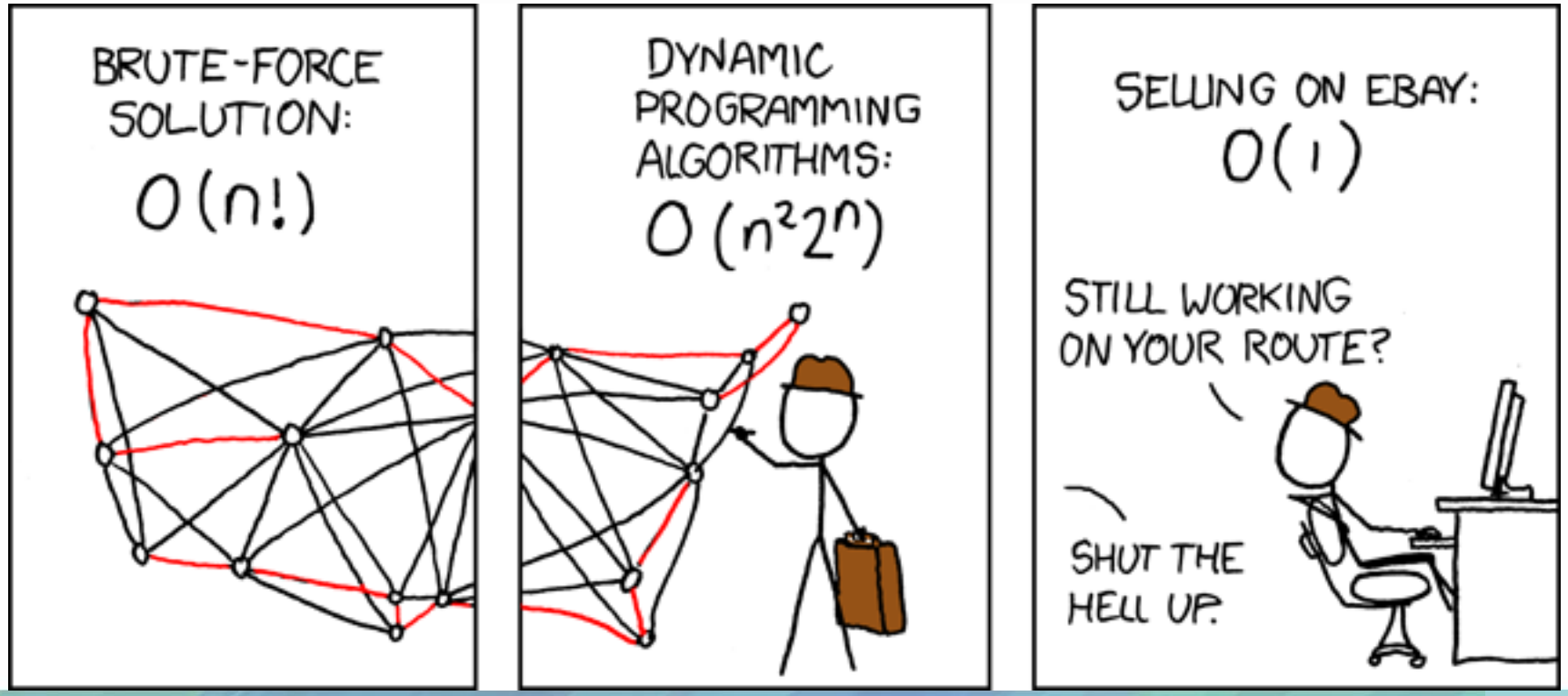
CS2110: SW Development Methods

Analysis of Algorithms

- Reading: Chapter 5 of MSD text
  - Except Section 5.5 on recursion (next unit)

Announcements

- **Homework #4** (GUIs) – due: 11:30pm on April 1, 2020
- **Exam 2** – moved to April 3, 2020
- **Weekly Quiz** – out Friday, due 11:30pm Sunday as usual

