CSC 172– Data Structures and Algorithms

Lecture 4

Spring 2018

TuTh 3:25 pm – 4:40 pm

Agenda

- Administrative aspects
- Java Generics (Wildcard)
- Chapter 3 (Mathematical Background)

Chapter 1

ADMINISTRATIVE ASPECTS

Concerns

- That the materials that we cover in class won't be enough to complete the labs or projects

- Answer: We will not cover Java Graphics. Except that we will cover everything you need to complete the labs and projects.

Concerns

- The lights on the projector make it difficult to see the slides in the recording of the lecture. Would it be possible to dim the lights that are directed towards the projector so that it would be easier to see the slides in future lecture videos?
- Answer: I will do this and probably everything is fine now. If you can't see the projection, just let me know right then and there.
  - Also applicable for audio, empty screen, typos, and other issues.

Concerns

- The name of the workshop leaders hasn't been uploaded on the excel sheet
- Answer: You can find it on the same google sheet I shared.

Concerns

- The mobile css for your website is not linked so your header and all the links don't show up on my iPad
- Answer: I have fixed the issue. Still if it does not work, let me know.

Concerns

- The class resources seem very redundant. From memorizing an arbitrary class id, to requiring the use of the course web page in lieu of blackboard, even the use of piazza.

- Answer:
- Don’t memorize the class id, just keep a note (probably as a contact on phone! or on the notebook you are using)

- Webpage vs Blackboard:
  - Pros:
    - You do not need to log in every time
    - You can access all the resources from there. Lectures are dated
  - Cons: ?
- Piazza:
  - Pros:
    - Better interface than Blackboard. Searching is easier.
  - Cons: ?

Concerns

- Do I need to worry if you keep note of the students who are answering questions?
- Answer:
  - No. First of all, you are not losing any points for not participating.
  - I keep a note to know the students who participates often by their name. Might be useful when a student ask for a recommendation letter or when we offer a TA position to the student.
- A few of them will get extra credit for their performance (not more than 5 students)
  - Most of these students often get As even without these points.
- If you answer a question and I do not mark your ClassID, feel free to send an email stating the same. Remind me what the question was and what your answered (helpful for me to make a face-to-name connection).

Readings

- Java Generics (For the lab and next quiz)

- Chapter 3 and Chapter 4
  - From http://lti.cs.vt.edu/LTI_ruby/Books/CS172/html/

JAVA GENERICS

- Resources:
  - https://docs.oracle.com/javase/tutorial/java/generics/index.html
  - https://docs.oracle.com/javase/tutorial/extra/generics/index.html

What did we cover last time?

- How Java collections framework handles generic
  - We talked about: java.util.List
- How you can make a custom class generic
- How you can make methods generic

Method (Not Generic)

```
public static int countGreaterThan(int[] anArray, int elem)
{
    int total = 0;
    for (int val:anArray) {
        if (val > elem) {
            total++;
        }
    }
    return total;
}
```

Coding Time

- Now covert the method into a generic method which can handle any array.

- Your answer does not necessarily need to be correct (there are a few unknown we did not talk about)
- A good way to learn!

Method (Generic)

```
public static <T extends Comparable<T>> int
              countGreaterThan(T[] anArray, T elem)
{
    int total = 0;
    for (T val:anArray) {
        if (val.compareTo(elem) > 0) {
            total++;
        }
    }
    return total;
}
```

WILDCARDS (?)

Method that Prints All Elements in a Collection

```
void printCollection(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

only takes Collection<Object>
which is **not** a supertype of all kinds of collections!

Generic Inheritance

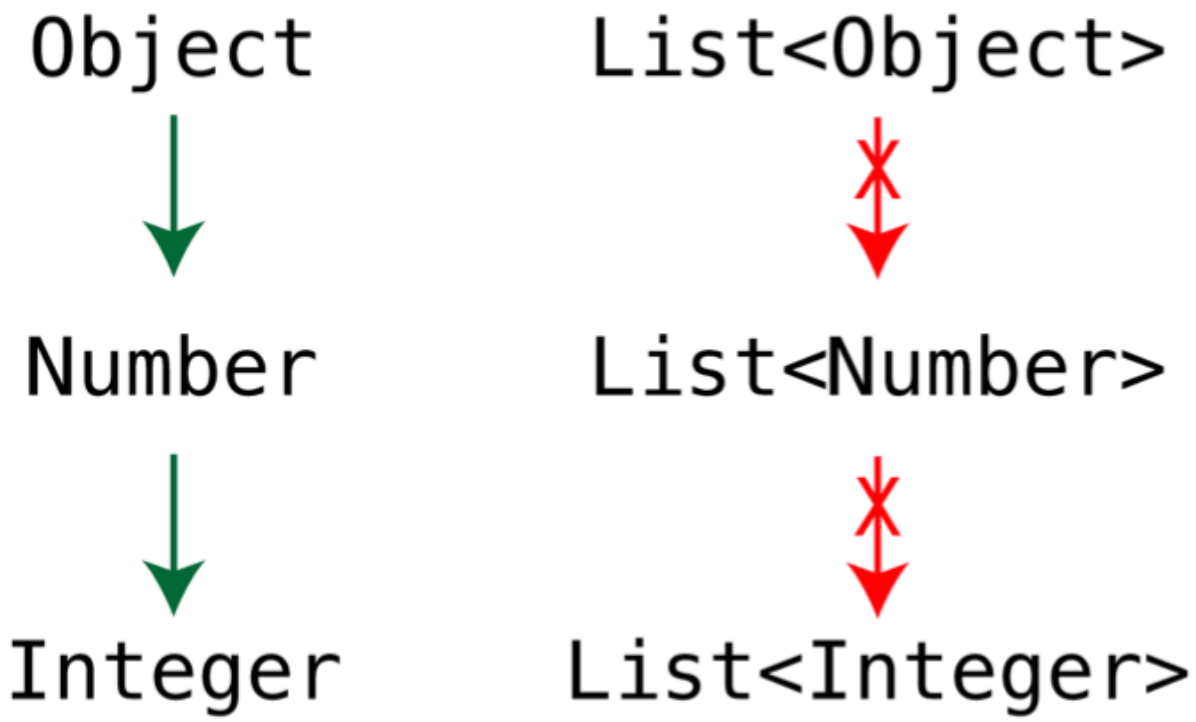
OK

