

# Tries; Asymptotic Analysis 1

CSE 332 Spring 2020

**Instructor:** Richard Jiang

## Teaching Assistants:

Hamsa Shankar Kristin Li Winston Jodjana

Maggie Jiang Hans Zhang Michael Duan

Jeffery Tian Annie Mao

*Lecture Q&A: [pollev.com/332summer](https://pollev.com/332summer)*

*Lecture clarifications: [tinyurl.com/332-06-24A](https://tinyurl.com/332-06-24A)*

# Announcements

## ❖ Due dates:

- Exercise 1 is due this Friday
- Quiz 1 will be available on Gradescope 12:00am Thursday and due 11:59 Friday
- P1 Ckpt 1 is next Tuesday

## ❖ Before section tomorrow:

- Try gitlab and IntelliJ, so TAs can help debug any issues during section
- Please verify your section's Zoom links are in Canvas

# Lecture Outline

- ❖ **Review: List, Stack, Queue, Dictionary and Set ADTs**
- ❖ The Trie data structure
- ❖ How do we Compare Algorithms?
- ❖ Analyzing Code
- ❖ How to Analyze Large Inputs - Asymptotics
- ❖ Big-Oh Definitions

*[pollev.com/332summer](http://pollev.com/332summer) :: [tinyurl.com/332-06-24A](http://tinyurl.com/332-06-24A)*

# ADTs So Far (1 of 2)

**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*

- ❖ Data structures that implement the List ADT include LinkedList and ArrayList
- ❖ When we restrict List's functionality, we end up with the 2 other ADTs we've seen so far

## ADTs So Far (2 of 2)

**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”)

- ❖ Data structures that implement these ADTs are variants of `LinkedList` and `ArrayList`

# Dictionary ADT

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

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

**Terminology:** a dictionary maps *keys* to *values*; an *item* or *data* refers to the (key, value) pair

❖ Also known as “**Map ADT**”

- `add(k, v)`
- `contains(k)`
- `find(k)`
- `remove(k)`

# Set ADT

**Set ADT.** A collection of keys.

- A set has a size defined as the number of elements in the set
- You can add and remove keys, but the contained values are unique
- Each key is accessible via a “contains” operation

❖ Operations:

- `add(v)`
- `contains(v)`
- `remove(v)`

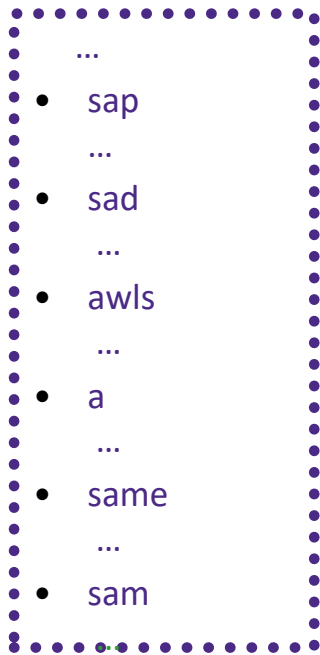
# Lecture Outline

- ❖ Review: List, Stack, Queue, Dictionary and Set ADTs
- ❖ **The Trie data structure**
- ❖ How do we Compare Algorithms?
- ❖ Analyzing Code
- ❖ How to Analyze Large Inputs - Asymptotics
- ❖ Big-Oh Definitions

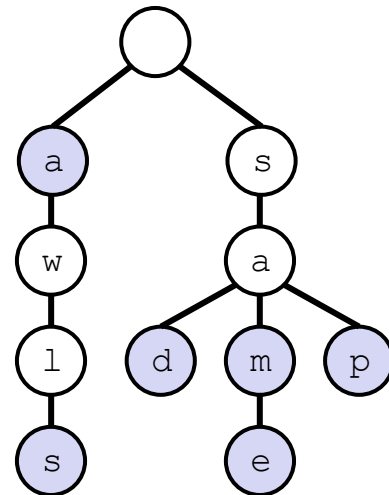
*[pollev.com/332summer](http://pollev.com/332summer) :: [tinyurl.com/332-06-24A](http://tinyurl.com/332-06-24A)*

