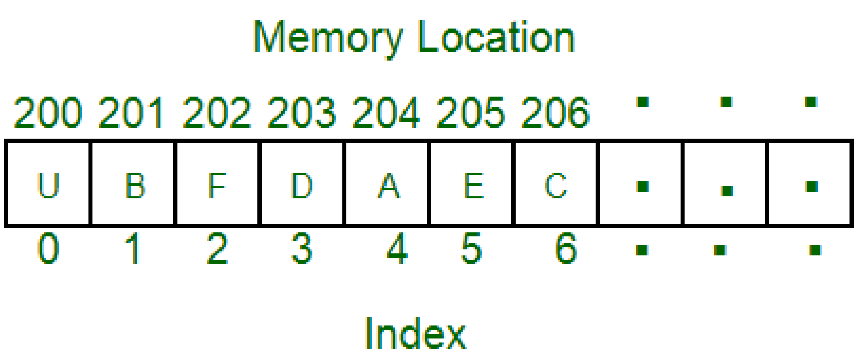


Image credit: https://bit.ly/3mFxvqP

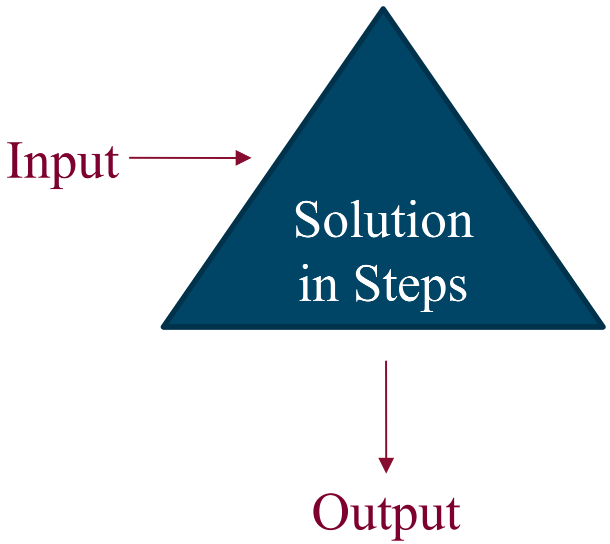
Algorithms

- What is an algorithm?
  - A solution method for an algorithmic problem
    - Example: sorting a list of names
- Presentation:
  - "In words", flow chart, pseudocode, code in Java, C, C++, ...
- Requirements :
  - correctness
  - consists of concrete steps ("recipe", executable)
  - unambiguously
  - finite number of steps (in description)
  - must end (no infinite loop)

