Tries, Asymptotic Analysis
Summer 2021

Instructor: Kristofer Wong

Teaching Assistants:

Alena Dickmann	Arya GJ	Finn Johnson
Joon Chong	Kimi Locke	Peyton Rapo
Rahul Misal	Winston Jodjana	

Announcements (1 of 2)

- Lecture recordings are linked from the website and also can be found in the Zoom tab on Canvas.
  - If not up within 5 hours of lecture, make an Ed post please!
- Ed: Feel free to post anonymously, but this will not contribute to participation.
  - Private posts should only be made regarding bugs / specifics
  - People have similar questions and we want to make sure everyone gets the answer!
- Project 1 repos created this morning! We will be reaching out to introduce you and your partner soon

Announcements (2 of 2)

- Textbook out of stock at bookstore 
  - Recommend used 2nd edition ~$15 on amazon
- Lecture Questions Doc
- Office Hours Queue
- Fun ~~Office~~ Hours

Lecture Outline

- **ADT Summary**
- The Trie Data Structure
- Algorithms – How do we compare them??
- Analyzing Code
- Asymptotics
- Big-Oh

Summary: Common ADTs

List ADT. A collection storing an ordered sequence of elements.

- Each element is accessible by a zero-based index
- A list has a size defined as the number of elements in the list
- Elements can be added to the front, back, *or any index in the list*
- Optionally, elements can be removed from the front, back, *or any index in the list*

Stack ADT. A collection storing an ordered sequence of elements.

- A stack has a size defined as the number of elements in the stack
- Elements can only be added and removed from the top ("LIFO")

Queue ADT. A collection storing an ordered sequence of elements.

- A queue has a size defined as the number of elements in the queue
- Elements can only be added to one end and removed from the other ("FIFO")

Summary: Common ADTs

Dictionary ADT. A collection keys, each associated with a value

- A dictionary has a size defined by the number of elements in the dictionary (key/value pairs)
- You can add and remove key/value pairs, but the keys must be unique
- Each value is accessible by its key via a "find" or "contains" operation

Set ADT. A collection keys.

- A set has a size defined by the number of elements in the set
- You can add and remove keys, but the keys must be unique
- Each value is accessible by its key via a "find" or "contains" operation

- Check out the java interface documentation (remember, Dictionary in Java is a Map)
- Geeks4geeks has great visualizations as well

Lecture Outline

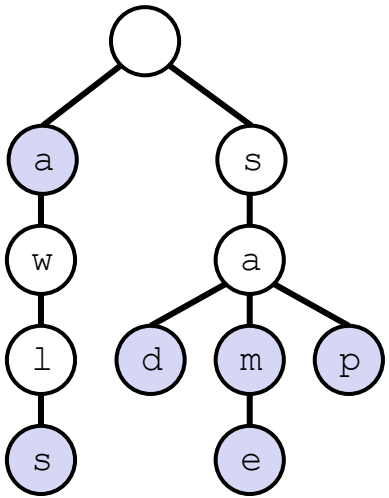
- ADT Summary
- **The Trie Data Structure**
- Algorithms – How do we compare them??
- Analyzing Code
- Asymptotics
- Big-Oh

The Trie: A Specialized Data Structure

- What ADT does this implement…? Set!
- Tries view its keys as:
  - a **sequence of characters**
  - some (hopefully many!) sequences share common prefixes

- sap
- sad
- awls
- a
- same
- sam

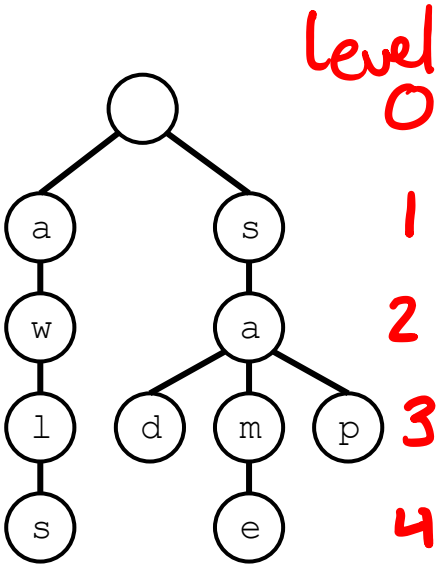
Set ADT



Trie

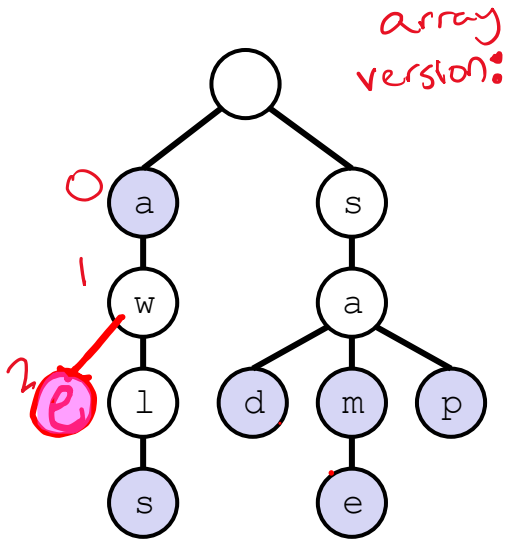
Trie: An Introduction

- Each level of the tree represents an index in the string
  - Children at that level represent possible characters at that index
- This abstract trie stores the set of strings:
  - `awls, a, sad, same, sap, sam`
- How to deal with `a` and `awls`?
  - Mark which nodes *complete* a string (shown in purple)