```
Integer b = new Integer(5);
Number a = b;
```

Error

```
ArrayList<Integer> intList = new ArrayList<>();
ArrayList<Number> numList = intList;
```

Why?



ArrayList<Integer> is not subtype of ArrayList<Number>, even though Integer is a subclass of Number.
If we want to use generics on a more general level, we need to use *wildcards*, which are denoted by a ? symbol.

Bounded Wildcards

```
public abstract class Shape {
    public abstract void draw(Canvas c);
}

public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) {
        ...
    }
}

public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) {
        ...
    }
}
```

Bounded Wildcards

These classes can be drawn on a canvas:

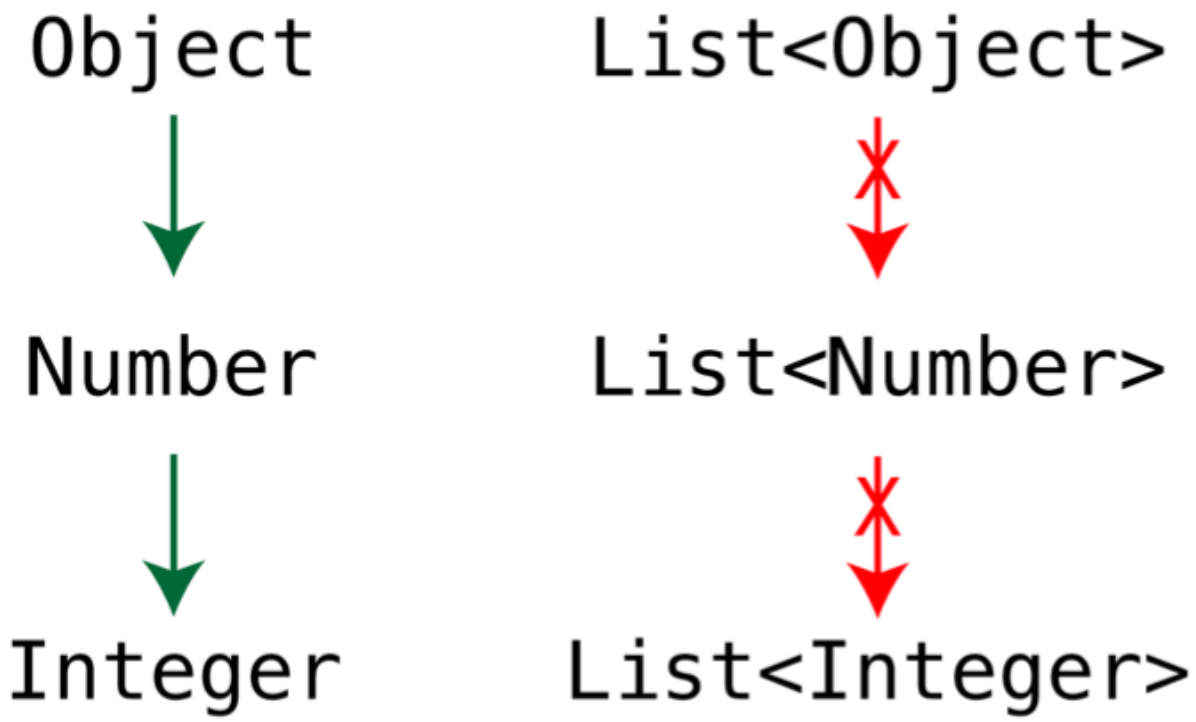
```
public class Canvas {
    public void draw(Shape s) {
        s.draw(this);
    }
}
```

Draw them all