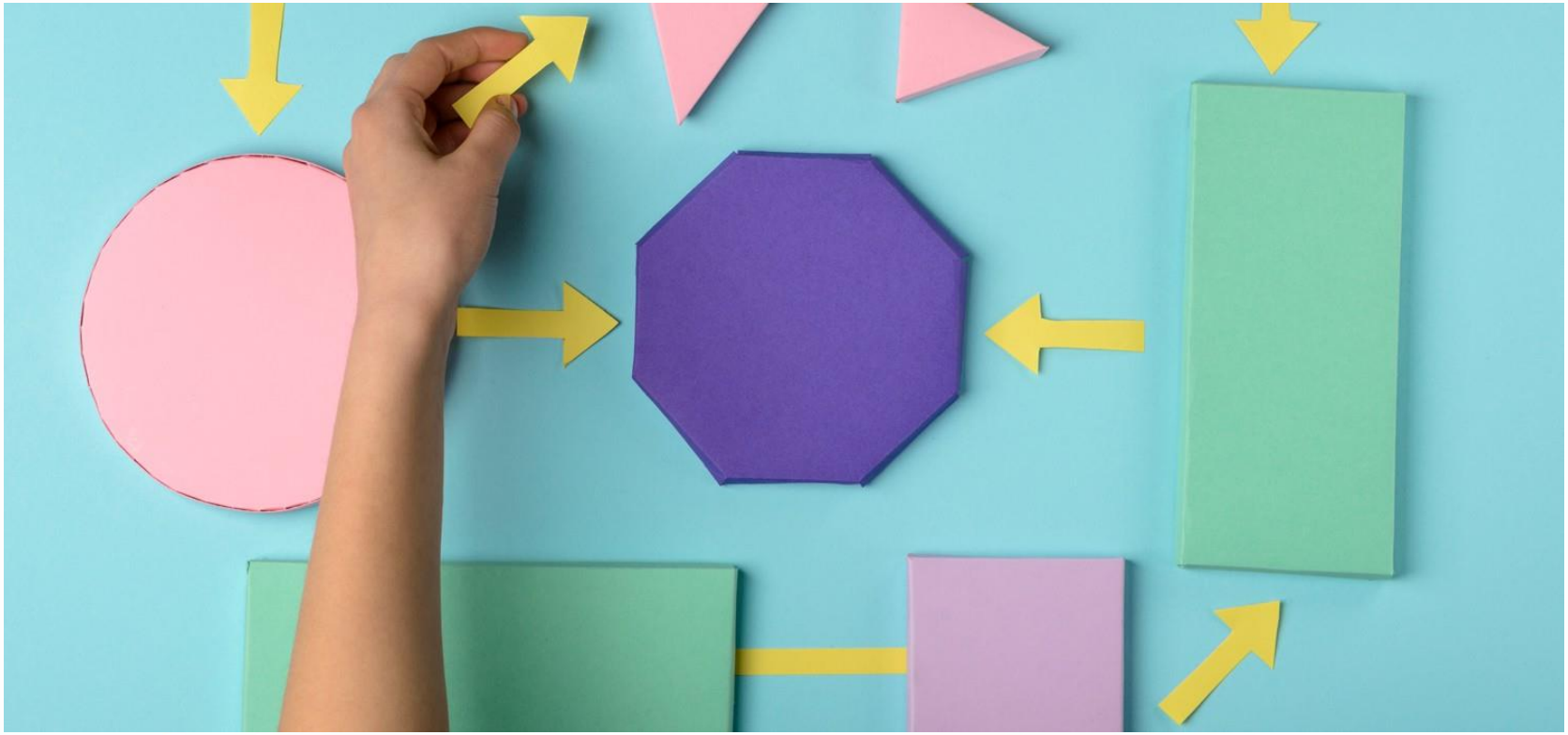


Goals for this Unit

- Begin a focus on data structures and algorithms
- Understand the nature of the performance of algorithms
- Understand how we measure performance
  - Used to describe performance of various data structures
- Begin to see the role of algorithms in the study of Computer Science
- Note: CS2110 starts the study of this
  - Continued in CS2150 and CS4102

Algorithm

- An **algorithm** is a detailed step-by-step method for solving a problem
- Computer algorithms, but other kinds too! [*Such as?*]



Algorithm

- An **algorithm** is a detailed step-by-step method for solving a problem
- Computer algorithms, but other kinds too! [*Such as?*]
- *Properties of algorithms*
  - Steps are **precisely stated**
    - No ambiguity
    - Cannot be interpreted in more than one way
  - **Deterministic:** behaves the same way (based on inputs, previous steps)
  - **Terminates:** the end is clearly defined
  - Other properties: correctness, generality, efficiency

How Do We Compare the Efficiency of Implemented Algorithms?

Core question we would like to address today

What things are we comparing?

What do we mean by *efficiency*?