Trie Insertion

What happens when a user calls `insert`("awe")



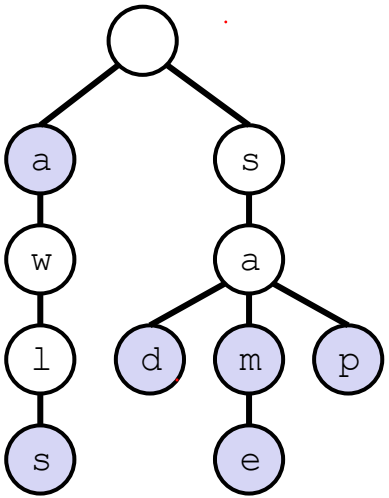
array version:

a	w	e
0	1	2

Searching in Tries

Two ways to have a **search miss:**

1. If we fall off the tree
2. If the final node isn't marked (not a key, denoted in purple)



Input String	Fall Off? / Is Key?	Result
contains("sam")	hit / purple	True
contains("sa")	hit / white	False
contains("a")	hit / purple	True
contains("saq")	fell off / n/a	False

Keys as "a sequence of characters"

- Most dictionaries treat their keys as an "atomic blob": you can't disassemble the key into smaller components

- Tries take the opposite view: keys are a **sequence of characters**
  - `String`s are made of `Character`s

- But "characters" don't have to come from the Latin alphabet
  - `Character` includes most Unicode codepoints (eg, 🥺,👉,👈)
  - `List<E>`
  - `byte[]`

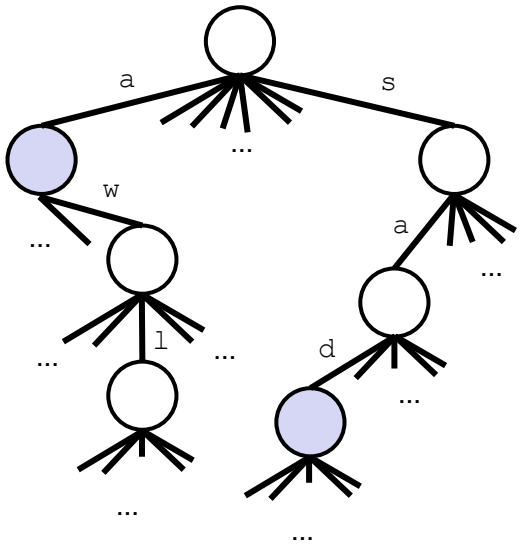
Definitions

- Tries are defined by 2 types:
  - An "alphabet": the domain of the characters
  - A "key": a sequence of "characters" from the alphabet
- 3, actually:
  - Remember how Sets can be thought of as Dictionaries without values?
  - Tries can also be defined by a third, arbitrary, "value" type

Gradescope Activity!!

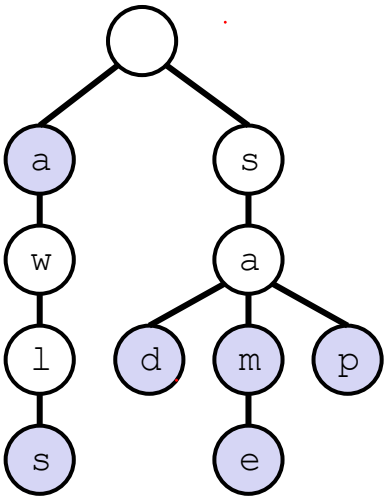
Does the structure of a trie depend on the order in which strings are inserted?

A. Yes
B. No
C. I'm not sure



Trie Deletion

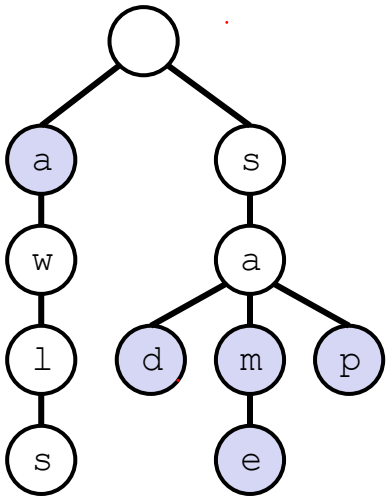
What happens a user calls `delete`("awls")?