Effectiveness

Definiteness

Finiteness

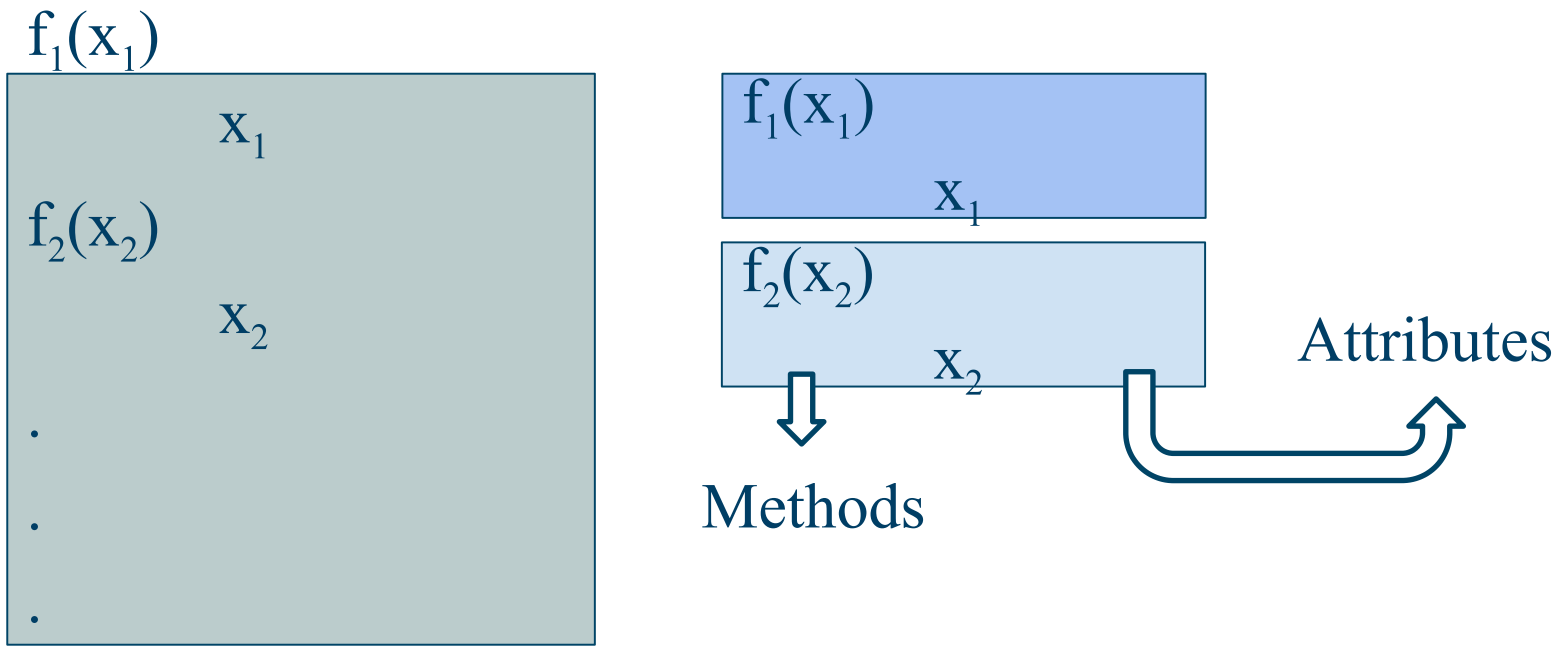


Part 1: Review of Object Oriented Programming

Goodrich Chapter 2: Sections 2.1, 2.2

OOP vs Procedural Programming

$f(x)$: functions where x: arbitrary variables



Video Review: https://bit.ly/3cf1dyd

Benefits of OOP

- Organized and structured code
- Eliminates spaghetti code

- Similar attributes and methods in a single object → Encapsulation
- Simpler and elegant interface for the users / reduced complexity → Abstraction
- Reusability of components and elimination of redundancy → Inheritance
- Methods take different (many) forms → Polymorphism

Design Patterns

- Algorithmic design problems
- Software Engineering problems (5-SD)

- Pattern is a general solution applicable in various distinct scenarios
- Properties:
  - **Name:** identifies the pattern
  - **Context:** application scenarios
  - **Template:** how the pattern is applied
  - **Result:** describes and analyzes the pattern result
- Some of the algorithm design patterns include the following:
  - Recursion
  - Divide-and-conquer
  - Brute force
  - The greedy method

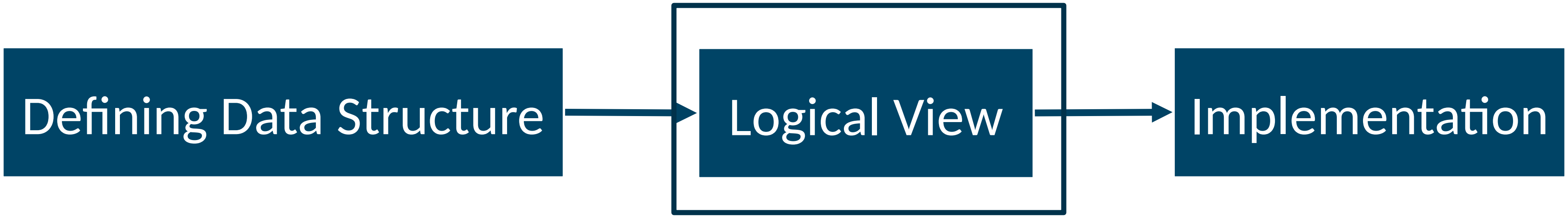
Further Reading: Goodrich: Sec 2.1.3; Shaffer: Sec 1.3

Part 2: Abstract Data Types

Shaffer Chapter 1: Sections 1.2

ADT (Abstract Data Type)

- ADT specifies the type of the data stored, the operations supported on them, and the types of parameters of the operations.
- ADT specifies what each operation does, but not how it does it.
- Each ADT is determined by its interface, which defines the set of available operations
- ADT is realized by a class (in C++). A class defines the data being stored and the operations supported by the objects that are instances of the class.



Implementing ADT in C++ (Lab Sessions)

- To be useful, an ADT must usually contain some internal data. These are declared as data members of the class.
- Many ADTs are rich in attributes and lean in operations. That means that many of the function members will be “gets” and “sets” that do little more than fetch and store in private data members.
- C++ provides a special kind of member function to streamline the initialization process. It’s called a constructor. A constructor is called when we define a new variable, and any parameters supplied in the definition are passed as parameters to the constructor.
- Just as C++ provides special functions, constructors, for handling initialization, it also provides special functions, destructors, for handling clean-up. Destructors are never called explicitly. Instead, the compiler generates a call to an object’s destructor for us.

Example ADT

An ***array*** is a collection of objects of the same type:

- Stores a required amount of elements of a specific data type;
- Inserts or modifies the elements at a given position;
- Reads elements at certain position;
- Supports logical operations like sorting.

Many more operations can be defined, can you think of any?

Part 3: Algorithm Efficiency

Goodrich Chapter 4

Shaffer Chapter 3

Analysis of Algorithms

Efficiency (complexity) is how well you're using your computer's resources to get a particular job done.

- Time complexity means how long does your code take to run.
- Space complexity means how much storage space do you need for your code.

Usually, the complexity class of the algorithm is simply determined by the number of loops and how often the content of those loops are being executed.

Analysis of Algorithms

How do we measure the performance of an algorithm?

- **Experimental Studies**
  - Run programs and measure the running time.
- **Theoretical Analysis**
  - Determine the factors that affect the execution time:
    - for most algorithms, the transit time depends on "size" of the input;
    - this term is expressed as a function T(n) over input size n. This T(n) indicates the growth of running time.