# The Trie: A Specialized Data Structure

Tries view keys as a **sequence of characters**, keys must be **alphabetic**



Abstract Set ADT



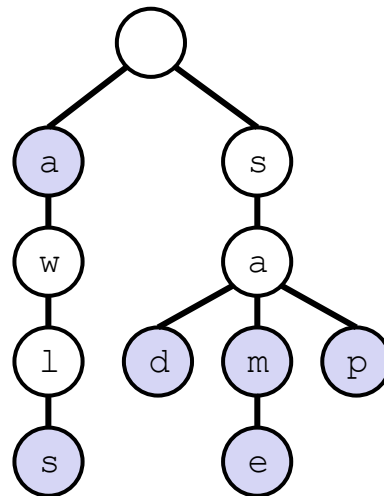
Trie

# An Abstract Trie

Each level of the tree represents an index, and the children represent possible characters at that index.

This trie stores the set of strings:

awls, a, sad,  
same, sap, sam



How to deal with a and awls?

- Mark which nodes *complete* strings (shown in purple)

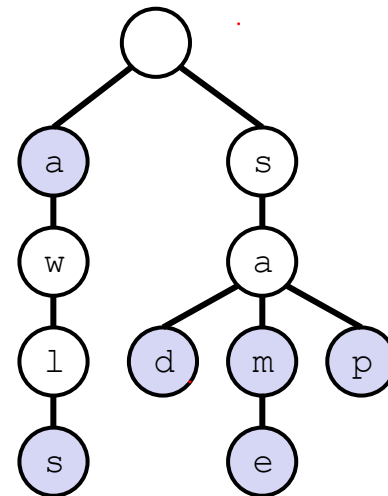
# Searching in Tries

contains("sam"): true, purple. **hit**.

contains("sa"): false, white. **miss**.

contains("a"): true, purple. **hit**.

contains("saq"): false, fell off. **miss**.



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

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

# Keys as “a sequence of tokens” (1 of 2)

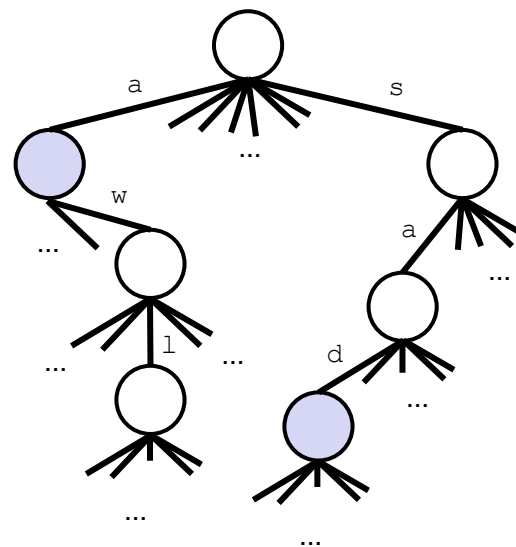
- ❖ Most dictionaries treat their keys as an “atomic blob”: you can’t disassemble the key into smaller components
- ❖ Tries take the opposite view: keys are a **sequence of tokens**
  - `Strings` are made of `Characters`
- ❖ But “tokens” don’t have to come from the Latin alphabet
  - `Character` includes most Unicode codepoints (eg, 蛋糕)
  - `List<E>`
  - `byte[]`

# Keys as “a sequence of tokens” (2 of 2)

- ❖ But “tokens” don’t have to come from the Latin alphabet
  - `Character` includes most Unicode codepoints (eg 蛋糕)
  - `List<E>`
  - `byte[]`
- ❖ Tries are defined by 3 types instead of 2:
  - An “alphabet”: the domain of the characters/tokens
  - A “key”: a sequence of “characters” from the alphabet
  - A “value”: the usual Dictionary value

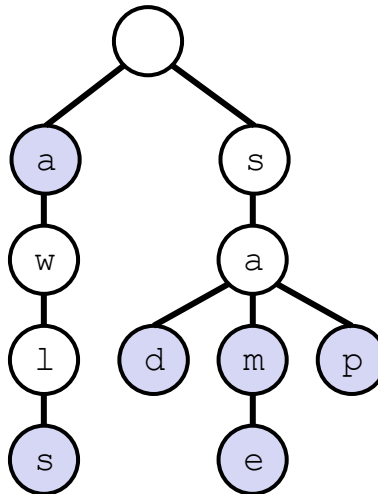
❖ Does the structure of a trie depend on the order in which strings are inserted?