```
public void drawAll(List<Shape> shapes) {
    for (Shape s: shapes) {
        s.draw(this);
    }
}
```

drawAll() can only be called on lists of exactly Shape
it cannot, for instance, be called on a List<Circle>

Why?



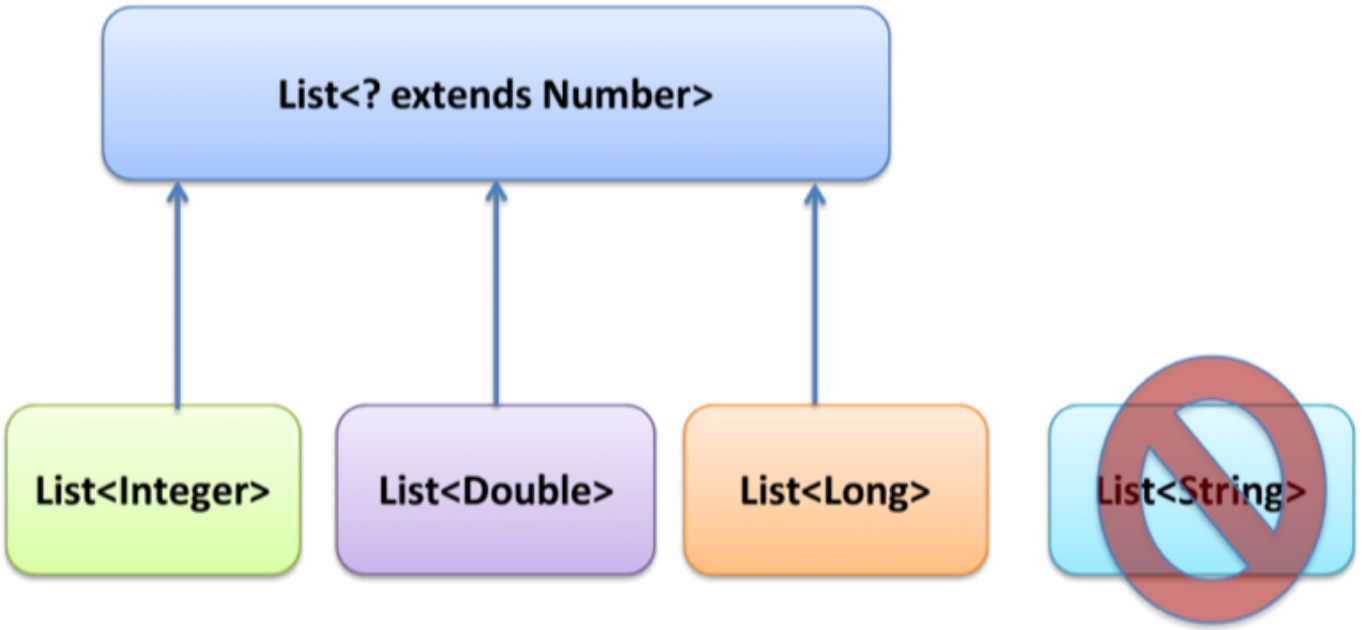
ArrayList<Integer> is not subtype of ArrayList<Number>, even though Integer is a subclass of Number.
If we want to use generics on a more general level, we need to use *wildcards*, which are denoted by a ? symbol.

Solution

```
public void drawAll(List<? extends Shape> shapes) {
    ...
}
```

List<? **extends** Shape> is an example of a *bounded (upper bound) wildcard*.
we know that this unknown type is in fact a subtype of Shape.

Similar Idea



Still!

```
public void addRectangle(List<? extends Shape> shapes) {
    // Compile-time error!
    shapes.add(0, new Rectangle());
}
```

Solution: super

```
public void addRectangle(List<? super Rectangle> shapes)
{
    shapes.add(0, new Rectangle());
}
```

super (lower bound)

```
public void addRectangle(List<? super Rectangle> shapes)
{
    shapes.add(0, new Rectangle());
}
```

Another Solution: Generic Method

```
public void addRectangle(List<? super Rectangle> shapes)
{
    shapes.add(0, new Rectangle());
}
```

VS.

```
public <T extends Shape> void addRectangle(List<T> shapes)
{
    shapes.add(0, new T());
}
```

Which one to choose?

- Generic Method vs Wildcard

```
interface Collection<E> {
    public boolean containsAll(Collection<?> c);
    public boolean addAll(Collection<? extends E> c);
}
```

```
interface Collection<E> {
    public <T> boolean containsAll(Collection<T> c);
    public <T extends E> boolean addAll(Collection<T> c);
    // Hey, type variables can have bounds too!
}
```

Generic Method vs Wildcard

```
class Collections {
    public static <T> void copy(List<T> dest, List<? extends T> src) {
    ...
}
```

VS

```
class Collections {
    public static <T, S extends T> void copy(List<T> dest, List<S> src) {
    ...
}
```

Using wildcards is clearer and more concise than declaring explicit type parameters , and should therefore be preferred whenever possible.

(NOT REQUIRED FOR QUIZ OR EXAM)

A great explanation

- https://stackoverflow.com/a/4343547
- Remember *PECS*: **"Producer Extends, Consumer Super"**.
-

extends (stack overflow)

The wildcard declaration of `List<? extends Number> foo3` means that any of these are legal assignments:

