CS 5012: Foundations of Computer Science

Analysis of Algorithms

- Reading: Chapter 5 of MSD text (see Collab)
  - Except Section 5.5 on recursion
- Reading: Chapters 1, 2 & 3 of Algorithms text

UNIVERSITY OF VIRGINIA
DATA SCIENCE INSTITUTE

Goals for this Unit

- Begin a focus on data structures and algorithms
- Understand the nature of the performance of algorithms
- Understand how we measure performance
- Begin to see the role of algorithms in the study of Computer Science

Algorithm

- An **algorithm** is
  - a detailed step-by-step method for solving a problem
- Computer algorithms
- *Properties of algorithms*
  - Steps are precisely stated
  - Determinism: based on inputs, previous steps
  - Algorithm terminates
  - Also: correctness, generality, efficiency

Data Structures

- Some definitions of *data structure*:
  - A scheme for organizing related pieces of information
  - A logical relationship among data elements that is designed to support specific data manipulation functions
  - Contiguous memory used to hold an ordered collection of fields of different types
- (B seems to be the best one of the above)
- Examples: ArrayList, HashSet, trees, tables, stacks, queues

Efficiency of Implementations

- There are various operations that are useful (such as *sorting* and *searching*)
- There may be more than one way to implement these operations
  - Advantages and disadvantages
  - Efficiency / performance is often a major consideration

- **Question**: How do we compare <u>efficiency</u> of implementations?

- **Answer**: We compare the <u>algorithms</u> that implement the operations

Efficiency?

- The **efficiency** of an algorithm measures the amount of resources consumed in solving a problem of size ***n***
  - CPU (time) usage, memory usage, disk usage, network usage, …
-
- In general, the resource t1hat interests us the most is **time**
  - That is, how fast an algorithm can solve a problem of size ***n***
  - We can use the same techniques to analyze the consumption of other resources, such as memory space

Why Not Just Time Algorithms?

What do you think?

Why Not Just Time Algorithms?

- We want a measure of work that gives us a direct measure of the *efficiency* of the algorithm
  - independent of computer, programming language, programmer, and other implementation details
  - Usually depending on the ***size of the input***
  - Also often dependent on the ***nature of the input***
    - Best-case, worst-case, average

Efficiency?

- It would seem that the most obvious way to measure the efficiency of an algorithm is to run it with some **specific input** and measure how much **processor time** is needed to produce the correct solution
- This type of "wall clock" timing is called **benchmarking**
- However, this produces a measure of efficiency for only one particular case, and is *inadequate* for predicting how the algorithm would perform on a different data set

- Therefore, benchmarking is not an appropriate way to mathematically analyze the general properties of algorithms

A Measure Independent of Input

- We need a way to formulate general guidelines that allow us to state that, for any arbitrary input, one method is likely to perform better than the other

- The time it takes to solve a problem is usually an increasing function of its size – *the bigger the problem, the longer it takes to solve*
- We need a formula that associates ***n***, the problem size, with ***t***, the processing time required to obtain a solution
- This relationship can be expressed as: $t = f(n)$

Analysis of Algorithms

- Use mathematics as our tool for analyzing algorithm performance
  - Measure the algorithm itself, its nature
  - Not its implementation or its execution
- Need to count something
  - Cost or number of steps is a function of input size *n*: e.g. for input size *n*, cost is ***f(n)***
  - Count <u>all steps</u> in an algorithm? (Hopefully avoid this!)

Example of Total Execution Time

- One might find the total time of an algorithm by adding the times for *all* statements:

```
•     statement 1;          t1
•     statement 2;          t2
•     …                     …
•     statement k;          tk
```

- Total time = t**1** + t**2** + … + t**k**
- **Probably want to avoid doing this**

Counting Operations

- Strategy: choose one operation or one section of code such that
  - The total work is always roughly proportional to how often that's done
- So we'll just count:
  - An algorithm's "basic operation"
  - Or, an algorithms' "**critical section**"
- Sometimes the basic operation is some action that's fundamentally central to how the algorithm works
  - Example: Search a List for a target involves comparing each list-item to the target
  - The comparison operation is "fundamental"