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



# Trie Deletion

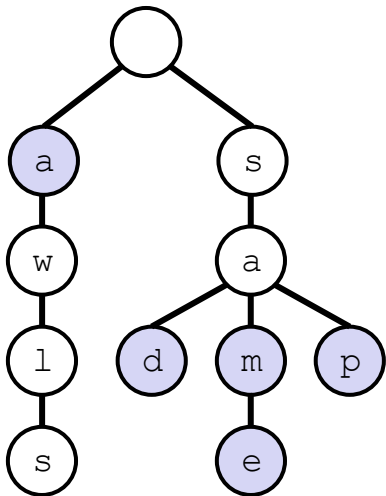
What do we do when a user calls `delete("awls")`?



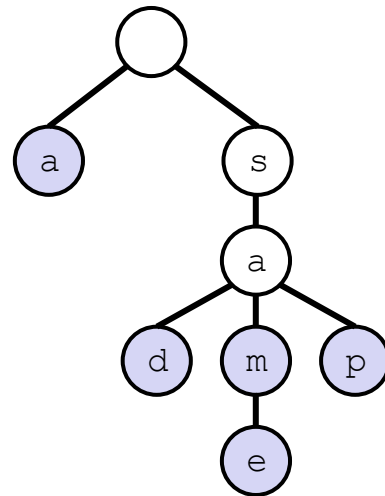
# The Trie: A Specialized Data Structure

What are the tradeoffs of each deletion algorithm?

```
delete("awls")
```



Lazy Deletion



Normal Deletion

# The Trie: A Specialized Data Structure

What are the tradeoffs of each deletion algorithm?

```
delete("awls")
```

Better if..

- You want a faster delete
- Insertion is common
- Reinserting deleted *keys* is expected
- Easier to code (yes, this counts)

Lazy Deletion

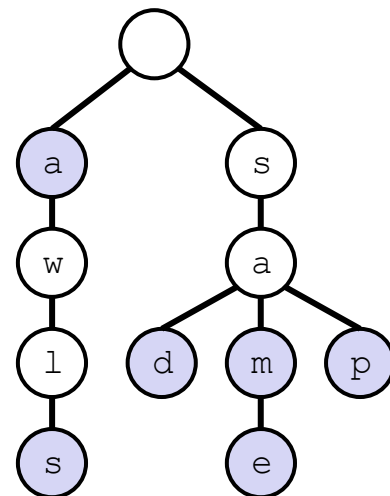
Better if..

- Find/prefix operations are common
- Better space management

Normal Deletion

# String-Specific Operations

- ❖ The main appeal of tries is their efficient prefix matching!
- ❖ **Prefix match**
  - `findPrefix("sa")`
- ❖ **Longest prefix**
  - `longestPrefixOf("sample")`

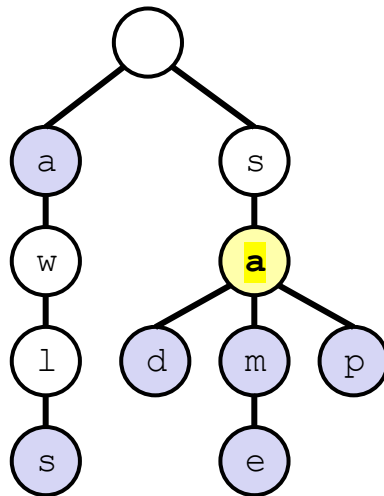


# Prefix Operations with Tries

Describe in English an algorithm for `findPrefix`.

```
findPrefix("sa") :
```

```
["sad", "sam", "same", "sap"]
```

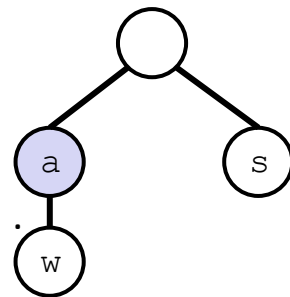


# Collecting Trie Keys

Describe in English an algorithm to collect all the keys in a trie.

```
collect(): ["a", "awls", "sad", "sam", "same", "sap"]
```

1. Create an empty list of results `x`.
2. For character `c` in `root.next.keys()`:  
    Call `colHelp(r + c, x, root.next.get(c))`.
3. Return `x`.



```
colHelp(String s, List<String> x, Node n)
```

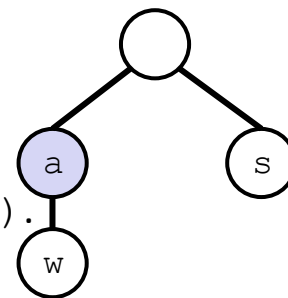
1. ???