```
List<? extends Number> foo3 = new ArrayList<Number>();  // Number "extends" Number (
List<? extends Number> foo3 = new ArrayList<Integer>(); // Integer extends Number
List<? extends Number> foo3 = new ArrayList<Double>();  // Double extends Number
```

1. **Reading** - Given the above possible assignments, what type of object are you guarenteed to read from `List foo3`:
   - You can read a `Number` because any of the lists that could be assigned to `foo3` contain a `Number` or a subclass of `Number`.
   - You can't read an `Integer` because `foo3` could be pointing at a `List<Double>`.
   - You can't read a `Double` because `foo3` could be pointing at a `List<Integer>`.
2. **Writing** - Given the above possible assignments, what type of object could you add to `List foo3` that would be legal for **all** the above possible `ArrayList` assignments:
   - You can't add an `Integer` because `foo3` could be pointing at a `List<Double>`.
   - You can't add a `Double` because `foo3` could be pointing at a `List<Integer>`.
   - You can't add a `Number` because `foo3` could be pointing at a `List<Integer>`.

You can't add any object to `List<? extends T>` because you can't guarantee what kind of `List` it is really pointing to, so you can't guarantee that the object is allowed in that `List`. The only "guarantee" is that you can only read from it and you'll get a `T` or subclass of `T`.

super (stack overflow)

The wildcard declaration of `List<? super Integer> foo3` means that any of these are legal assignments:

```
List<? super Integer> foo3 = new ArrayList<Integer>();  // Integer is a "superclass"
List<? super Integer> foo3 = new ArrayList<Number>();   // Number is a superclass of
List<? super Integer> foo3 = new ArrayList<Object>();   // Object is a superclass of
```

1. **Reading** - Given the above possible assignments, what type of object are you guaranteed to receive when you read from `List foo3`:
   - You aren't guaranteed an `Integer` because `foo3` could be pointing at a `List<Number>` or `List<Object>`.
   - You aren't guaranteed an `Number` because `foo3` could be pointing at a `List<Object>`.
   - The **only** guarantee is that you will get an instance of an `Object` or subclass of `Object` (but you don't know what subclass).
2. **Writing** - Given the above possible assignments, what type of object could you add to `List foo3` that would be legal for **all** the above possible `ArrayList` assignments:
   - You can add an `Integer` because an `Integer` is allowed in any of above lists.
   - You can add an instance of a subclass of `Integer` because an instance of a subclass of `Integer` is allowed in any of the above lists.
   - You can't add a `Double` because `foo3` could be pointing at a `ArrayList<Integer>`.
   - You can't add a `Number` because `foo3` could be pointing at a `ArrayList<Integer>`.
   - You can't add a `Object` because `foo3` could be pointing at a `ArrayList<Integer>`.

Functional Programming with Java 8

```
import java.util.function.*;

public class FuncObject {
  public static void main(String[] args) {

Function<Integer,Integer> add1 = x -> x + 1;
Function<String,String> concat = x -> "Hello, "  +  x;


Integer six= add1.apply(5);
String answer = concat.apply("Tom");

System.out.println(six);
System.out.println(answer);