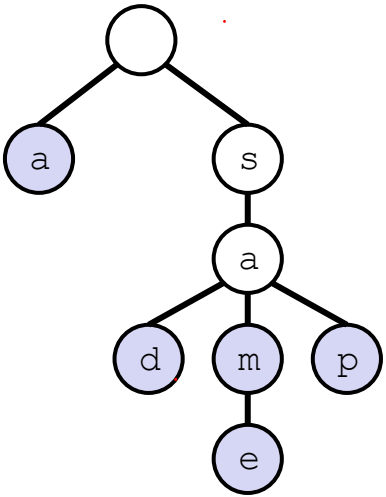
Two Deletion Algorithms

What are the tradeoffs of each?

`delete`("awls")



Lazy Deletion



Normal Deletion

Two Deletion Algorithms

What are the tradeoffs of each?

`delete`("awls")

Lazy Deletion

Better if:

- You want a faster deletion
- Insertion is common
- Reinserting deleted keys is expected
- Easier to code (!?)

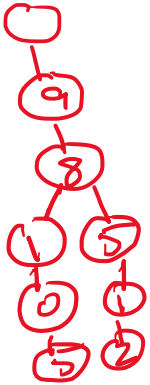
Normal Deletion

Better if:

- Find/prefix operations are common
- You need to manage space more efficiently

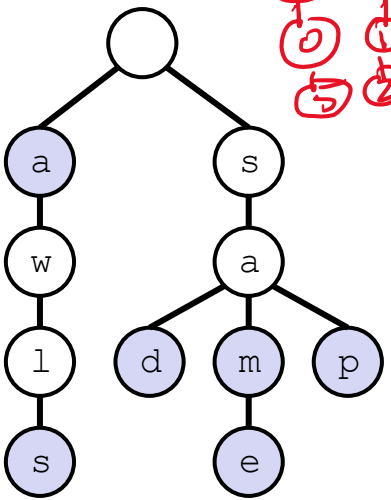
So… Why?

Zip Codes:
98105
98512



- The main appeal of tries is prefix matching!

- **Common Specialized Algorithms**
  - **Longest prefix**
    - `longestPrefixOf("sample")`
    - Want: `{"sam"}`
  - **Prefix match**
    - `findPrefix("sa")`
    - Want: `{"sad", "sam", "same", "sap"}`



Lecture Outline

- ADT Summary
- The Trie Data Structure
- **Algorithms – How do we compare them??**
- Analyzing Code
- Asymptotics
- Big-Oh

Describing Algorithms: What Do We Care About?

- Correctness:
  - Does the algorithm do what is intended
- Performance:
  - Speed **time complexity**
  - Memory **space complexity**
- Other attributes:
  - Clarity, security, … equity?!?!

- Why analyze performance?
  - To make good design decisions
  - Enable you to examine an algorithm (or code) and identify bottlenecks

A: How Should We Describe An Algorithms' Performance?

- Uh, why NOT just run the program and time it??
  - Too much ***variability***; not reliable or ***portable***
    - Hardware: processor(s), memory, etc.
    - Firmware: OS, Java version, libraries, drivers
    - Other: implementation-specific quirks, other programs running, …
  - Choice of input
    - (Non-exhaustive) testing may ***miss*** worst-case input
    - Benchmarks don't ***describe*** or ***predict*** the relationship between input sizes

- Often want to evaluate an ***algorithm***, not an ***implementation***

An ***algorithm*** is more ***performant*** than another when, for sufficiently large inputs, it runs in less time *(our focus)* or less space than the other

Comparing Algorithms (1 of 2)

- When is one ***algorithm*** (not ***implementation***) better than another?
  - Various possible answers (clarity, security, ...)
  - But a big one is ***performance***: for sufficiently large inputs, runs in less time (our focus) or less space than the other
- Large inputs (n) because probably any algorithm is "plenty good" for small inputs (if *n* is 10, probably anything is fast enough)

Comparing Algorithms (2 of 2)

Want a model that's:

- ***Independent*** of CPU speed, programming language, coding tricks, etc.
- General and rigorous, ***complementary*** to "coding it up and timing it on some test cases": can do analysis before coding!
- ***Descriptive of large inputs***; most algorithms are "good enough" for small inputs
  - If *n* is 10, probably anything is fast enough
  - But what consider to be "large"?

Lecture Outline

- ADT Summary
- The Trie Data Structure
- Algorithms – How do we compare them??
- **Analyzing Code**
- Asymptotics
- Big-Oh

Analyzing Code: Our Model (1 of 2)

- We abstract away the computer by counting:
  - "elements" (space complexity)
  - "operations" (time complexity)

- Remember: "Independent of CPU, programming language, coding tricks, etc."