# Collecting Trie Keys

Describe in English an algorithm to collect all the keys in a trie.

```
collect(): ["a", "awls", "sad", "sam", "same", "sap"]
```

1. Create an empty list of results `x`.
2. For character `c` in `root.next.keys()`:  
    Call `colHelp(r + c, x, root.next.get(c))`.
3. Return `x`.



```
colHelp(String s, List<String> x, Node n)
```

1. If `n.isKey`, then `x.add(s)`.
2. For character `c` in `n.next.keys()`:  
    Call `colHelp(s + c, x, n.next.get(c))`.

# Lecture Outline

- ❖ Review: List, Stack, Queue, Dictionary and Set ADTs
- ❖ The Trie data structure
- ❖ **How do we Compare Algorithms?**
- ❖ Analyzing Code
- ❖ How to Analyze Large Inputs - Asymptotics
- ❖ Big-Oh Definitions

*[pollev.com/332summer](http://pollev.com/332summer) :: [tinyurl.com/332-06-24A](http://tinyurl.com/332-06-24A)*

# What do we care about?

## ❖ Correctness:

- Does the algorithm do what is intended.

## ❖ Performance:

- Speed **time complexity**
- Memory **space complexity**

## ❖ Why analyze?

- To make good design decisions
- Enable you to look at an algorithm (or code) and identify the bottlenecks, etc.

# Q: How should we compare two algorithms?

# A: How should we compare two algorithms?

- ❖ Uh, why NOT just run the program and time it??
  - Too much *variability*, not reliable or *portable*:
    - Hardware: processor(s), memory, etc.
    - OS, Java version, libraries, drivers
    - Other programs running
    - Implementation dependent
  - Choice of input
    - Testing (inexhaustive) may *miss* worst-case input
    - Timing does not *explain* relative timing among inputs (what happens when  $n$  doubles in size)
  
- ❖ Often want to evaluate an *algorithm*, not an implementation
  - Even *before* creating the implementation (“coding it up”)

# 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)
- ❖ 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!

# 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

- ❖ Review: List, Stack, Queue, Dictionary and Set ADTs
- ❖ The Trie data structure
- ❖ How do we Compare Algorithms?
- ❖ **Analyzing Code**
- ❖ How to Analyze Large Inputs - Asymptotics
- ❖ Big-Oh Definitions

*[pollev.com/332summer](http://pollev.com/332summer) :: [tinyurl.com/332-06-24A](http://tinyurl.com/332-06-24A)*

# Analyzing Code (1 of 2)

- ❖ We abstract away the computer by counting “operations” (time complexity) and “elements” (space complexity)
  - Remember: “Independent of CPU speed, 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 (2 of 2)

- ❖ Basic *operations* take “some amount of” *constant time*
  - Arithmetic
  - Assignment
  - Access one Java field **or array index**
  - 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'll start by focusing on 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
- ❖ Unless otherwise stated, we usually refer to the ***worst case***
  - But there are uses for other types of analyses
  - So for now we'll analyze the ***slower branch***

# Examples

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

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

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

## 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
}
```

# Analyzing Loops, Formally

- ❖ We use summations to quantify the runtime

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

# Example Problem

Find an integer in a *sorted* array

2	3	5	16	37	50	73	75	126
---	---	---	----	----	----	----	----	-----

```
// requires array is sorted
// returns whether k is in array
boolean find(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 array is sorted
// returns whether k is in array
boolean find(int[] arr, int k) {
    for(int i=0; i < arr.length; ++i)
        if(arr[i] == k)
            return true;
    return false;
}
```

Best case:

Worst case:

# Example Solution: Linear Search Runtimes

Find an integer in a *sorted* array

2	3	5	16	37	50	73	75	126
---	---	---	----	----	----	----	----	-----

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k) {
    for(int i=0; i < arr.length; ++i)
        if(arr[i] == k)
            return true;
    return false;
}
```

Best case: 6 “ish” steps =  $O(1)$

Worst case: 5 “ish” \* (arr.length) + 1  
=  $O(\text{arr.length})$

# Lecture Outline

- ❖ Review: List, Stack, Queue, Dictionary and Set ADTs
- ❖ The Trie data structure
- ❖ How do we Compare Algorithms?
- ❖ Analyzing Code
- ❖ **How to Analyze Large Inputs - Asymptotics**
- ❖ Big-Oh Definitions