Abstract Data Types

- **Data Structures**: A logical relationship among data elements designed to support specific data manipulation functions
  - Concrete: defined as an implementation
  - Examples: ArrayList, HashSet, trees, tables, stacks, queues
- **Abstract Data Types (ADT)**: a *model* of data items stored and a *set of operations* that manipulate the data model
  - Abstract: no implementation implied (**data abstraction**)
  - Examples: List, Set, … (think Java interfaces)
  - A particular data structure implements an ADT and defines *how* it is implemented

Example ADTs

- Example: ADT List can be implemented by an array, an ArrayList, a LinkedList, …

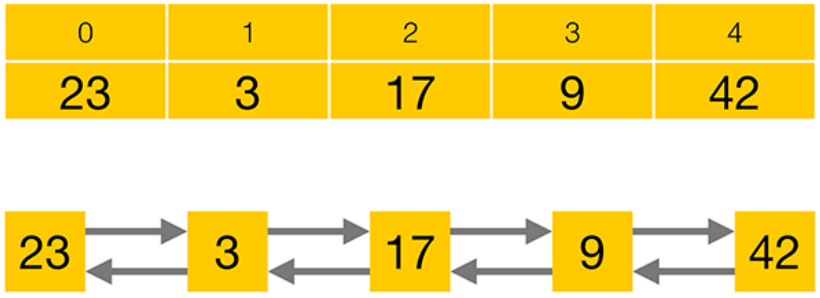
- Example: ADT Priority Queue can be implemented by an array, an ArrayList, or a Binary Heap (base structure is a binary *tree*)

ADTs in Real Life

- Java *interfaces* roughly match ADTs
  - Interfaces don't have data fields, *but*…
  - Interfaces are good abstractions
  - Inheritance of an interface is important for ADTs (data structures will implement an ADT)

ADTs

- ADTs define **operations** and a given data structure implements them
  - Think and design using ADTs, then code using data structures
  - There may more than one data structure that implements the ADT we need – *so how do we decide?*
    - Compare the Advantages and disadvantages
      - Efficiency / performance is often a major consideration
      - Example: ArrayList vs. LinkedList



ADTs

- **So …** how do we compare efficiency of implementations?
- **Answer**: We compare the algorithms that implement the operations

- E.g.:
  the remove method of an **ArrayList**
  vs.
  the remove method of a **LinkedList**

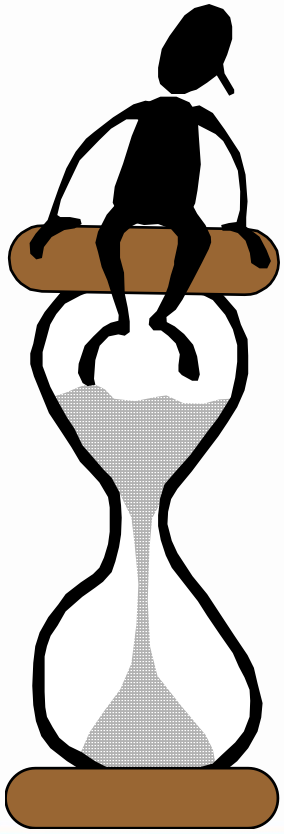
Let's Define Efficiency…

- 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 that 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?



How about…

MacBook 2018 (6-core Intel Core i7 processors up to 2.9GHz with Turbo Boost up to 4.8GHz) vs
Rasberry Pi (4× ARM Cortex-A53, 1.2GHz) ???

Not a fair comparison!

(This is called **BENCHMARKING**, and is useful in certain circumstances.)

Considerations…

Algorithm A (implementing task "X") **Algorithm B (implementing task "X")**

- PC vs. MAC
- Python vs. Java
- Programmer 1 vs. Programmer 2
- Laptop vs. super computer
- MAC vs. MAC

- (Data) Input: 10 vs. Input: 100,000

- (Data) Input: 10,000 Sorted vs. Input: 10,000 Random

Why Not Just Time Algorithms?

- We want a measure of work that gives us a direct measure of the *efficiency* of the algorithm without introducing differences in:
  - Computer hardware
  - Programming language
  - Programmer skills (in coding the algorithm)
  - (Other implementation details)
  - The *size of the input*
  - The *nature of the input*
    - Best-case, worst-case, average
    - *E.g. searching a sorted vs. a randomized list*

Measuring Performance

- 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** (n) – *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

- Analysis of Algorithms: use mathematics as our tool for analyzing algorithm performance
  - Measure the algorithm itself, *its nature*
  - Not its implementation or its execution

- We need something to **count**!
  - Cost or number of steps is a function of input size n: e.g. for input size n, cost is $\boldsymbol{f(n)}$
  - Count all steps in an algorithm? (*Hopefully avoid this!*)