- Basic ***elements*** take "some amount of" ***constant space***
  - Integers in an array
  - Nodes in a linked list
  - Etc.
  - (This is an *approximation of reality*: a very useful "lie".)

Analyzing Code: Our Model (2 of 2)

- Basic ***operations*** take "some amount of" ***constant time***
  - Arithmetic (+,-,*,/, %)
  - Assignment (=)
  - Access one Java field or array index (Object.field, arr[i])
  - Etc.
  - (Again, this is an *approximation of reality*)

Consecutive statements	Sum of time of each statement
Loops	Num iterations * time for loop body
Recurrence	Solve recurrence equation
Function Calls	Time of function's body
Conditionals	Time of condition + time of {slower/faster} branch

Which Branch To Analyze?

- Case Analysis != Asymptotic Analysis

- We generally talk about two cases:
  - ***Worst-case complexity***: max # steps algorithm takes on "most challenging" input of size N
  - ***Best-case complexity*****:** min # steps algorithm takes on "easiest" input of size N
  - (there are other cases, but they're harder to reason about)

- Unless otherwise stated, we usually refer to the ***worst case***
  - There are uses for other analyses
  - For now, we'll analyze the ***slower branch***

Examples: From Code to Our Model

```
b = b + 5
c = b / a
b = c + 100
```

6

```
for (i = 0; i < n; i++) {
   sum++;
}
```

$1 + \sum_{i=0}^{n-1} 5 = 5n + 1$

```
if (j < 5) {
   sum++;
} else {
   for (i = 0; i < n; i++) {
      sum++;
   }
}
```

$1 + \sum_{i=0}^{n-1} 5$

Another Example

```
int coolFunction(int n, int sum) {
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      sum++;
    }
  }
  print "This program is great!";
  for (i = 0; i < n; i++) {
    sum++;
  }
  return sum
}
```

$1 + \left(\sum_{i=0}^{n-1}\left(\sum_{j=0}^{n-1} 5\right) + 3\right)$

$= 1 + n(5n+3)$

$1 + \sum_{i=0}^{n} 5 = 5n+1$

$\text{Total} \approx 5n^2 + 8n + 4$

Example Problem

Find an integer in a sorted array

2	3	5	16	37	50	73	75	126

```
// Requires arr to be sorted
// Returns whether k is in array
boolean findSorted(int[] arr, int k) {
   ???
}
```

Example Solution: Linear Search

Find an integer in a sorted array

2	3	5	16	37	50	73	75	126

```
// Requires arr to be sorted
// Returns whether k is in array
boolean findSorted(int[] arr, int k) {
  for(int i=0; i < arr.length; ++i) {
    if(arr[i] == k)
      return true;
    else if(arr[i] > k)
      return false;
  }
  return false;
}
```

In this case: $n = arr.length$

Best k: 2

Worst k: 7, 126

$\sum_{i=0}^{n-1} 8 = 1 + 8 \cdot arr.length$

Lecture Outline

- ADT Summary
- The Trie Data Structure
- Algorithms – How do we compare them??
- Analyzing Code
- **Asymptotics**
- Big-Oh

Analyzing Large Inputs (1 of 2)

- "Binary search is $O(\log n)$ and linear is $O(n)$ "
  - But which algorithm is faster?
  - Depending on ***specific case***, ***constant factors***, and ***size of n,*** linear search could be faster!

- What is a constant factor?
  - How *many* assignments, additions, etc. for each n

- Size of n:
  - Any Data Structure or Algorithm's behavior can vary for any finite n
  - Remember: "Descriptive of large inputs"
  - "Large": $n \to \infty$

** slide 9*

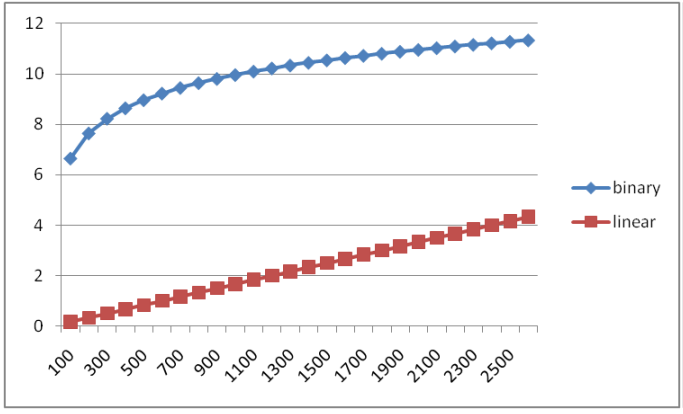
Analyzing Large Inputs (2 of 2)