Asymptotic Analysis

- **Algorithmic complexity** is a very important topic in computer science. Knowing the complexity of algorithms allows you to answer questions such as
  - How long will a program run on an input?
  - How much space will it take?
  - Is the problem solvable?
- These are important bases of comparison between different algorithms
- An understanding of algorithmic complexity provides programmers with insight into the efficiency of their code

Asymptotic Analysis

- Analysis of the running time of programs usually involves an estimate of time **as a function of the input size, *n***
- As an example, we may say
  - The standard *insertion* sort takes time T(n)
    - $\mathrm{T(n)} = cn^2 + k$, for some constants *c* and *k*
  - *Merge* sort takes time T'(n)
    - $T'(n) = c'nlog_2(n) + k'$, for some constants *c'* and *k'*
- But what does this mean?
- Which method is "**better**"?

Asymptotic Analysis

- The asymptotic behavior of a function $f(n)$ such as $f(n) = cn$ or $f(n) = cn^2$, etc. refers to the **growth** of $f(n)$ as ***n* gets large**

- Why "as n gets large"? Typically small values of n are ignored because typically it is only until n becomes large that the differences in performance become apparent

Asymptotic Analysis

- Additionally, there is much interest in estimating how slow the program will be on large inputs (how **scalable** is it)

- Rule of thumb: the **slower** the asymptotic growth rate, the *better* the algorithm