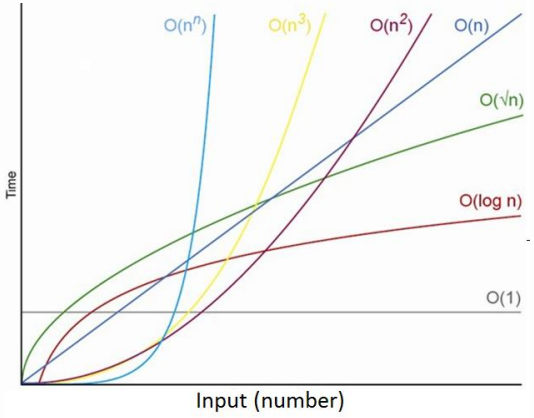
Counting Operations

- **Strategy**: choose one operation or section of code to count
  - Total work is always roughly proportional to how often that part is done
- So we'll just count:
  - An algorithm's "basic operation," or "**critical section**"

- Sometimes the basic operation is some action that is 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 (Characterizing the performance of an algorithm)

- **Algorithmic complexity** is concerned with how fast or slow a particular algorithm performs.
  - How long will a program run on an input?
  - How much space will it take?
  - Is the problem solvable?



- An understanding of algorithmic complexity provides programmers with insight into the efficiency of their code

Asymptotic Analysis

- **Asymptotic Analysis**: an estimate of time as a function of the input size, *n*, *as n gets large*

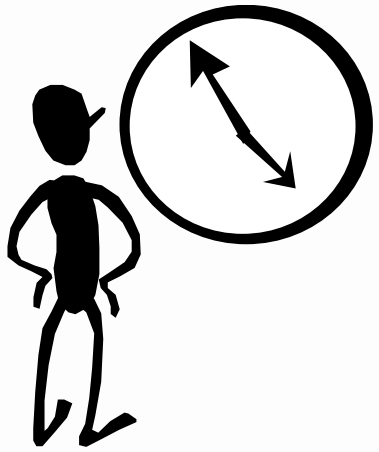
- Examples: [which is "better"?]
  - The standard *insertion* sort takes time T(n)
    - $T(n) = c_1 n^2 + k_1$, for some constants c_1 and k_1
  - *Merge* sort takes time T'(n)
    - $T'(n) = c_2 n log_2(n) + k_2$, for some constants c_2 and k_2

- **As n gets large:** it's only when n becomes large that differences become apparent

Asymptotic Analysis

- The **asymptotic behavior** of a function $f(n)$ refers to the growth of $f(n)$ as ***n* gets large**

- "**Asymptotic**" – how do things change as the input size n gets larger?
  - How **scalable** the algorithm is
    - how slow will it be on large inputs?
  - Rule of thumb: the **slower** the asymptotic growth rate, the *better* the algorithm