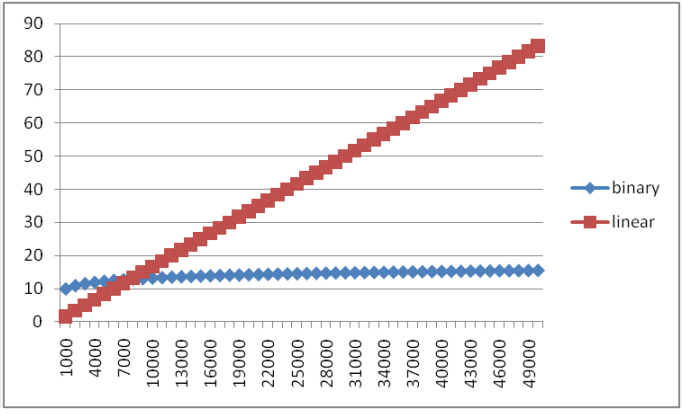
- How formalize the idea of how an algorithm behaves as $n \to \infty$?
  - There exists some n_0 such that for all $n > n_0$ ***binary search "wins"***

- Let's play with a couple plots to get some intuition…

Example: Binary Search vs Linear Search

- Let's "help" linear search "win"
  - Run it on a computer 100x as fast (say 2018 model vs. 1990)
  - Use a new compiler/language that is 3x as fast
  - Be a clever programmer to eliminate half the work
  - Each iteration is 600x as fast as in binary search





When we're dealing with infinity, constants and lower-order terms don't meaningfully add to the final result

Intuitive Simplifications

- When we're dealing with infinity, constants and lower-order terms don't meaningfully add to the final result

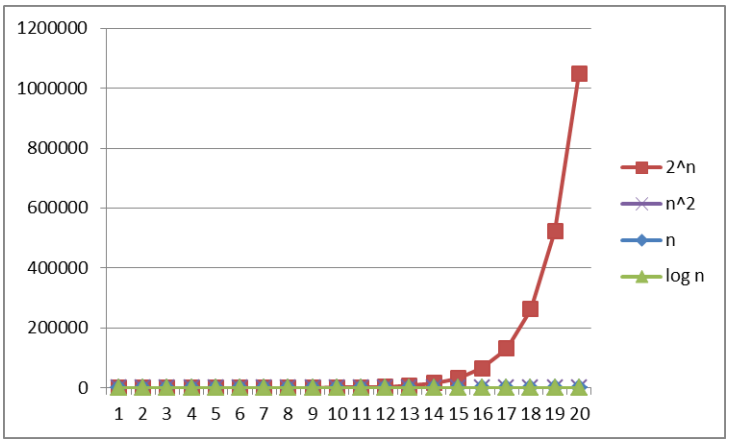
- (1) Eliminate lower-order terms
  - $6 + \frac{1}{2}N^2 + \frac{3}{2}N + 1 + \frac{1}{2}N^2 + \frac{1}{2}N + \frac{1}{2}N^2 - \frac{1}{2}N + N^2 + N$
  - $6 + \frac{1}{2}N^2 + \frac{3}{2}N + 1 + \frac{1}{2}N^2 + \frac{1}{2}N + \frac{1}{2}N^2 - \frac{1}{2}N + N^2 + N$
  - $\frac{5}{2}N^2$

- (2) Ignore multiplicative constants
  - $\frac{5}{2}N^2$
  - N^2

Logarithms and Exponents

- Definition: $\log_2 x = y$ if $x = 2^y$
  - Note: since so much is binary in CS, `log` (typically `lg`) almost always means $\log_2$

- Just as exponents grow *very* quickly, logarithms grow *very* slowly
  - So, $\log_2$ 1,000,000 = "a little under 20"



Log base doesn't matter (much)