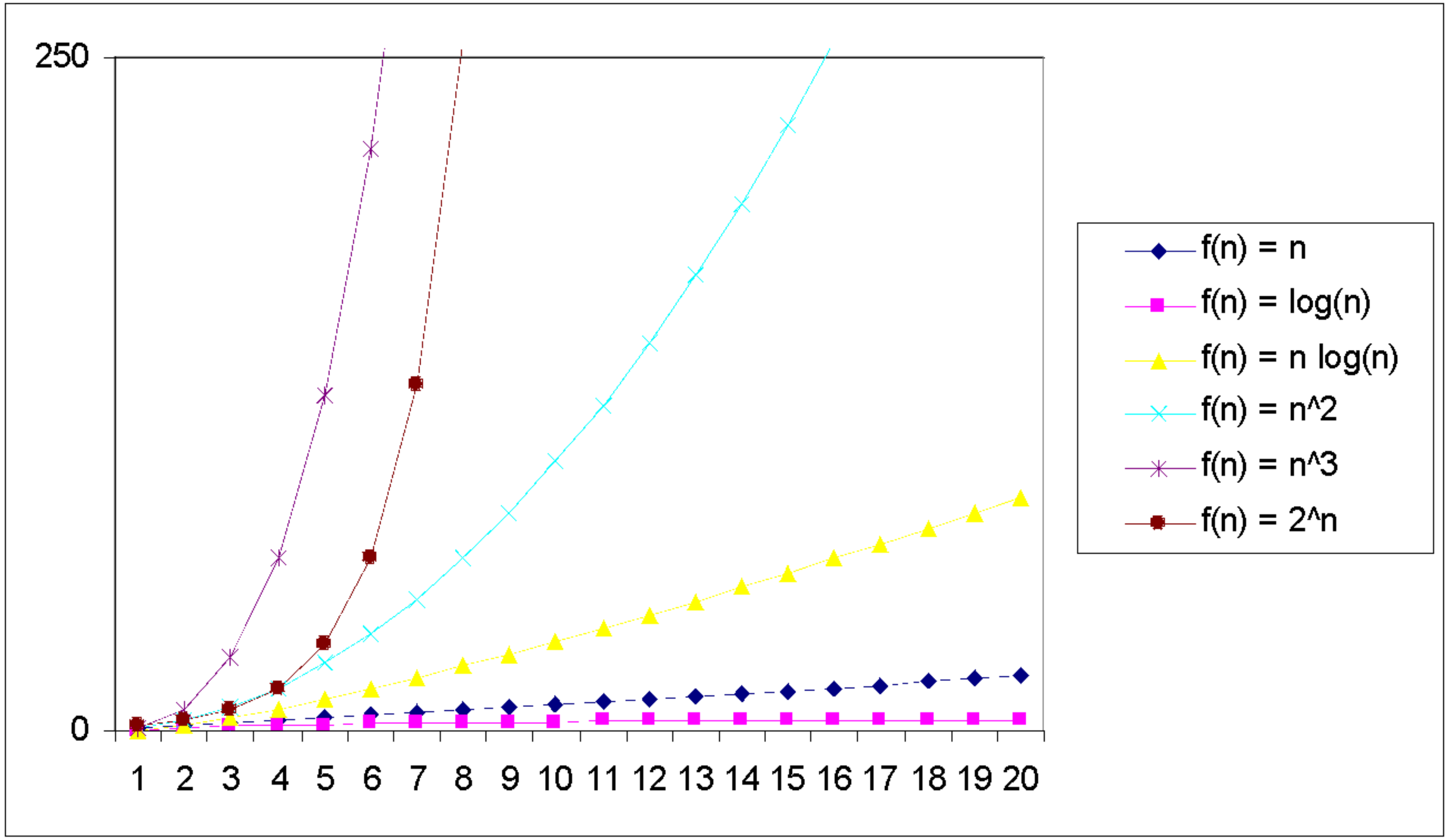
Asymptotic Analysis

- By this measure, a **linear** algorithm (*i.e., $f(n)=dn+k$*) is always asymptotically better than a **quadratic** one (*e.g., $f(n)=cn^2+q$*)
- That is because for any given (positive) *c, k, d*, and *q* there is always some *n* at which the magnitude of cn^2+q overtakes $dn+k$
- For moderate values of *n*, the quadratic algorithm could very well take less time than the linear one, for example if *c* is significantly smaller than *d* and/or *k* is significantly smaller than *q*. However, the *linear algorithm* will always be better for sufficiently large inputs