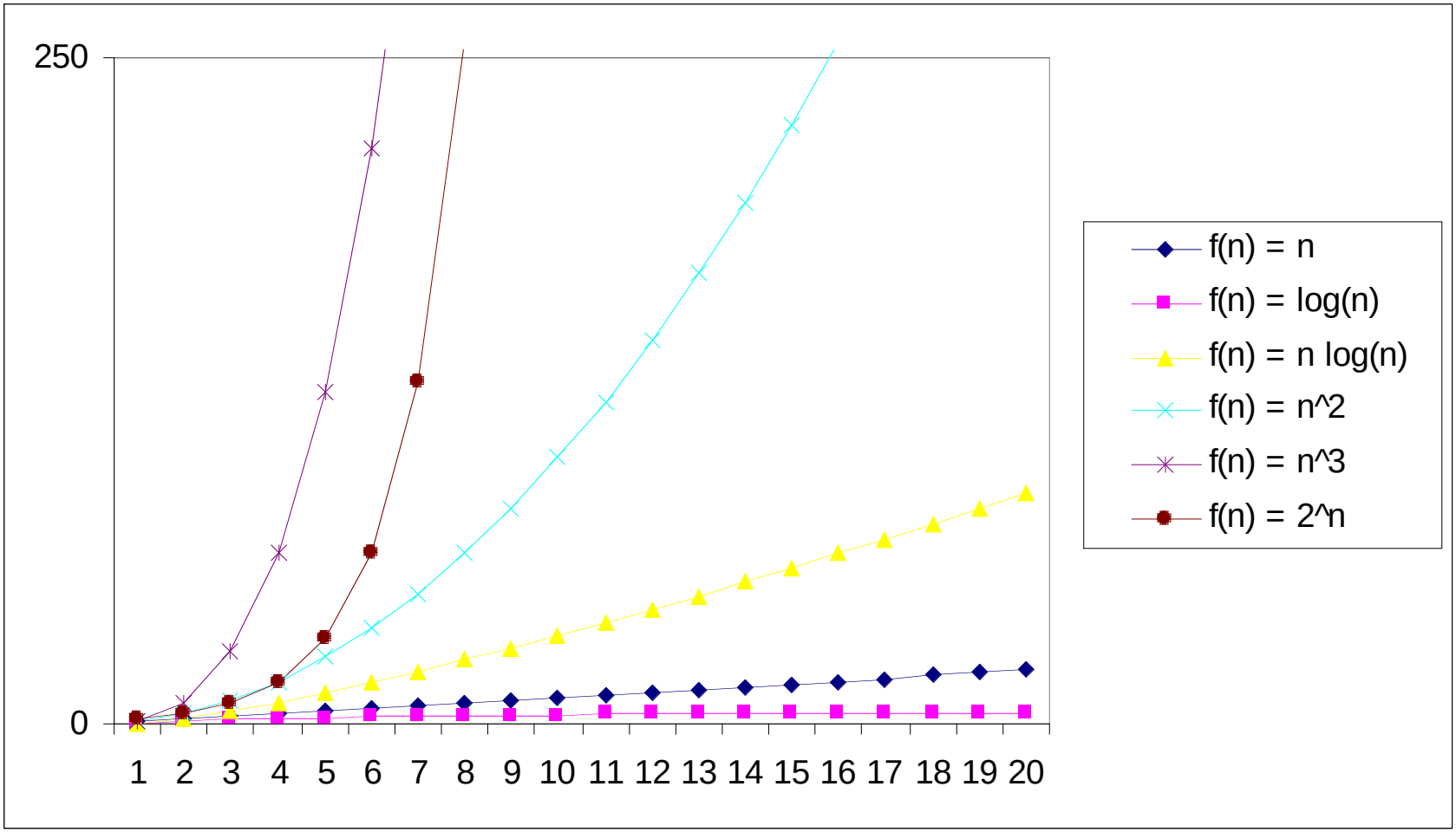


Asymptotic Analysis

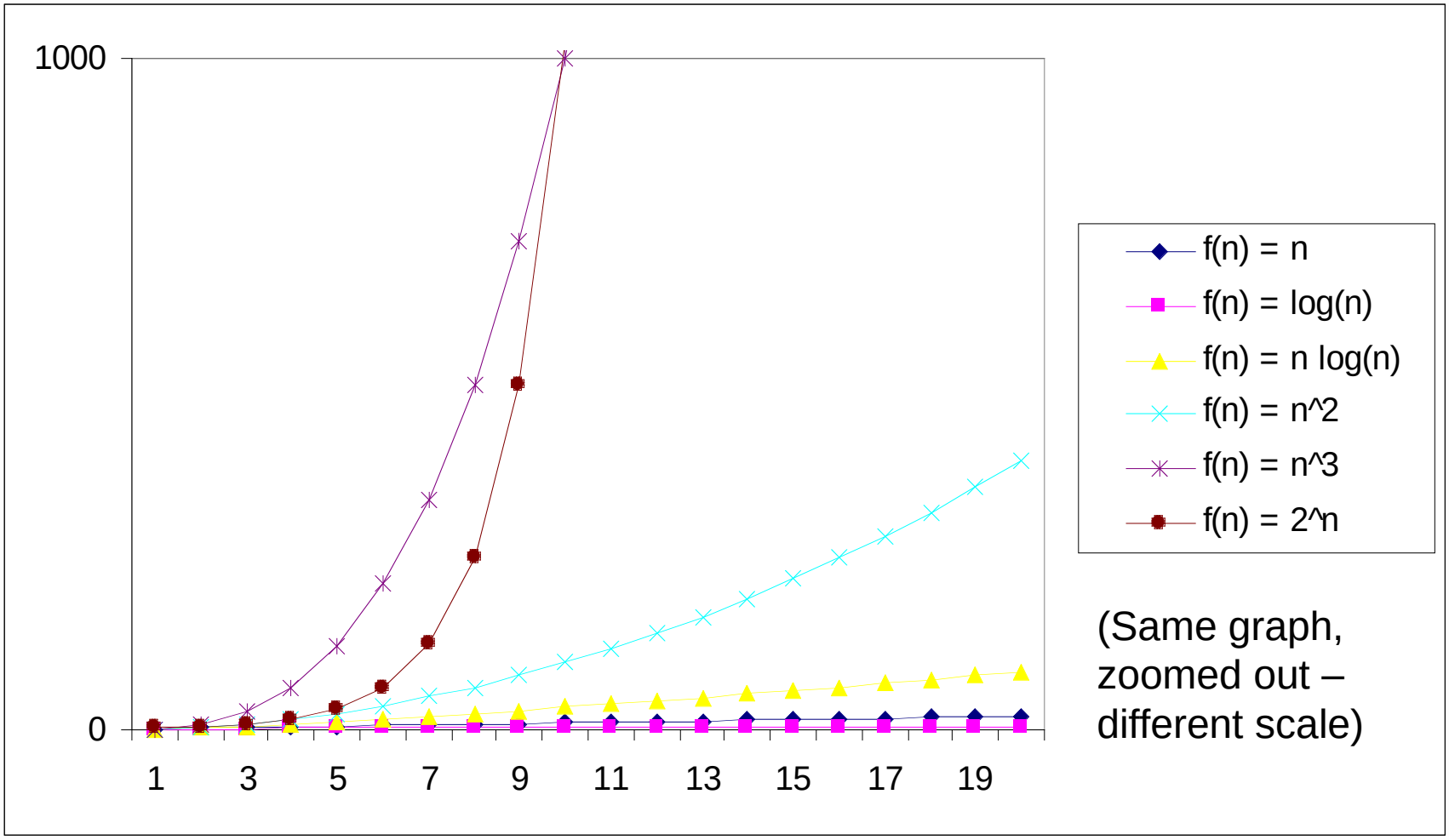
- **Linear-time** algorithm: $f(n) = dn + k$ for constants d, k
- **Quadratic-time** algorithm: $f(n) = cn^2 + q$ for constants c, q
- ***Which is better?*** The linear-time algorithm will always be *asymptotically better*!