*[pollev.com/332summer](http://pollev.com/332summer) :: [tinyurl.com/332-06-24A](http://tinyurl.com/332-06-24A)*

# Ignoring Constant Factors (1 of 2)

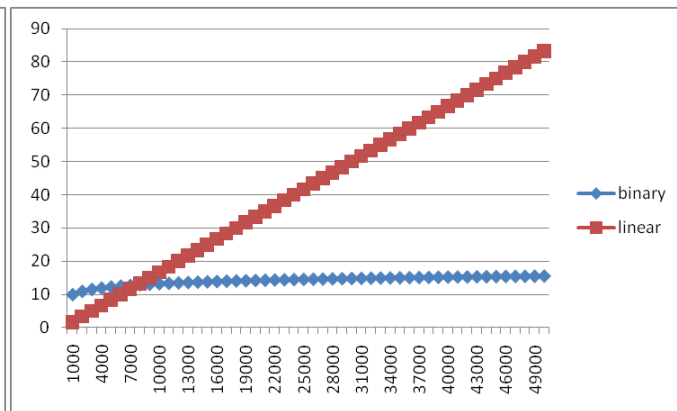
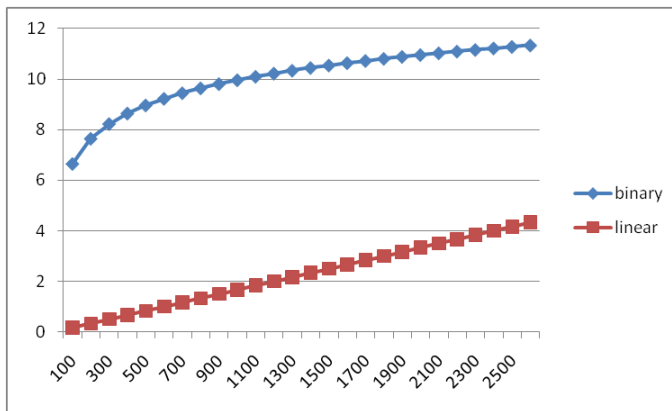
- ❖ Binary search is  $O(\lg n)$  and linear is  $O(n)$ 
  - But which is actually *faster*?
  - Depending on *constant factors* and *size of  $n$* , in a particular situation, *linear search could be faster!*
- ❖ Depends on constant factors:
  - How *many* assignments, additions, etc. for each  $n$
- ❖ Depends on size of  $n$ :
  - Remember: “Descriptive of large inputs; most algorithms are ‘good enough’ for small inputs”
  - Each data structure’s and algorithm’s behavior can vary for every finite  $N$
  - So we pick  $N \rightarrow \infty$  as our definition of “large”

## Ignoring Constant Factors (2 of 2)

- ❖ How formalize the intuitive idea of how an algorithm behaves as  $N \rightarrow \infty$ ?
  - 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~~

- $\frac{5}{2}N^2$

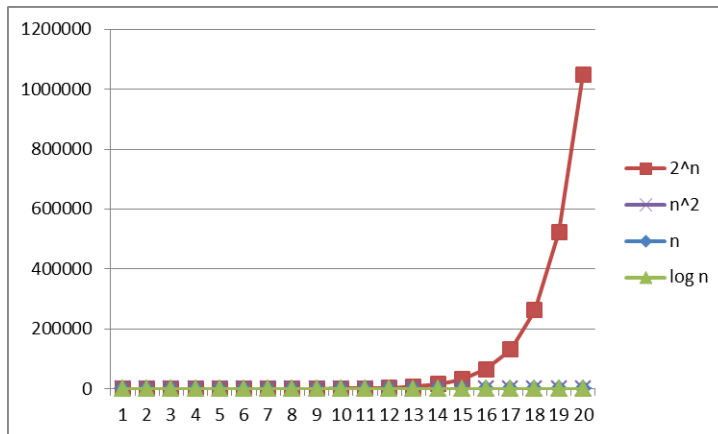
- ❖ (2) Ignore multiplicative constants

- ~~$\frac{5}{2}$~~   $N^2$

- $N^2$

# Logarithms and Exponents

- ❖ Definition:  $\log_2 x = y$  if  $x = 2^y$ 
  - Note: since so much is binary in CS,  $\log$  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