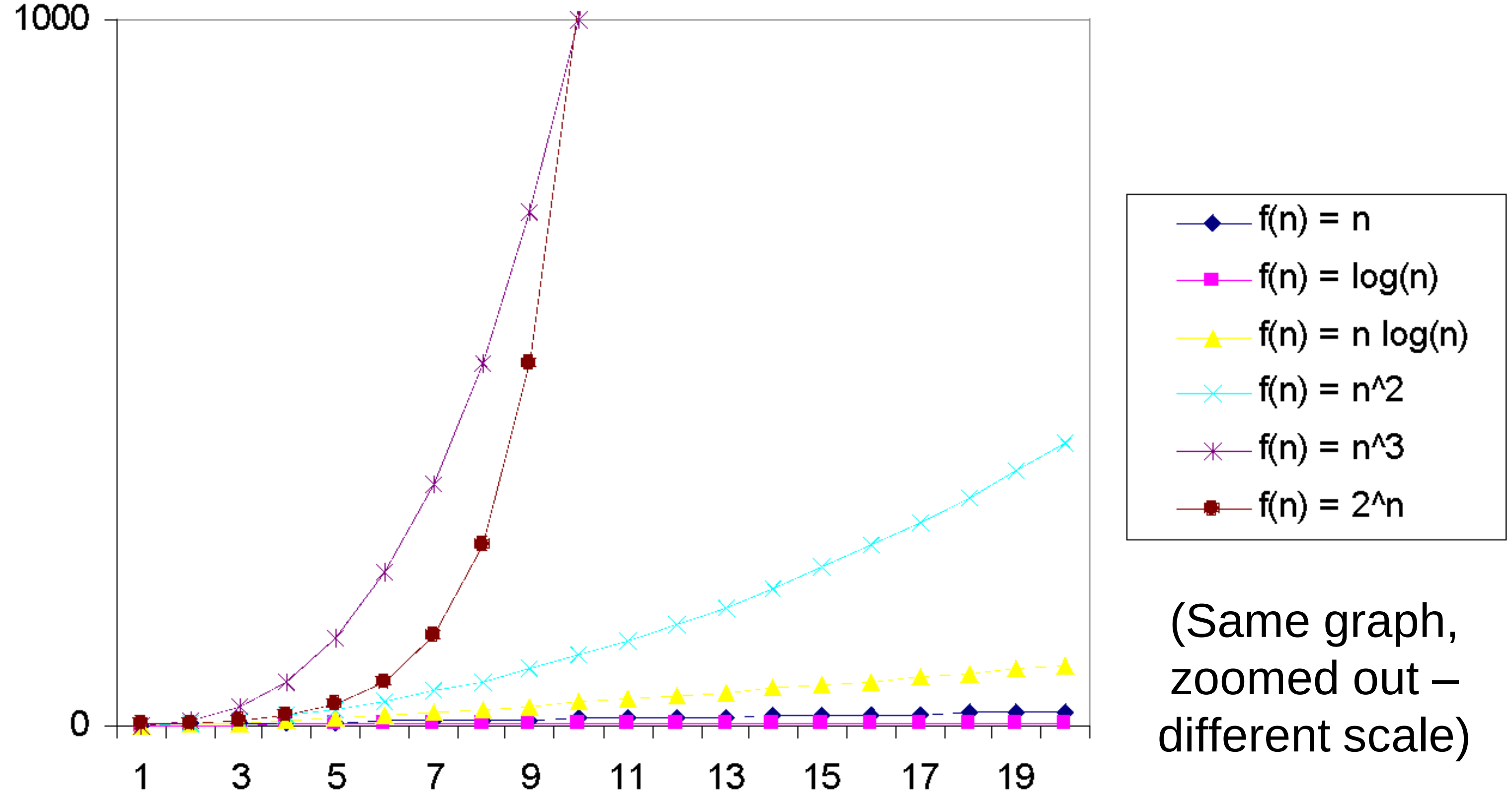
Asymptotic Analysis

- Given some formula f(n) for the count/cost of some thing based on the input size
  - We're going to focus on its "order"
  - $f(n) = 2^n$ ---> Exponential function
  - $f(n) = 100n^2 + 50n + 7$ ---> Quadratic function
  - $f(n) = 30\ n \lg n - 10$ ---> Log-linear function
  - $f(n) = 1000n$ ---> Linear function
- These functions ***grow*** at different rates
  - As inputs get larger, the amount they increase differs
- "**Asymptotic**" – how do things change as the input size *n* gets larger?

Comparison of Growth Rates (1)



Comparison of Growth Rates (2)



(Same graph, zoomed out – different scale)

Order Classes

- For a given algorithm, we count something:
  - $f(n) = 100n^2 + 50n + 7$ ---> Quadratic function
  - How different is this than this?
    $f(n) = 20n^2 + 7n + 2$
  - For large inputs?

- Order class: a "label" for all functions with the same *highest-order term*
  - Label form: **$O(n^2)$** or **$\Theta(n^2)$** or a few others
  - "**Big-Oh**" used most often than "Big-Theta"

Highest-order Term

- If a function that describes the growth of an algorithm has several terms, its order of growth is determined by the **fastest growing term**
- Smaller terms have some significance for small amounts of data
- However, when data becomes very large, a reasonably accurate estimate of the performance of the algorithm can be made by the term with the highest order

Highest-order Term (Example)

```
for (int i = 1;i<=n; i++) {                    O(1)
      perform execution of a statement         O(n)
            for (conditional statement) {
                  2nd loop                     O(n^2)
            }
}
```

- Total time = $O(n^2) + O(n) + 1$
- Simplify as per previous slide: Total execution time = $\mathbf{O(n^2)}$

Common Order Classes

- Order classes group “equivalently” efficient algorithms

  - O(1) – constant time! Input size doesn’t matter
  - O(lg n) – logarithmic time. Very efficient. E.g. binary search (after sorting)
  - O(n) – linear time
  - O(n lg n) – log-linear time. E.g. best sorting algorithms
  - O(n^2) – quadratic time. E.g. poorer sorting algorithms
  - O(n^3) – cubic time
  - ….
  - O(2^n) – exponential time. Many important problems, often about optimization

When Does this Matter?