- 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. However, the *linear algorithm* will always be better for sufficiently large inputs

Comparison of Growth Rates



Comparison of Growth Rates ("zoomed out")



(Same graph, zoomed out – different scale)

Time Complexity

- *Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity*
- For large inputs, are these functions really different?
  - $f(n) = 100n^2 + 50n + 7$
  - $f(n) = 20n^2 + 7n + 2$

  They are both quadratic functions

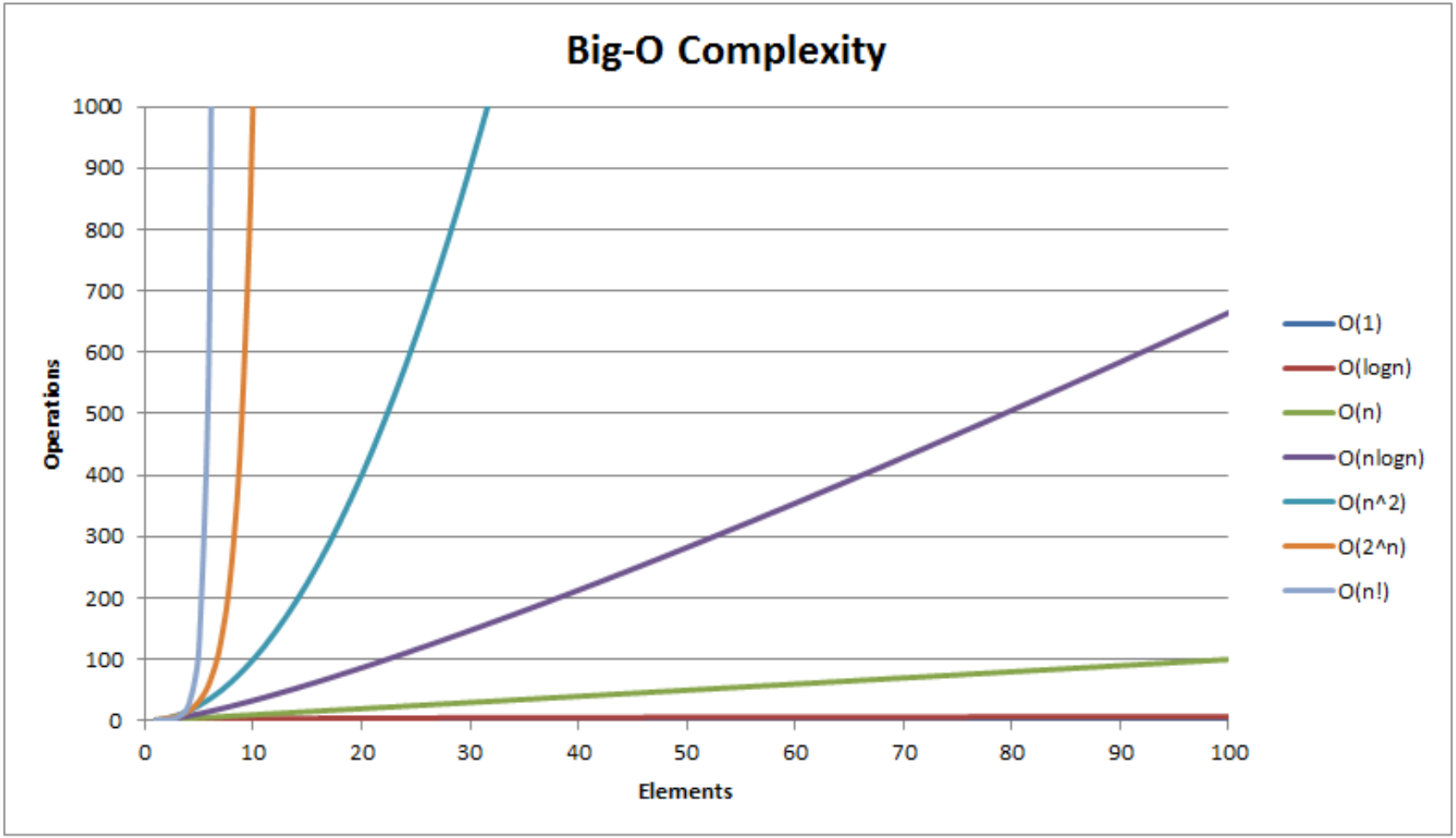
- **Order class:** a "label" for all functions with the same *highest-order term*
  - **$O(n^2)$** : ***Big-Oh*** notation [used more often]
  - **$\Theta(n^2)$** : ***Big-Theta*** notation