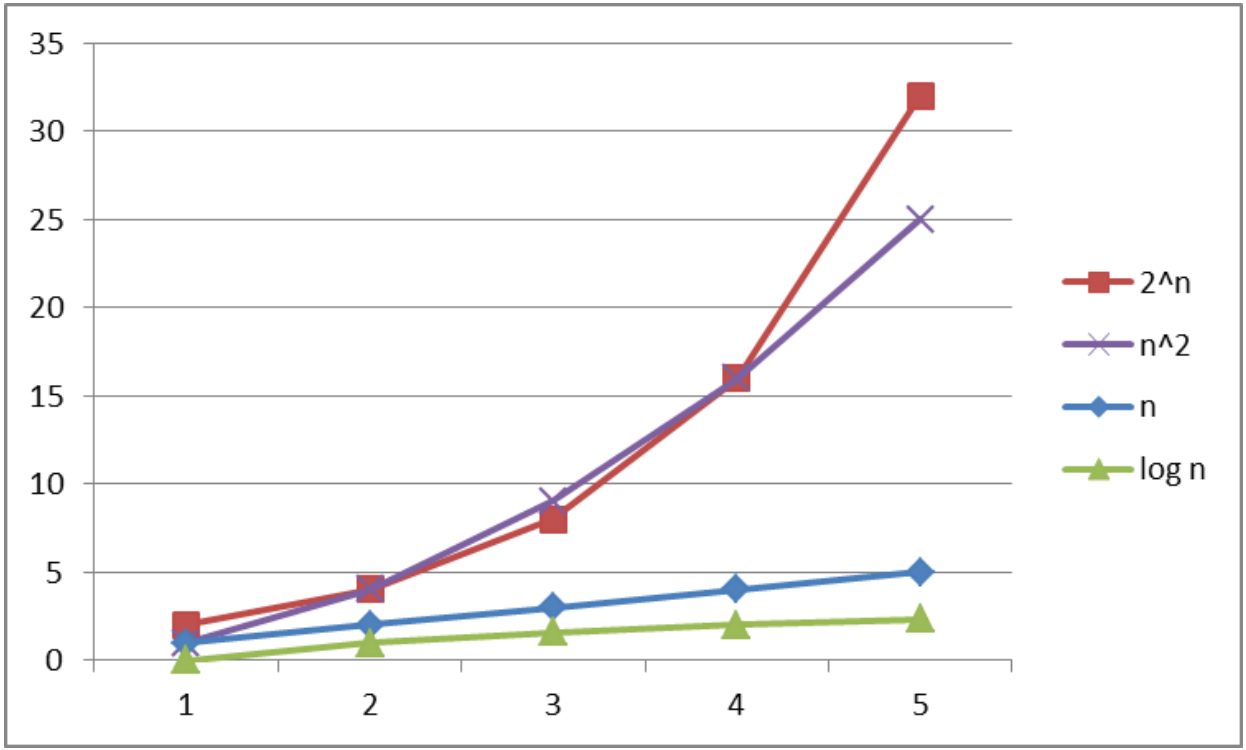
- "Any base *B* log is equivalent to base 2 log within a constant factor"
  - ***And we are about to prove constant factors don't matter!***
  - In particular, $\log_2 x = 3.22 \log_{10} x$