  }
}
```

- Link: https://dzone.com/articles/functional-programming-java-8

MATHEMETICAL BACKGROUND

Mathematics and engineering

For engineers, mathematics is a tool:

– It helps us to understand why certain approaches works!

Justification

However, as engineers, you will not be paid to say:

Method A is *better* than Method B

or

Algorithm A is *faster* than Algorithm B

Such comparisons are said to be *qualitative*:

qualitative, *a.* Relating to, connected or concerned with, quality or qualities.
Now usually in implied or expressed opposition to quantitative.

OED

Justification

We will look at a *quantitative* means of describing data structures and algorithms:

quantitative, *a.* Relating to, concerned with, quantity or its measurement; ascertaining or expressing quantity. **OED**

This will be based on mathematics, and therefore we will look at a number of properties which will be used again and again throughout this course

Floor and ceiling functions

The *floor* function maps any real number x onto the greatest integer less than or equal to x:

$$\lfloor 3.2 \rfloor = \lfloor 3 \rfloor = 3$$
$$\lfloor -5.2 \rfloor = \lfloor -6 \rfloor = -6$$

– Consider it *rounding towards negative infinity*

The *ceiling* function maps x onto the least integer greater than or equal to x:

$$\lceil 3.2 \rceil = \lceil 4 \rceil = 4$$
$$\lceil -5.2 \rceil = \lceil -5 \rceil = -5$$

– Consider it *rounding towards positive infinity*

Logarithms

We will begin with a review of logarithms:

If $n = e^m$, we define

$$m = \ln(\, n \,)$$

It is always true that $e^{\ln(n)} = n$;

however, $\ln(e^n) = n$ requires that n is real

Logarithms

Exponentials grow faster than any non-constant polynomial

$$\lim_{n \to \infty} \frac{e^n}{n^d} = \infty$$

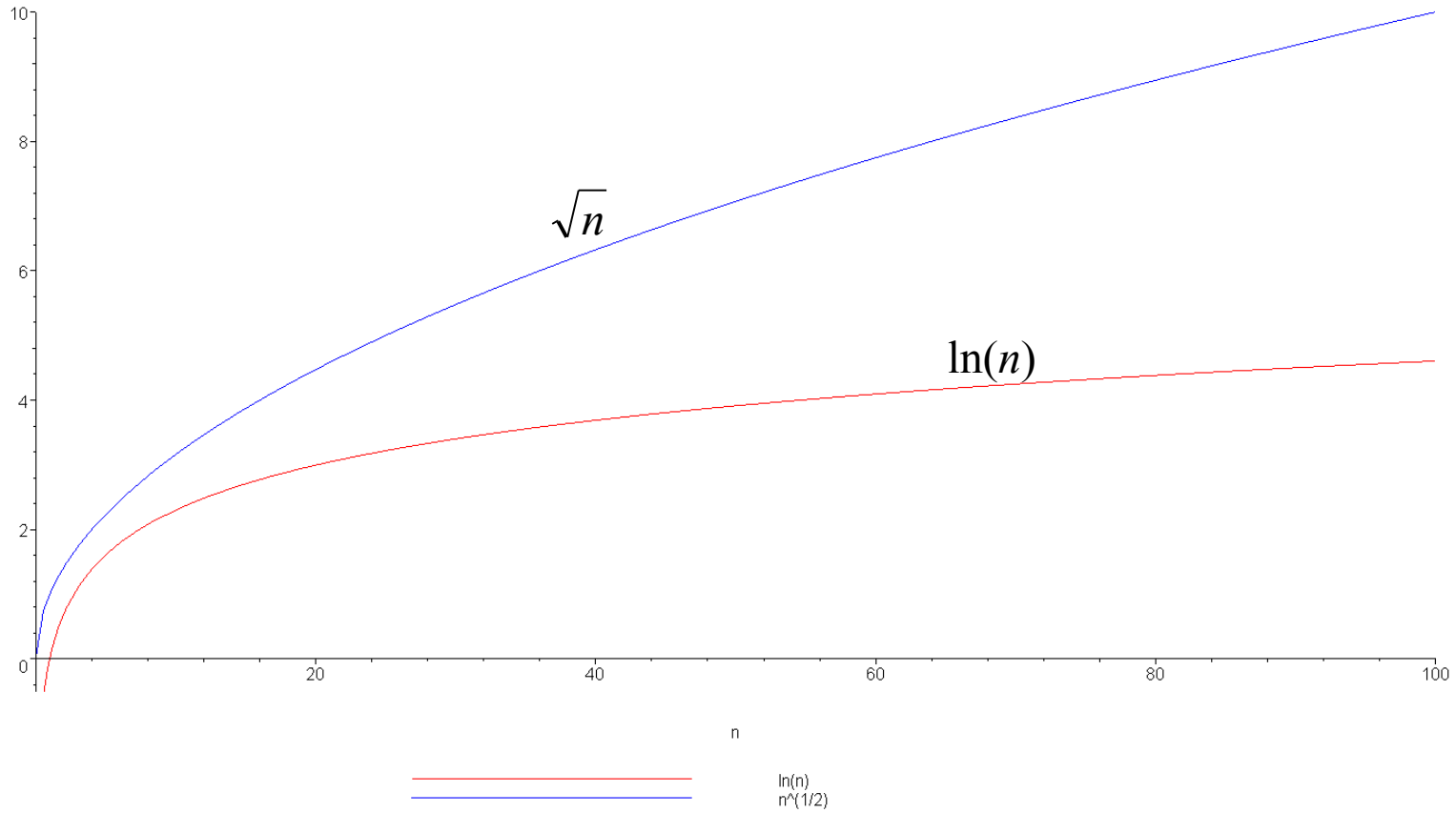
for any $d > 0$