Running Time

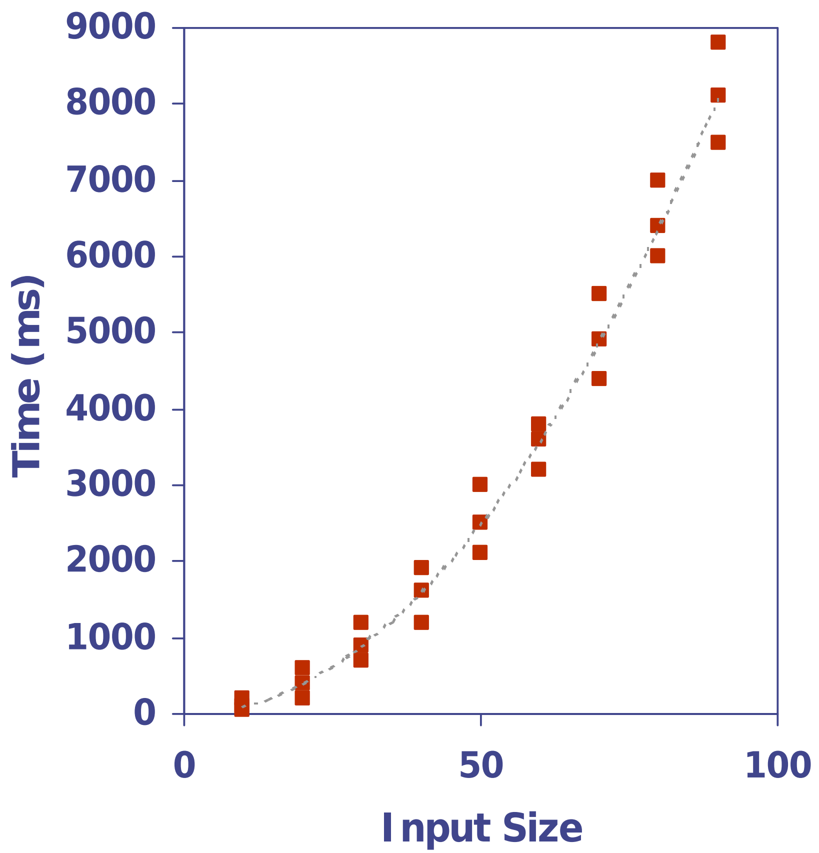
- **Data structure** is a systematic way of organizing and accessing data
- **Algorithm** is a step-by-step procedure for performing some task in a finite amount of time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time:
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics

Upper bound on arbitrary # inputs

Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like *clock()* to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult.
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size n
- Contemplates all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

- High-level description of an algorithm
- More structured
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

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

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

Pseudocode Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***…])

  Input …

  Output …

Method call

var.method (***arg*** [, ***arg***…])

Return value

return ***expression***

Expressions

$\leftarrow$ Assignment (like = in C++)

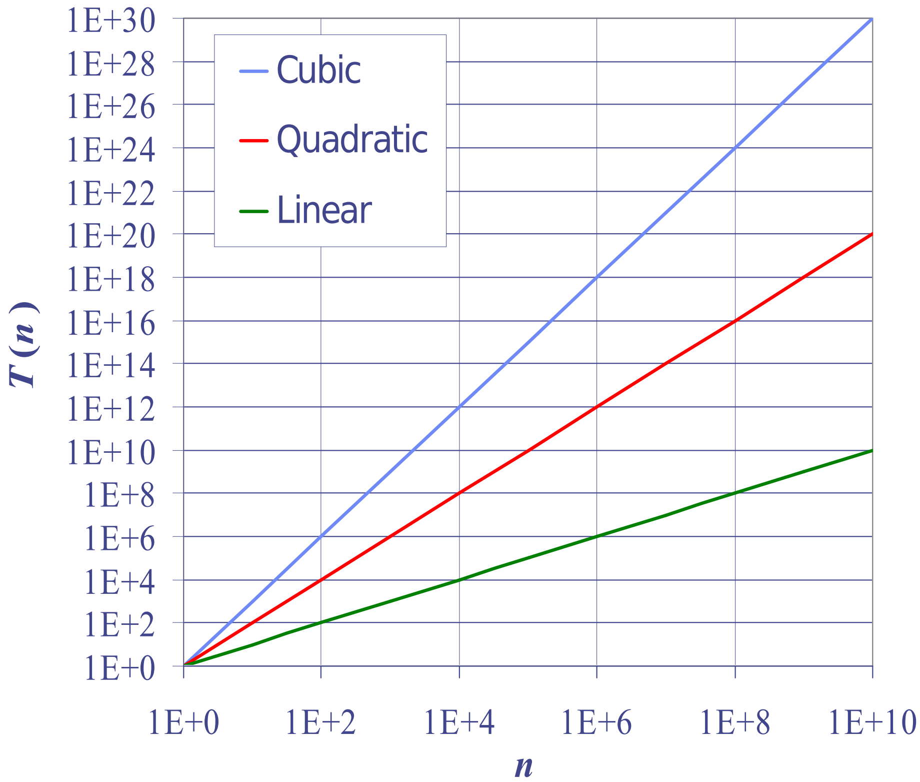
$=$ Equality testing (like == in C++)

n^2 Superscripts and other mathematical formatting allowed

Seven Important Functions

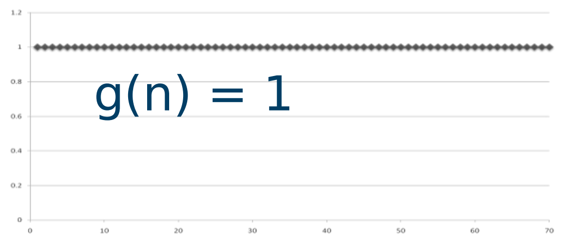
Seven functions that often appear in algorithm analysis:

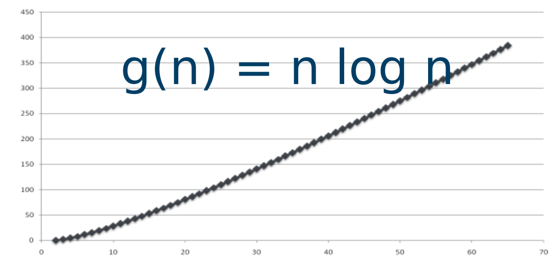
- Constant ≈ ***1***
- Logarithmic ≈ **log *n***
- Linear ≈ ***n***
- N-Log-N ≈ ***n* log *n***
- Quadratic ≈ n^2
- Cubic ≈ n^3
- Exponential ≈ 2^n

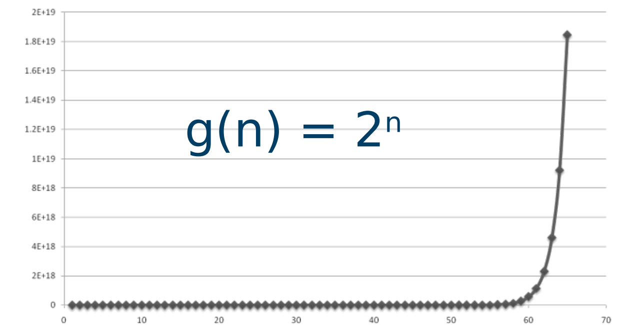


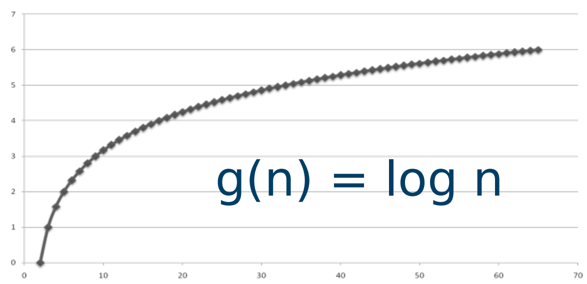
In a log-log chart, the slope of the line corresponds to the growth rate

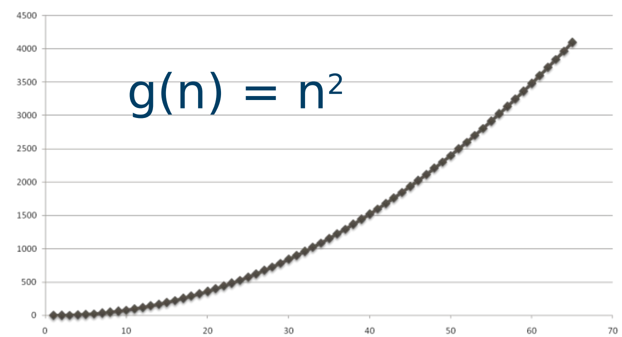
7 Functions Graphed Using "Normal" Scale