- Why a constant multiplier ?
  - $\log_B x = (\log_A x) / (\log_A B)$

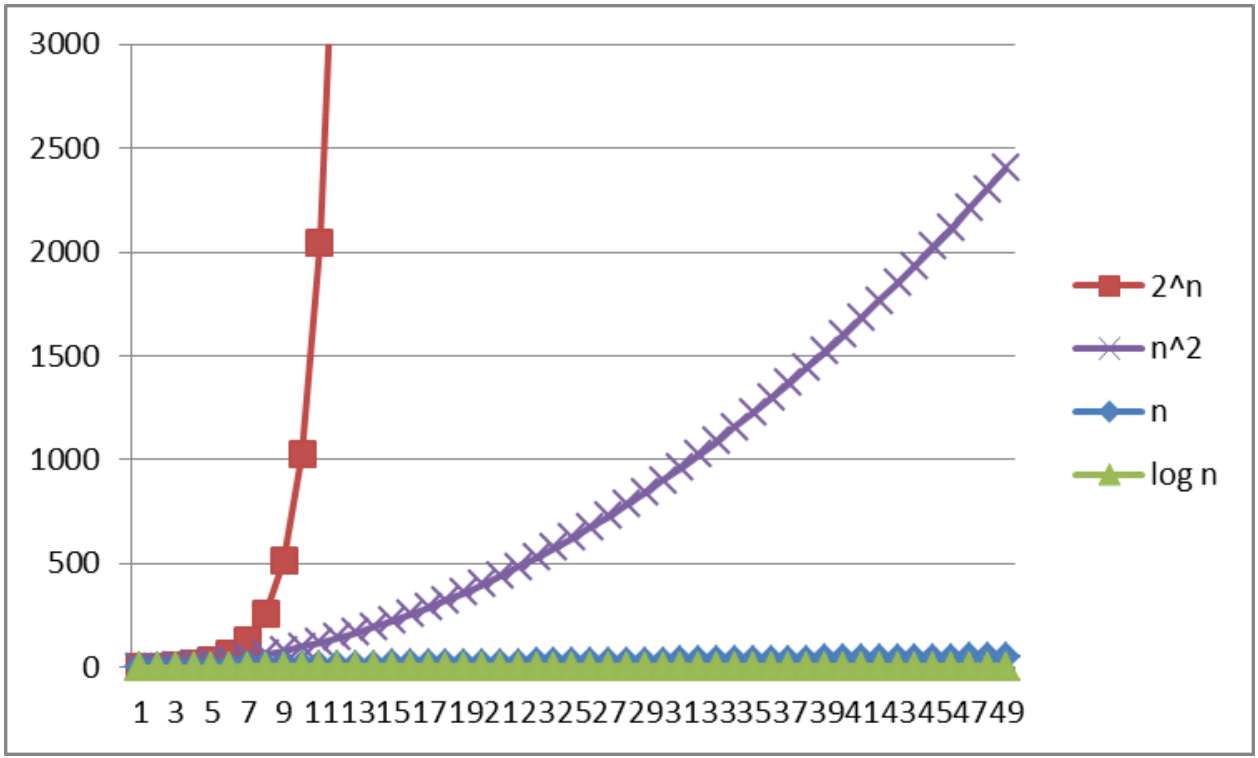
Review: Properties of logarithms

- **log(A*B) = log A + log B**
  - So **log(N^k) = k log N**
- **log(A/B) = log A – log B**
- **X =** $\log_2 2^x$
- **log(log x)** is written $\log^y x$ **log log x**
  - Grows as slowly as 2^2 grows fast
  - Ex: $\log_2 \log_2 4billion \sim \log_2 \log_2 2^{32} = \log_2 32 = 5$
- **(log x)(log x)** is written **$\log^2 x$**
  - It is greater than **log x** for all **x > 2**

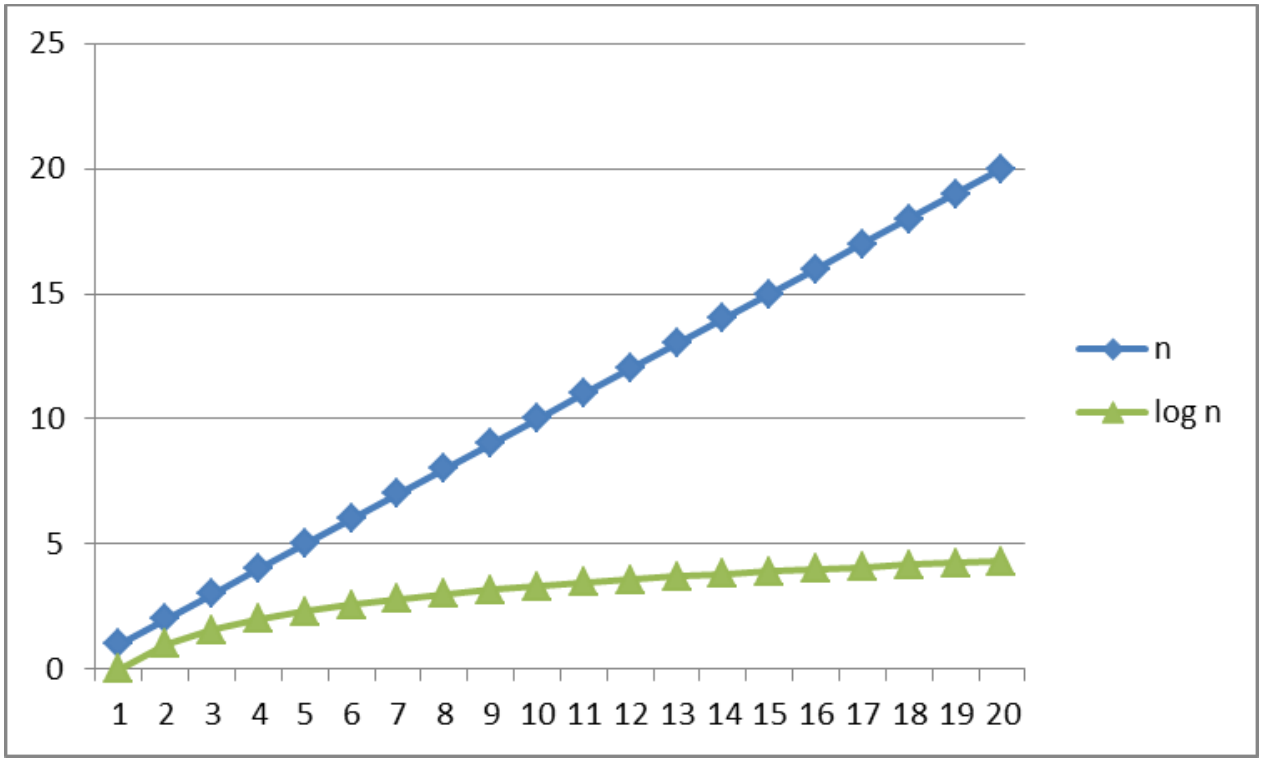
Logarithms and Exponents



Logarithms and Exponents



Logarithms and Exponents



Lecture Outline

- ADT Summary
- The Trie Data Structure
- Algorithms – How do we compare them??
- Analyzing Code
- Asymptotics
- **Big-Oh**

Introduction: Asymptotic Notation

- About to show formal definition, which amounts to our earlier intuitive simplifications:
  - Eliminate lower-order terms
  - Ignore multiplicative constants

- Examples:
  - $4n + 5$
  - $0.5n \log n + 2n + 7$
  - $n^3 + 2^n + 3n$
  - $n \log (10n^2)$