$$\lim_{n \to \infty} \frac{\ln(n)}{n^d} = 0$$

Thus, their inverses—logarithms—grow slower than any polynomial

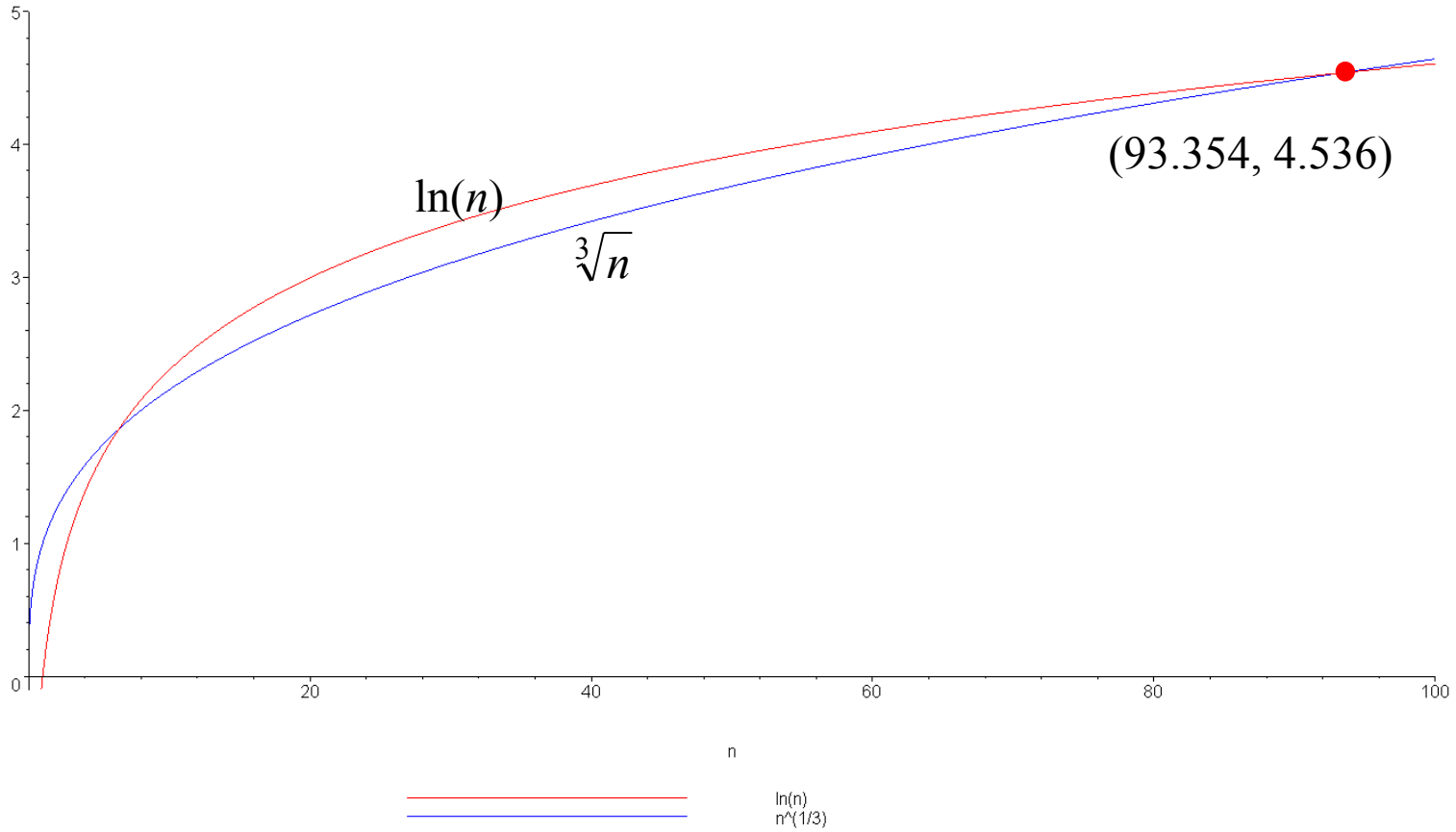
Logarithms

Example: $f(n) = n^{1/2} = \sqrt{n}$ is strictly greater than $\ln(n)$



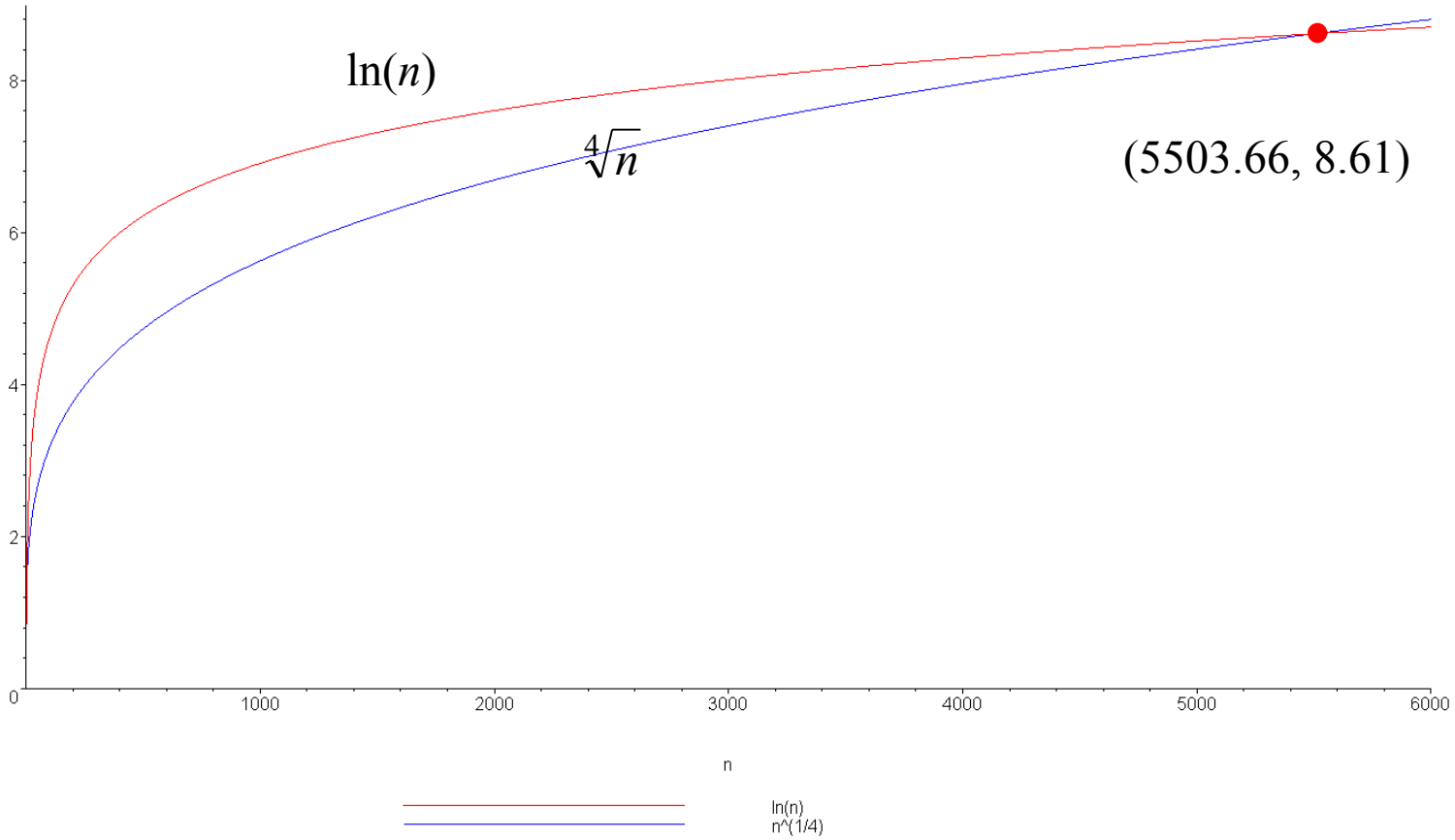
Logarithms

$f(n) = n^{1/3} = \sqrt[3]{n}$ grows slower but only up to $n = 93$



Logarithms

You can view this with any polynomial

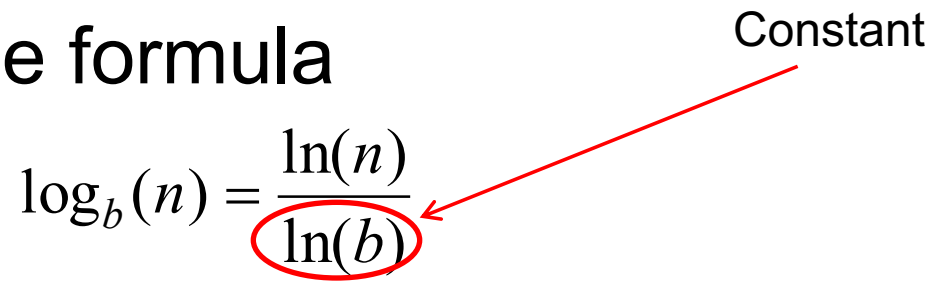


Logarithms

We have compared logarithms and polynomials

– How about $\log_2(n)$ versus $\ln(n)$ versus $\log_{10}(n)$

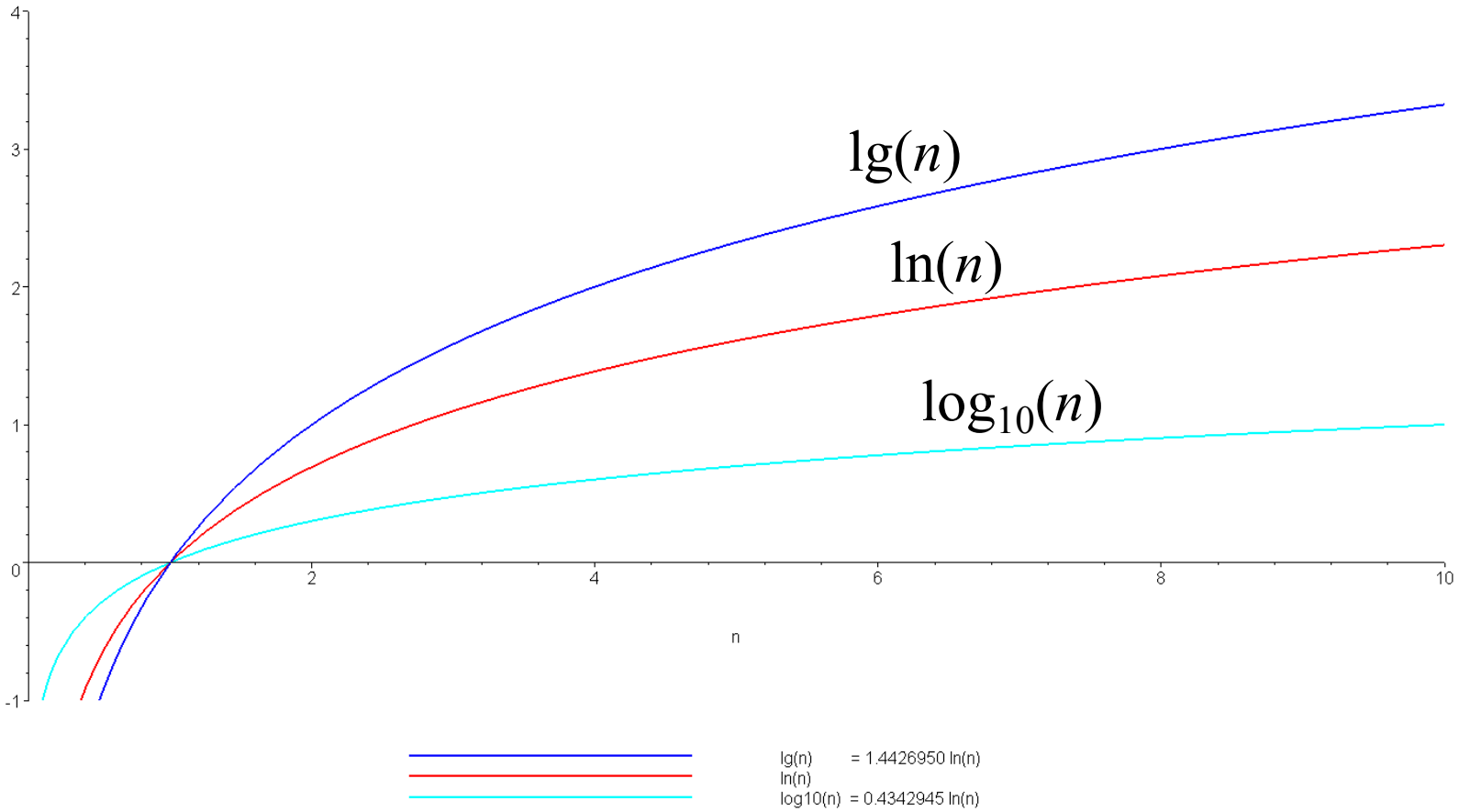
You have seen the formula



All logarithms are scalar multiples of each others

Logarithms

A plot of $\log_2(n) = \lg(n)$, $\ln(n)$, and $\log_{10}(n)$



Logarithms

Note: the base-2 logarithm $\log_2(n)$ is written as $\lg(n)$

It is an industry standard to implement the natural logarithm $\ln(n)$ as