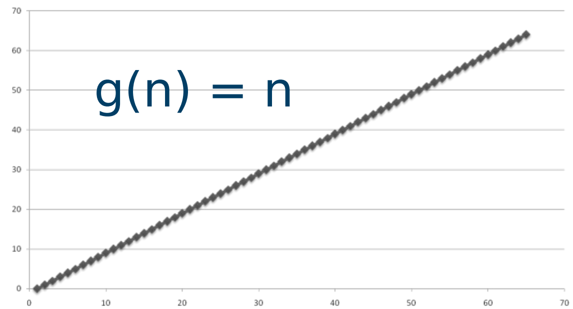


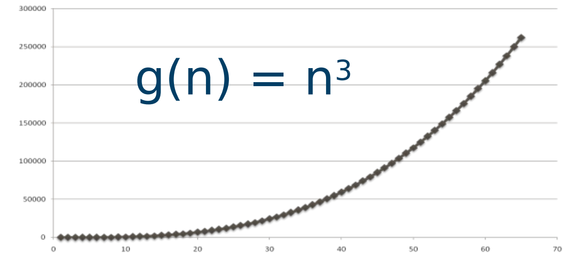




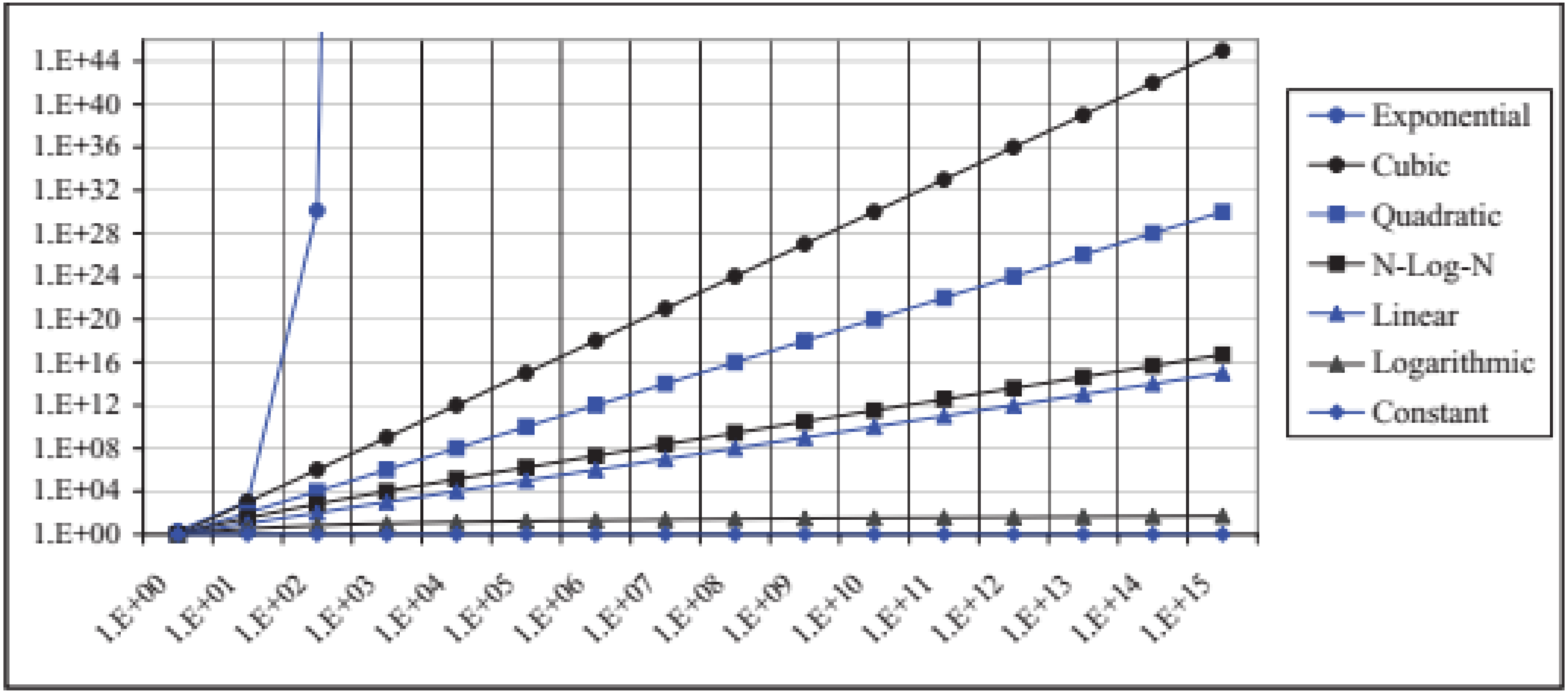








Growth Rates of 7 Important Functions



Source: Goodrich Fig 4.2

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM (Random Access Machine) model

- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm ***arrayMax(A, n)***	# operations	
currentMax ← *A*[0]	***1***	
for ***i*** ← 1 **to** ***n*** - 1 **do**	***n***	
if *A*[*i*] > *currentMax* **then**	***n***	
currentMax ← *A*[*i*]	***n***	
i ← *i* + *1*	***n***	
return *currentMax*	***1***	$\geq n$

Total: $4n + 2$

Estimating Running Time

- Algorithm ***arrayMax*** executes $4n + 2$ primitive operations in the worst case.

Define:

- a = time taken by the fastest primitive operation
- b = time taken by the slowest primitive operation

- Let T(n) be worst-case time of ***arrayMax***. Then

$$a\,(4n + 2) \le T(n) \le b(4n + 2)$$

- Hence, the running time ***T(n)*** is bounded by two linear functions

Growth Rate of Running Time

- Changing the hardware/ software environment:
  - Affects ***T(n)*** by a constant factor, but
  - Does not alter the growth rate of ***T(n)***
- The linear growth rate of the running time ***T(n)*** is an intrinsic property of algorithm ***arrayMax***

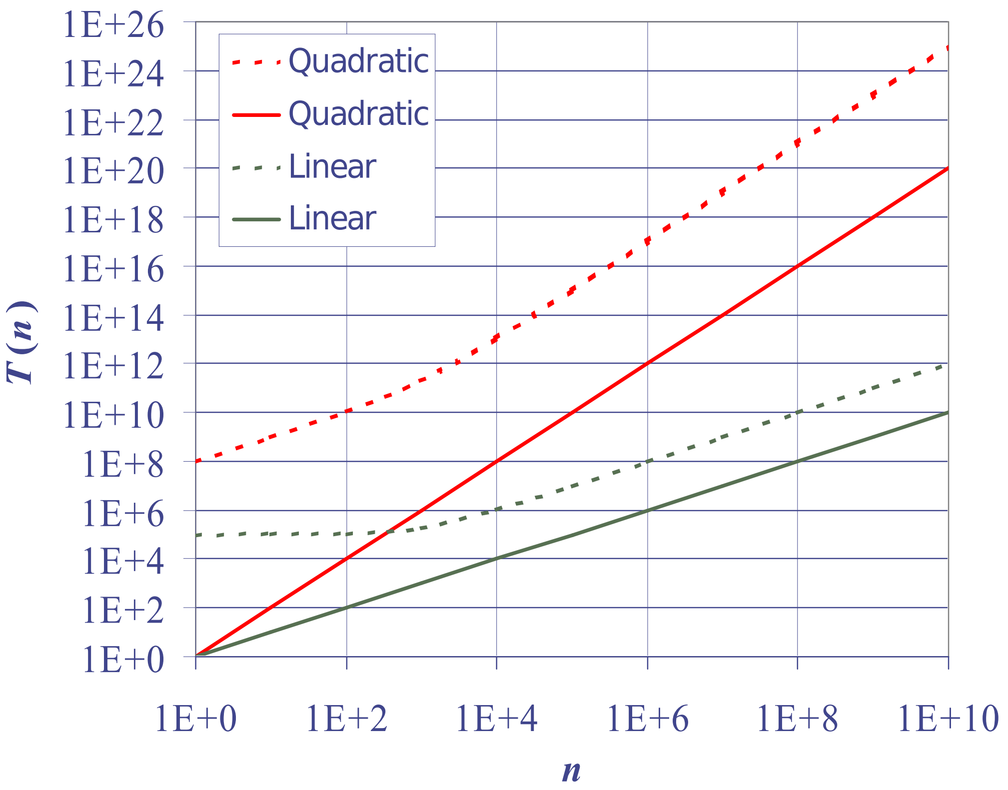
Why Growth Rate Matters?