$$\diamond \log(A \cdot B) = \log A + \log B$$

$$\blacksquare \text{ So } \log(N^k) = k \log N$$

$$\diamond \log(A/B) = \log A - \log B$$

$$\diamond x = \log_2 2^x$$

$$\diamond \log(\log x) \text{ is written } \log^y \log x$$

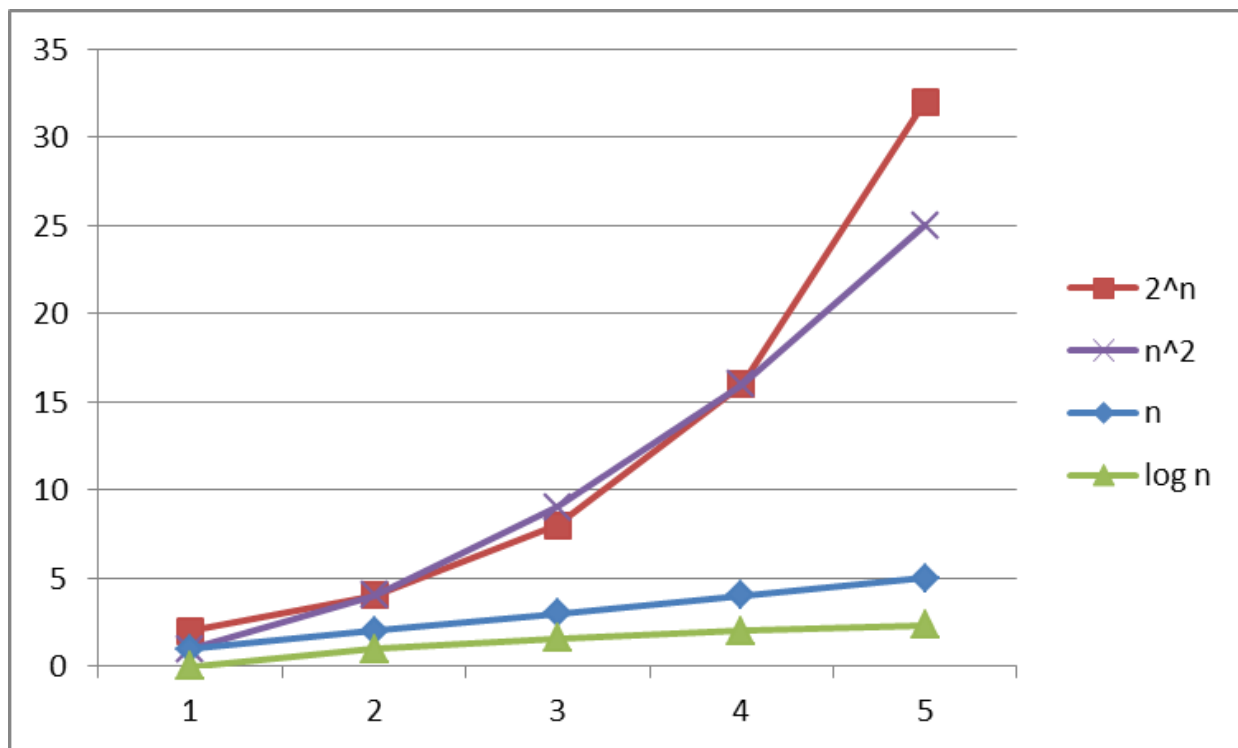
$$\blacksquare \text{ Grows as slowly as } 2^2 \text{ grows fast}$$

$$\blacksquare \text{ Ex: } \log_2 \log_2 4\textit{billion} \sim \log_2 \log_2 2^{32} = \log_2 32 = 5$$