- Size of input matters a lot!
  - For small inputs, we care a lot less
  - But what’s a big input?
    - Hard to know. For some algorithms, smaller than you think!

Order Classes Details

- What does the label mean? $O(n^2)$
  - Set of all functions that grow at the same rate as n^2 **or** more slowly
  - I.e. as efficient as any "n^2" or more efficient, but no worse
  - So this is an **upper-bound** on how inefficient an algorithm can be
- Usage: We might say: Algorithm A is $O(n^2)$
  - Means Algorithm A's efficiency grows like a quadratic algorithm **or** grows more slowly (*As good or better*)
- What about that other label, $\Theta(n^2)$?
  - Set of all functions that grow at **exactly** the same rate
  - *A more precise bound*

Big-O Notation Formal Definition

- If we want a general way to study the performance of an algorithm on data sets of arbitrary size we perform asymptotic analysis
- Through this analysis we develop an expression that links time ***t*** and size of input ***n***.
- This representation is called ***big-O notation***

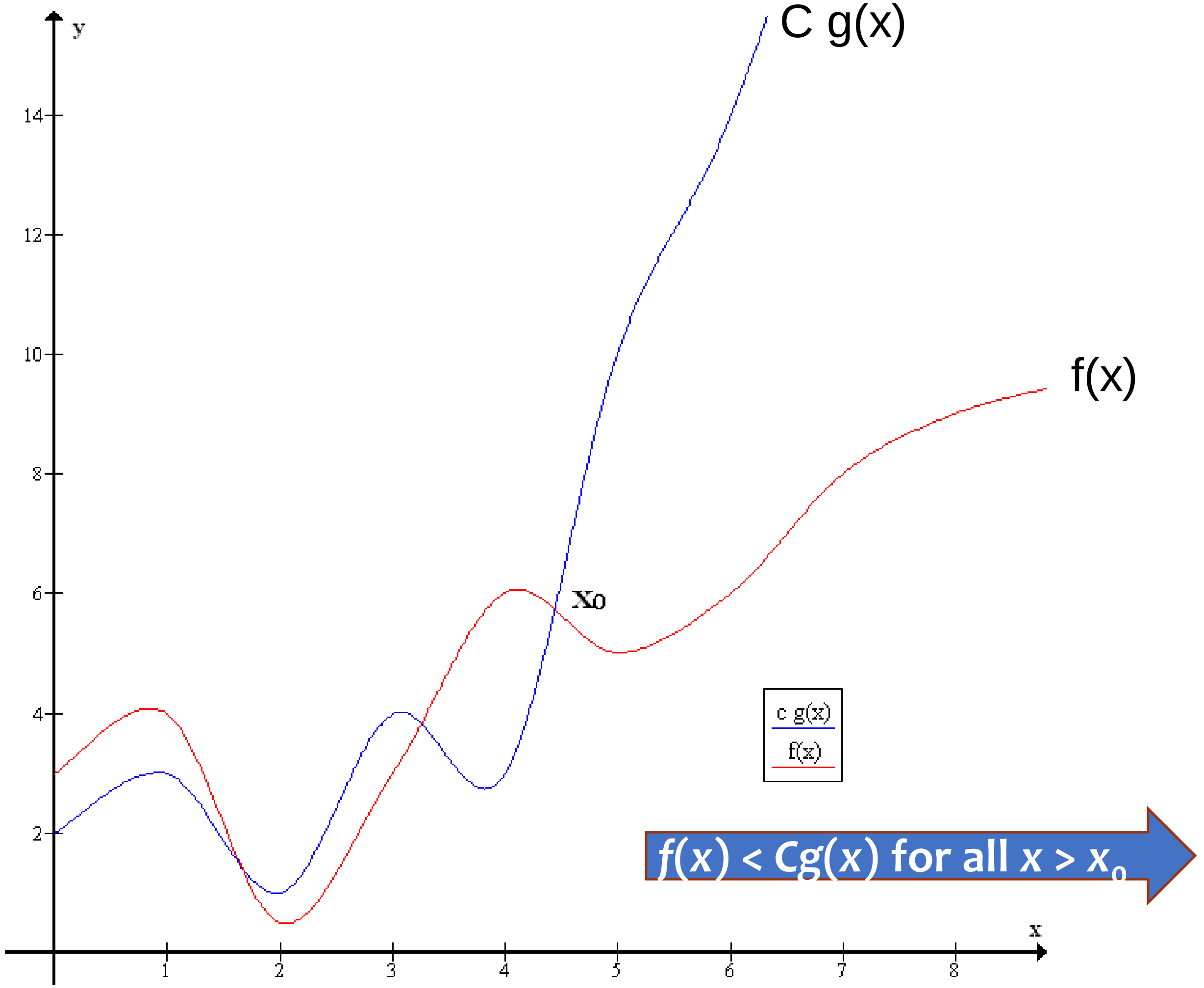
Big-O Notation Formal Definition

- Formally, the expression states that there are positive constants C and N_0 such that
  - if $t = O(f(n))$, then $0 \leq t \leq Cf(n)$ for all $n > N_0$

- This may sound confusing! But, it simply states that an algorithm's computing time grows no faster than (i.e., **is bounded by**) a constant times a function of the form f(n)