Big-Oh relates functions

- We use O on a function f(n) (for example n^2) to mean *the set of functions with asymptotic behavior* ***less than or equal to*** f(n)

- So $(3n^2+17)$ **is in** $O(n^2)$
  - $3n^2+17$ and n^2 have the same **asymptotic behavior**

- Confusingly, we also say/write:
  - $(3n^2+17)$ **is** $O(n^2)$
  - $(3n^2+17)$ **∈** $O(n^2)$
  - $(3n^2+17)$ **=** $O(n^2)$ ← *least ideal*

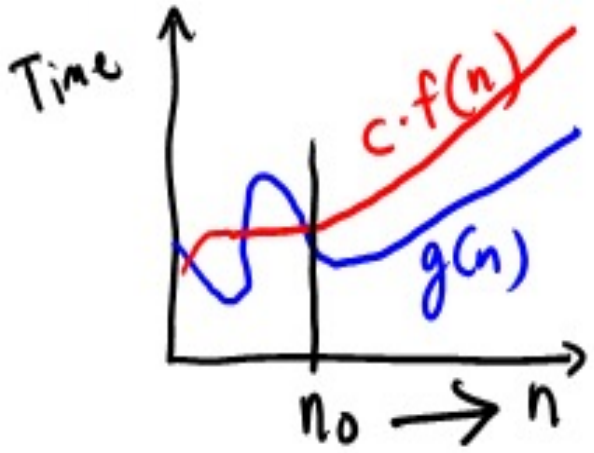
- But we would never say $O(n^2) = (3n^2+17)$

Big-Oh, Formally (1 of 3)

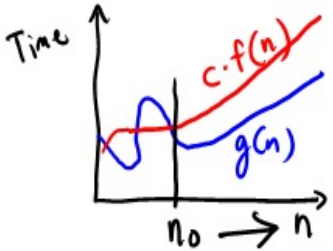
Definition: $g(n)$ is in $O(f(n))$ iff there exist positive constants c and n_0 such that

$g(n) \leq c\, f(n)$ for all $n \geq n_0$

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$



Big-Oh, Formally (2 of 3)



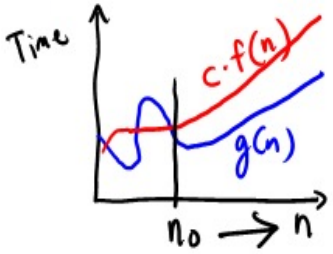
Definition: $g(n)$ is in $O(f(n))$ iff there exist positive constants c and n_0 such that

$g(n) \leq c\, f(n)$ for all $n \geq n_0$

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$

To show $g(n)$ is in $O(f(n))$, pick a c large enough to "cover the constant factors" and n_0 large enough to "cover the lower-order terms"

Big-Oh, Formally (3 of 3)



Definition: $g(n)$ is in $O(f(n))$ iff there exist positive constants c and n_0 such that

$g(n) \leq c\, f(n)$ for all $n \geq n_0$

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$

Example: Let $g(n) = 3n + 4$ and $f(n) = n$

$c = 4$ and $n_0 = 5$ is one possibility

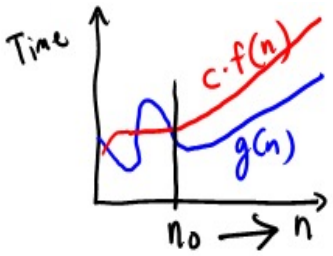
Example: Let $g(n) = 3n + 4$ and $f(n) = n^5$

$c = 3$ and $n_0 = 2$ is one possibility

Example: Let $g(n) = 3n + 4$ and $f(n) = 2^n$

$c = 100000000$ and $n_0 = 1$ is one possibility

Big-Oh, Formally (3 of 3)



Definition: **g(*n*)** is in O(**f(*n*)**) iff there exist positive constants ***c*** and n_0 such that

$g(n) \leq c\, f(n)$ for all $n \geq n_0$

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$

Example: Let **g(*n*)** = 3*n* + 4 and **f(*n*)** = *n*
$3n+4 \leq 4n \quad \forall n \geq 5$
$\therefore 3n+4 \in O(n)$
c = 4 and n_0 = 5 is one possibility

Example: Let **g(*n*)** = 3*n* + 4 and **f(*n*)** = n^5
$3n+4 \leq 3n^5 \quad \forall n \geq 2$
$\therefore 3n+4 \in O(n^5)$
c = 3 and n_0 = 2 is one possibility

Example: Let **g(*n*)** = 3*n* + 4 and **f(*n*)** = 2^n
$3n+4 \leq 100000000 \cdot 2^n \quad \forall n \geq 1$
$\therefore 3n+4 \in O(2^n)$
c = 100000000 and n_0 = 1 is one possibility