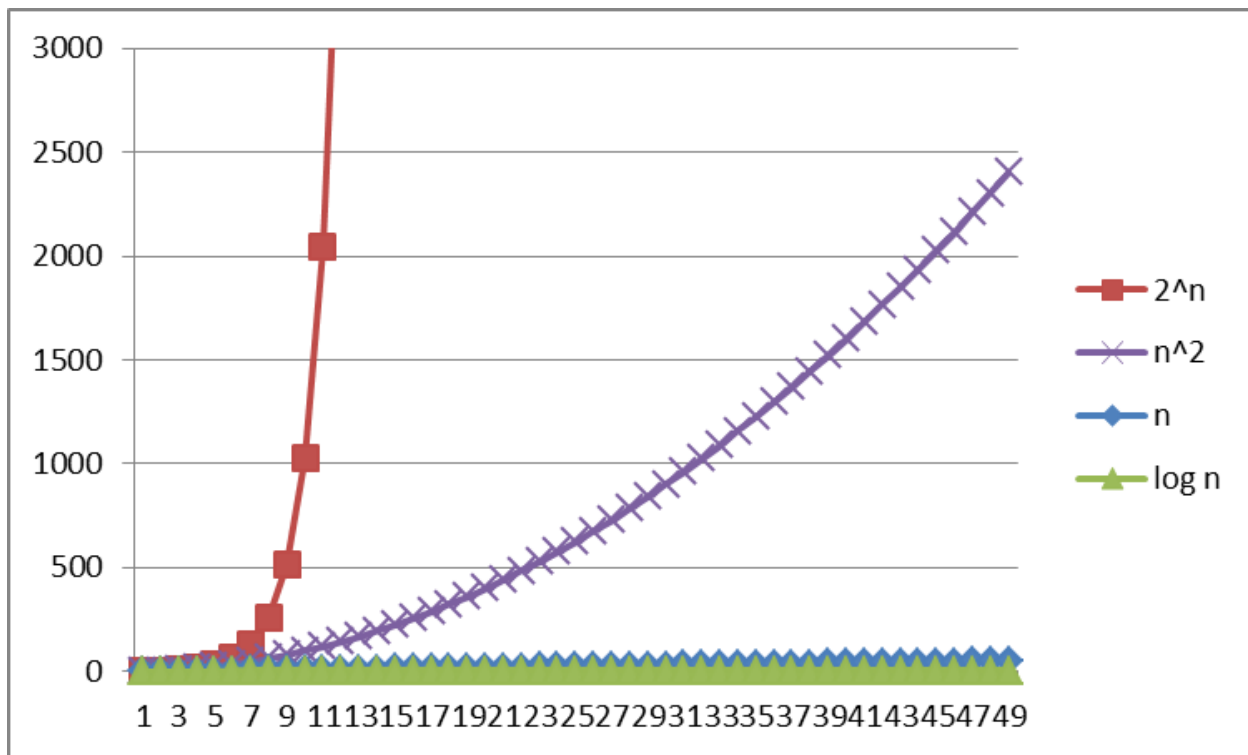
$$\diamond (\log x)(\log x) \text{ is written } \log^2 x$$