- When one algorithm is of a **lower** order than another, it is **asymptotically superior**

Big-O Notation Formal Definition

- Let f and g be two functions defined on some subset of the real numbers
- $f(x) = O(g(x))$ *as* $x \to \infty$ *(the last part often left unstated)*
- The above holds if and only if there is a positive constant C such that for all sufficiently large values of x, $f(x)$ is at most C multiplied by the absolute value of $g(x)$. That is, $f(x) = O(g(x))$ if and only if there exists a positive real number C and a real number x_0 such that
- $|f(x)| \leq C|g(x)|$ for all $x \geq x_0$



- $f(x) \in O(g(x))$ as there exists $C > 0$ (e.g., $C = 1$) and x_0 (e.g., $x_0 = 5$) such that $f(x) < Cg(x)$ whenever $x > x_0$

Big-O is a Good Estimate

- For large values of N, Big-O is a good approximation for the running time of a particular algorithm. The table below shows the observed times and the estimated times

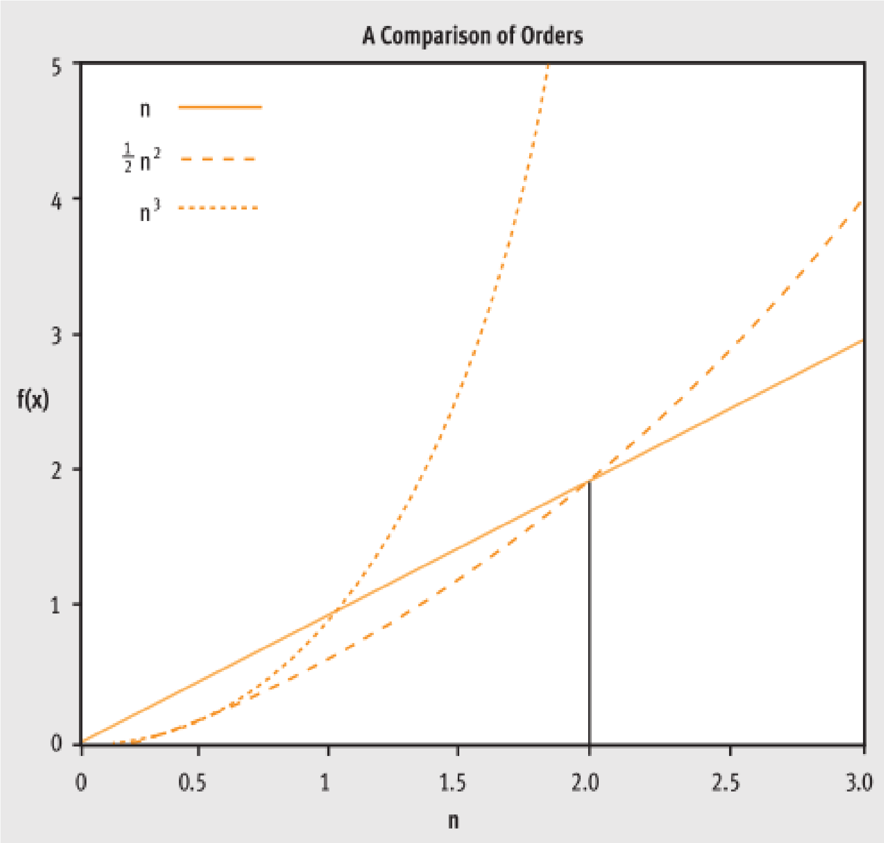
N	Observed time	Estimated time	Error
10	0.12 msec	0.09 msec	23%
20	0.39 msec	0.35 msec	10%
40	1.46 msec	1.37 msec	6%
100	8.72 msec	8.43 msec	3%
200	33.33 msec	33.57 msec	1%
400	135.42 msec	133.93 msec	1%
1000	841.67 msec	835.84 msec	1%
2000	3.35 sec	3.34 sec	< 1%
4000	13.42 sec	13.36 sec	< 1%
10,000	83.90 sec	83.50 sec	< 1%