if runtime is...	*time for n + 1*	*time for 2 n*	*time for 4 n*
$c \log n$	$c \log (n + 1)$	$c (\log n + 1)$	$c(\log n + 2)$
$c\, n$	$c\, (n + 1)$	$2c\, n$	$4c\, n$
$c\, n \log n$	$\sim c\, n \log n$ + $c \log n$	$2c\, n \log n + 2c\, n$	$4c\, n \log n + 4c\, n$
$c\, n^2$	$\sim c\, n^2 + 2c\, n$	$\boxed{\mathbf{4c\, n^2}}$	$16c\, n^2$
$c\, n^3$	$\sim c\, n^3 + 3c\, n^2$	$8c\, n^3$	$64c\, n^3$
$c\, 2^n$	$c\, 2^{n+1}$	$c\, 2^{2n}$	$c\, 2^{4n}$

runtime quadruples when problem size doubles

Constant Factors

- The growth rate is not affected by:
  - constant factors or
  - lower-order terms
- Examples:
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function



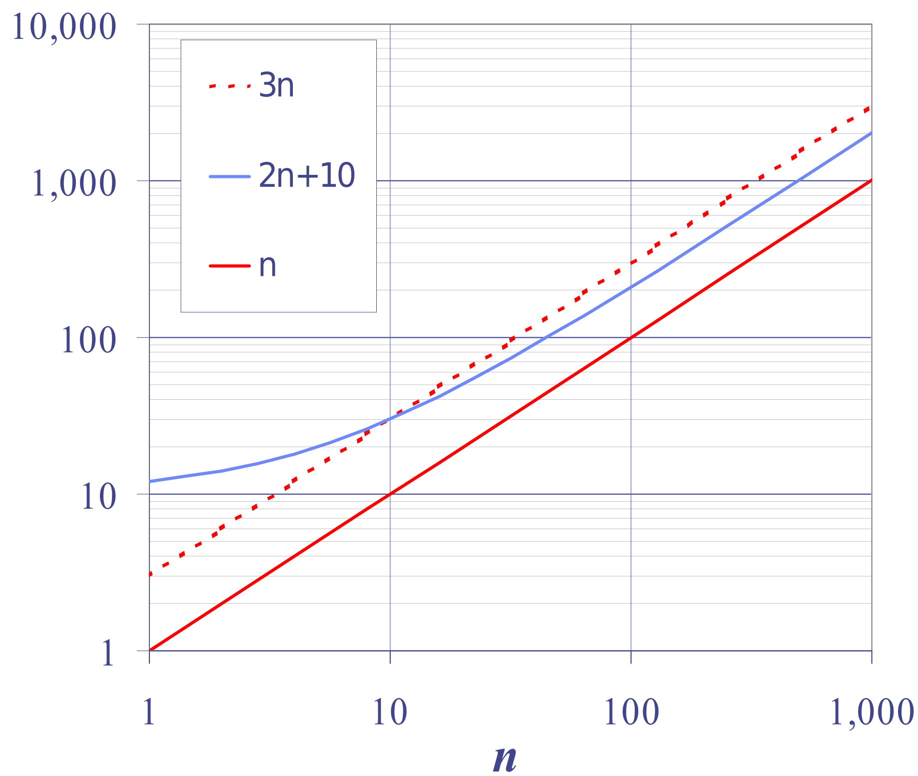
Big O Notation

How does the runtime of the function grow as the size of input '***n***' grows?