$$\blacksquare \text{ It is greater than } \log x \text{ for all } x > 2$$

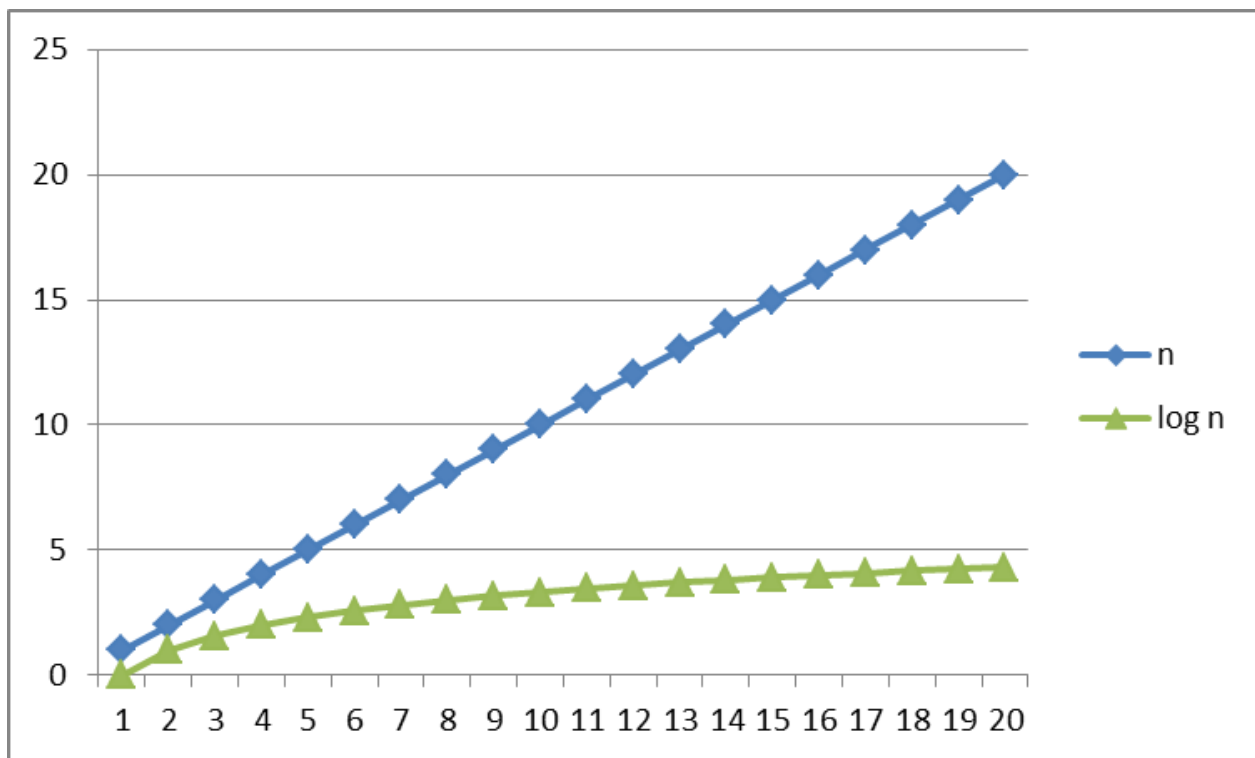
# Logarithms and Exponents



# Logarithms and Exponents



# Logarithms and Exponents



# Lecture Outline

- ❖ Review: List, Stack, Queue, Dictionary and Set ADTs
- ❖ The Trie data structure
- ❖ How do we Compare Algorithms?
- ❖ Analyzing Code
- ❖ How to Analyze Large Inputs - Asymptotics
- ❖ **Big-Oh Definitions**

*[pollev.com/332summer](http://pollev.com/332summer) :: [tinyurl.com/332-06-24A](http://tinyurl.com/332-06-24A)*

# 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

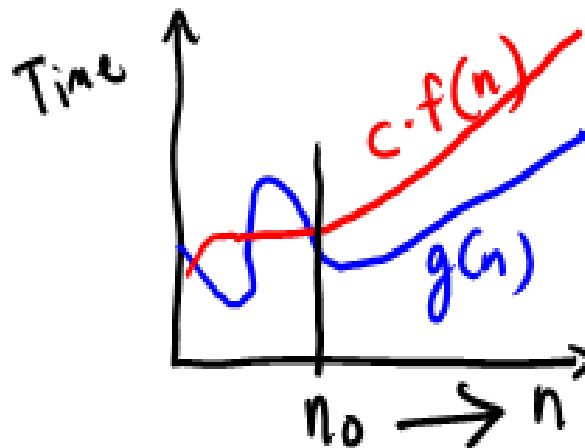
- ❖ We use  $O$  on a function  $f(n)$  (for example  $n^2$ ) to mean *the set of functions with asymptotic behavior less than or equal to  $f(n)$*
- ❖ So  $(3n^2+17)$  **is in**  $O(n^2)$ 
  - $3n^2+17$  and  $n^2$  have the same **asymptotic behavior**
- ❖ Confusingly, we also say/write:
  - $(3n^2+17)$  **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

$$g(n) \leq c f(n) \quad \text{for all } n \geq n_0$$

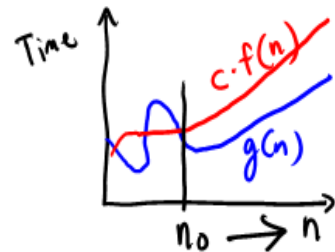
Note:  $n_0 \geq 1$  (and a natural number) and  $c > 0$



## 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

$$g(n) \leq c f(n) \quad \text{for all } n \geq n_0$$



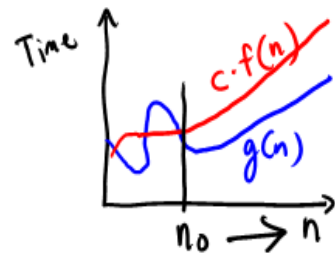
**Note:**  $n_0 \geq 1$  (and a natural number) and  $c > 0$

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

$$g(n) \leq c f(n) \quad \text{for all } n \geq n_0$$



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

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

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

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