```
double log( double );
```

The *common* logarithm $\log_{10}(n)$ is implemented as

```
double log10( double );
```

Logarithms

A more interesting observation we will repeatedly use:

$$n^{\log_b(m)} = m^{\log_b(n)},$$

$$n = b^{\log_b n}$$

a consequence of :

$$\begin{aligned} n^{\log_b(m)} &= \left(b^{\log_b(n)}\right)^{\log_b(m)} \\ &= b^{\log_b(n)\,\log_b(m)} \\ &= \left(b^{\log_b(m)}\right)^{\log_b(n)} \\ &= m^{\log_b(n)} \end{aligned}$$

Arithmetic series

Next we will look various series

Each term in an arithmetic series is increased by a constant value (usually 1) :

$$0+1+2+3+\quad +n=\sum_{k=0}^{n}k=\frac{n(n+1)}{2}$$

Arithmetic series

Proof 1: write out the series twice and add each column

$$\begin{array}{rcccccccccccc} & 1 & + & 2 & + & 3 & + \cdots + & n-2 & + & n-1 & + & n \\ + & n & + & n-1 & + & n-2 & + \cdots + & 3 & + & 2 & + & 1 \\ \hline & (n+1) & + & (n+1) & + & (n+1) & + \cdots + & (n+1) & + & (n+1) & + & (n+1) \end{array}$$

$$= n\,(n+1)$$

Since we added the series twice, we must divide the result by 2

Arithmetic series

Proof 2 (by induction):

Base Case:

The statement is true for $n = 0$:

$$\sum_{i=0}^{0} k = 0 = \frac{0 \cdot 1}{2} = \frac{0(0+1)}{2}$$

Induction Hypothesis:

Assume that the statement is true for an arbitrary n:

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2}$$

Arithmetic series

Using the assumption that

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2}$$

for n, we must show that

$$\sum_{k=0}^{n+1} k = \frac{(n+1)(n+2)}{2}$$

Arithmetic series

Then, for $n + 1$, we have:

$$\sum_{k=0}^{n+1} k = (n+1) + \sum_{i=0}^{n} k$$

By assumption, the second sum is known:

$$= (n+1) + \frac{n(n+1)}{2}$$

$$= \frac{(n+1)2 + (n+1)n}{2}$$

$$= \frac{(n+1)(n+2)}{2}$$

Arithmetic series

The statement is true for $n = 0$ and

the truth of the statement for n implies

the truth of the statement for $n + 1$.

Therefore, by the process of mathematical induction, the statement is true for all values of $n \geq 0$.

Acknowledgement

- ORACLE Java Documentation
- https://code.snipcademy.com/tutorials/java/generics/upper-lower-unbounded-wildcards
- Douglas Wilhelm Harder. Thanks for making an excellent set of slides for ECE 250 *Algorithms and Data Structures* course