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. Constants are also eventually ignored
- As ***n*** gets large, the highest order term will dominate (*asymptotically*)

$f(n) = 100n^2 + 50n + 7$ ➔ Simply $O(n^2)$

Time Complexity : Common order classes



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 E.g. linear search
  - $O(n \lg n)$ – 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
  - $O(n!)$ – factorial time. E.g. all permutations of a string

Time Complexity: 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
  - 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 – as efficient as n^2, no worse and no better*

Order Class Details
[Big-Oh; Big-Theta; Big-Omega]

$O(n^2)$: **upper-bound** on how inefficient an algorithm can be

- Set of all functions that are at most n^2
- Functions that grow the same rate or slower than n^2
- As efficient as n^2, but no worse

$\Theta(n^2)$: **tight-bound** on how inefficient an algorithm can be

- Set of all functions that are exactly n^2
- As efficient as n^2, no worse and no better

$\Omega(n^2)$: **lower-bound** on how inefficient an algorithm can be

- Set of all functions that are at least n^2
- As efficient as n^2, but no better

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

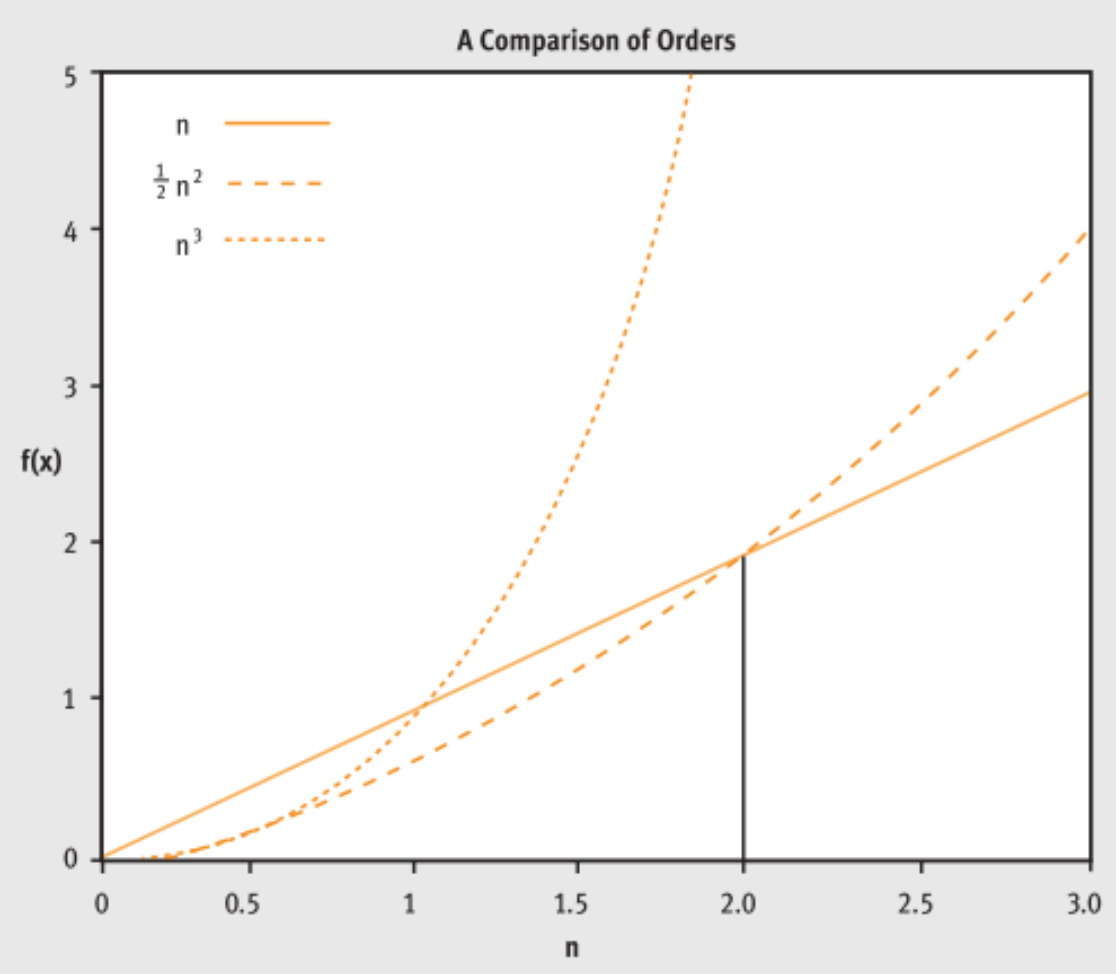
N	Observed time	Estimated time	Error
10	0.12 msec	0.09 msec	23%
20	0.39 msec	0.35 msec	10%
40	1.46 msec	1.37 msec	6%
100	8.72 msec	8.43 msec	3%
200	33.33 msec	33.57 msec	1%
400	135.42 msec	133.93 msec	1%
1000	841.67 msec	835.84 msec	1%
2000	3.35 sec	3.34 sec	< 1%
4000	13.42 sec	13.36 sec	< 1%
10,000	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 always has a superior running time and ***improves as n increases***

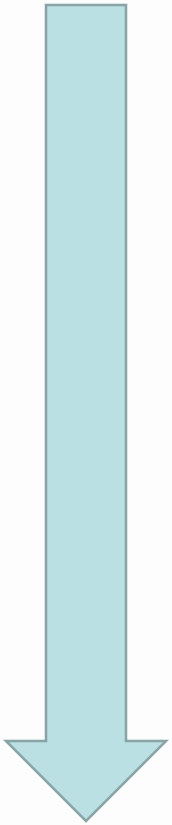


Summary

1. We need an accurate way to measure **algorithm efficiency,** which is independent of hardware or external to the program factors
2. **Time complexity** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the **size of the input**
3. **Asymptotic analysis** (using *Big-O*) is the fundamental technique for describing the efficiency properties of algorithms (time complexity)
4. **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

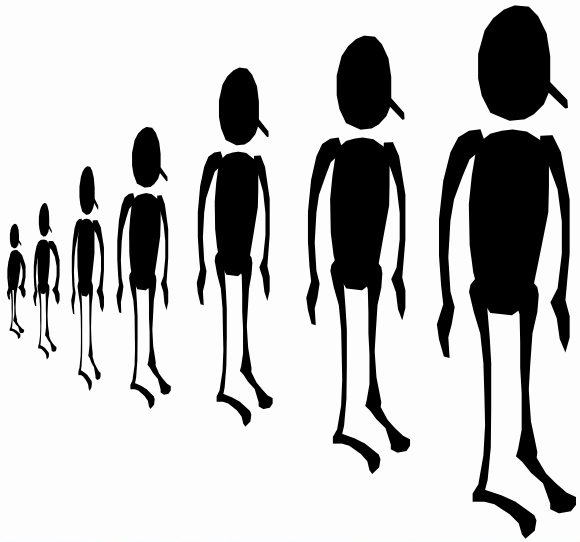
Examples of algorithms in various complexity classes

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

- **Some of the best sorting algorithms**
  - 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

- Some poorer sorting algorithms
  - Selection sort of n elements, Insertion sort …
- 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 … $O(2^n)$

Examples

- Recursive Fibonacci implementation
- Towers of Hanoi
- Generating all permutations of n symbols
- … many more!