- Given functions f(n) and g(n), we say that ***f(n)*** is ***O(g(n))*** if there are positive constants ***c*** and n_0 such that

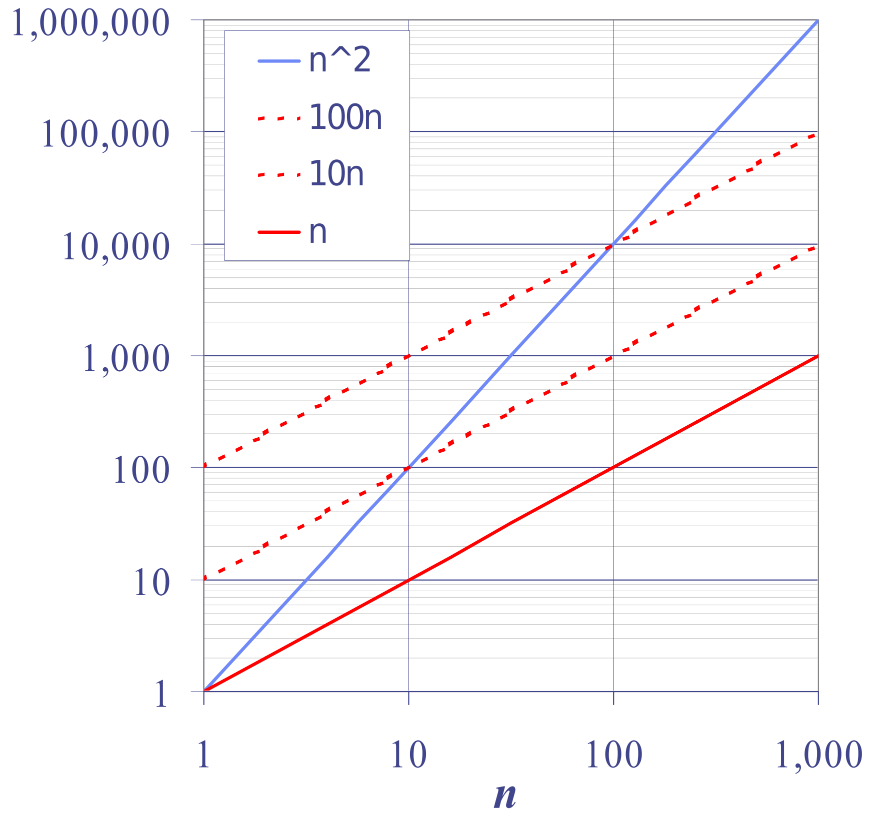
$$f(n) \leq cg(n) \quad \text{for} \quad n \geq n_0$$

- Example: 2n + 10 is ***O(n)***
  - $2n + 10 \leq cn$
  - $(c - 2)n \geq 10$
  - $n \geq 10 / (c - 2)$
  - Pick $c=3$ and $n_0=10$



Big O Example

- Example: the function $\boldsymbol{n^2}$ is not $\boldsymbol{O(n)}$:
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant



More Big O Examples

- **7n - 2**

  7n-2 is O(n)
  need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq c\,n$ for $n \geq n_0$
  this is true for $c = 7$ and $n_0 = 1$

- **$3n^3 + 20n^2 + 5$**

  $3n^3 + 20n^2 + 5$ is $O(n^3)$
  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c\,n^3$ for $n \geq n_0$
  this is true for $c = 4$ and $n_0 = 21$

- **3 log n + 5**

  3 log n + 5 is O(log n)
  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \log n$ for $n \geq n_0$
  this is true for $c = 8$ and $n_0 = 2$

Big O and Growth Rate

- The Big O notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the Big O notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big O Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors

- Use the smallest possible class of functions:
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"

- Use the simplest expression of the class:
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in Big O notation.
- To perform the asymptotic analysis:
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with Big O notation
- Example:
  - We determine that algorithm ***arrayMax*** executes at most $4\boldsymbol{n} + 2$ primitive operations
  - We say that algorithm ***arrayMax*** "runs in $\boldsymbol{O}(\boldsymbol{n})$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations.

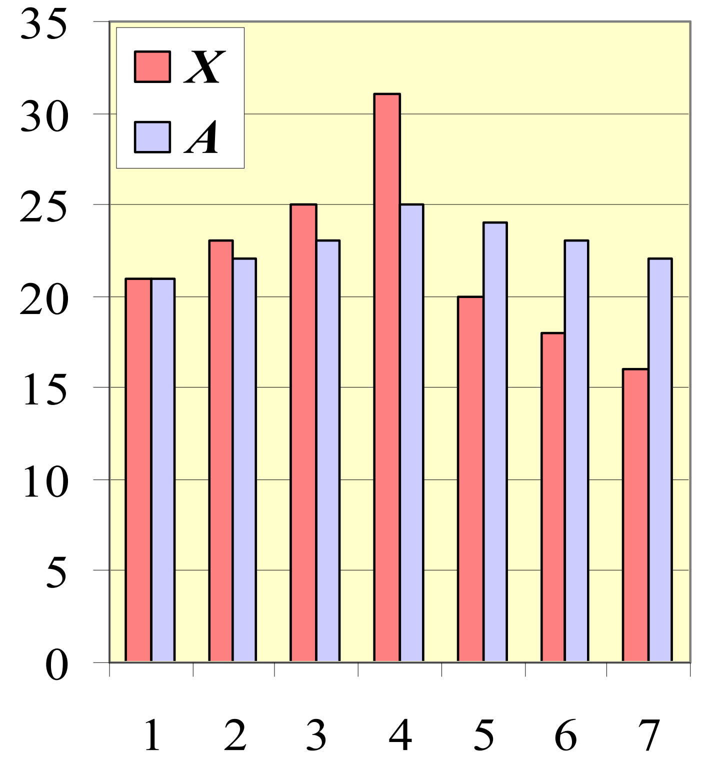
Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages

- The ***i***-th prefix average of an array ***X*** is average of the first (***i*** + 1) elements of ***X*:**

$$\boldsymbol{A}[\boldsymbol{i}] = (\boldsymbol{X}[0] + \boldsymbol{X}[1] + \ldots + \boldsymbol{X}[\boldsymbol{i}])/(i+1)$$

- Computing the array ***A*** of prefix averages of another array ***X*** has applications to financial analysis



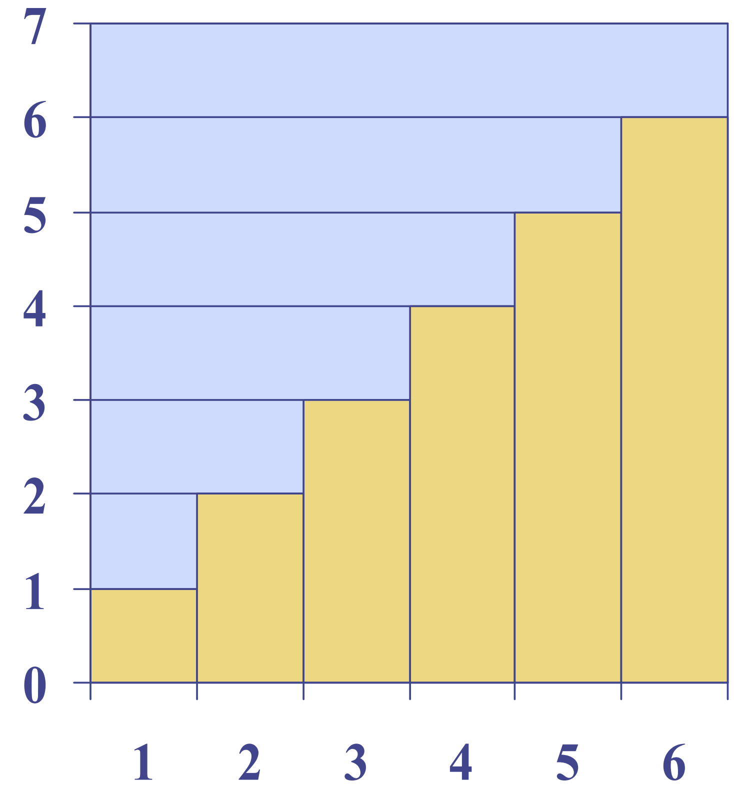
Prefix Averages (Quadratic)

The following algorithm computes prefix averages in quadratic time by applying the definition

Algorithm ***prefixAverages1*(*X*, *n*)**
Input array ***X*** of ***n*** integers
Output array ***A*** of prefix averages of ***X*** #operations
A ← new array of ***n*** integers ***n***
for ***i*** ← 0 **to** ***n*** **-** 1 **do** ***n***
s ← ***X***[0] ***n***
for ***j*** ← 1 **to** ***i*** **do** 1 **+** 2 **+** …**+** (***n***
- 1)
s ← ***s*** **+** ***X***[***j***] 1 **+** 2 **+** …**+** (***n***
- 1)
A[***i***] ← ***s*** / (***i*** **+** 1) ***n***
return ***A*** 1

Arithmetic Progression

- The running time of ***prefixAverages1*** is $\boldsymbol{O}(1 + 2 + \ldots + \boldsymbol{n})$
- The sum of the first ***n*** integers is $\boldsymbol{n}(\boldsymbol{n} + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm ***prefixAverages1*** runs in $\boldsymbol{O}(\boldsymbol{n}^2)$ time



Prefix Averages (Linear)

The following algorithm computes prefix averages in linear time by keeping a running sum

```
Algorithm prefixAverages2(X, n)
    Input array X of n integers
    Output array A of prefix averages of X
    #operations
    A ← new array of n integers                        n
    s ← 0                                         1
    for i ← 0 to n - 1 do                              n
            s ← s + X[i]                               n
            A[i] ← s / (i + 1)                         n
    return A                                           1
```

Algorithm ***prefixAverages2*** runs in $O(n)$ time

Math to Review

- Summations
- Logarithms and Exponents
  - Properties of logarithms:
    - $\log_b(xy) = \log_b x + \log_b y$
    - $\log_b (x/y) = \log_b x - \log_b y$
    - $\log_b xa = a\log_b x$
    - $\log_b a = \log_x a / \log_x b$
  - Properties of exponentials:
    - $a^{(b+c)} = a^b a^c$
    - $a^{bc} = (a^b)^c$
    - $a^b / a^c = a^{(b-c)}$
    - $b = a^{\log_a b}$
    - $b^c = a^{c*\log_a b}$
- Proof Techniques
- Basic Probability

Relatives of Big O

- **Big Omega**
  - $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$

- **Big Theta**
  - $f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \bullet g(n) \leq f(n) \leq c'' \bullet g(n)$ for $n \geq n_0$

Intuition for Asymptotic Notation

Big O – Upper Bound

- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

Big Omega – Lower Bound

- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

Big Theta – Sandwiched or tightest bound

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

Example Uses of the Relatives of Big O

- **$5n^2$ is $\Omega(n^2)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

- **$5n^2$ is $\Omega(n)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $\mathrm{f}(n) \geq c \cdot g(n)$ for $n \geq n_0$

  Let $c = 1$ and $n_0 = 1$

- **$5n^2$ is $\Theta(n^2)$**

  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We've already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $\mathrm{f}(n) \leq c \cdot g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$