Asymptotically Superior Algorithm

- If we choose an **asymptotically superior algorithm** to solve a problem, we will not know exactly how much time is required, but we know that as the problem size increases there will always be a point beyond which the lower-order method takes less time than the higher-order algorithm
- Once the problem size becomes sufficiently large, the asymptotically superior algorithm always executes more quickly
- The next figure demonstrates this behavior for algorithms of order $O(n)$, $O(n^2)$, and $O(n^3)$

Asymptotically Superior Algorithm

- For small problems, the choice of algorithms is not critical – in fact, the $O(n^2)$ or $O(n^3)$ may even be superior!
- However, as n grows large (larger than **2.0** in this case) the **O(n)** algorithm <u>always</u> has a superior running time and ***improves as n increases***



Summary

- Big-O notation, describes the **asymptotic behavior** of algorithms on **large problems**

- It is the fundamental technique for describing the **efficiency** properties of algorithms

Summary

- **Common complexity classes:**
  - $O(1)$ – constant time
  - $O(\lg n)$ – logarithmic time
  - $O(n)$ – linear time
  - $O(n \lg n)$ – log-linear time
  - $O(n^2)$ – quadratic time
  - $O(n^3)$ – cubic time
  - ….
  - $O(2^n)$ – exponential time

Increasing Complexity

O(1) – Constant time

- The algorithm requires a fixed number of steps regardless of the size of the task (input)
- **Examples**
- Push and Pop operations for a stack data structure (size n)
- Insert and Remove operations for a queue
- Conditional statement for a loop
- Variable declarations
- Assignment statements

O(log n) – Logarithmic time

- Operations involving dividing the search space in ***half*** each time (taking a list of items, cutting it in half repeatedly until there's only one item left)
- **Examples**
- Binary search of a sorted list of n elements
- Insert and Find operations for binary search tree (BST) with n nodes

O(n) – Linear time

- The number of steps increase in proportion to the size of the task (input)
- **Examples**
- Traversal of a list or an array… (size n)
- Sequential search in an unsorted list of elements (size n)
- Finding the max or min element in a list

O(n lg n) – Log-linear time

- Typically describing the behavior of more advanced sorting algorithms
- **Examples**
- Quicksort
- Mergesort

$O(n^2)$ – Quadratic time

- For a task of size 10, the number of operations will be 100
- For a task of size 100, the number of operations will be 100x100 and so on…
- **Examples**
- A selection sort of n elements
- Finding duplicates in an unsorted list of size n

  - *Think: doubly nested loops*

$O(a^n)$ $(a>1)$ – Exponential time

- Many interesting problems fall into this category…
- **Examples**
- Recursive Fibonacci implementation
- Towers of Hanoi
- Generating all permutations of n symbols
- … many more!

Code Examples

- Review the document
- “CS 5012 - Code Examples - Asymptotic Analysis.pdf”

Reminder: Readings

- Chapter 5 of MSD text (see Collab)
  - Except Section 5.5 on recursion
- Chapters 1, 2 & 3 of Algorithms text