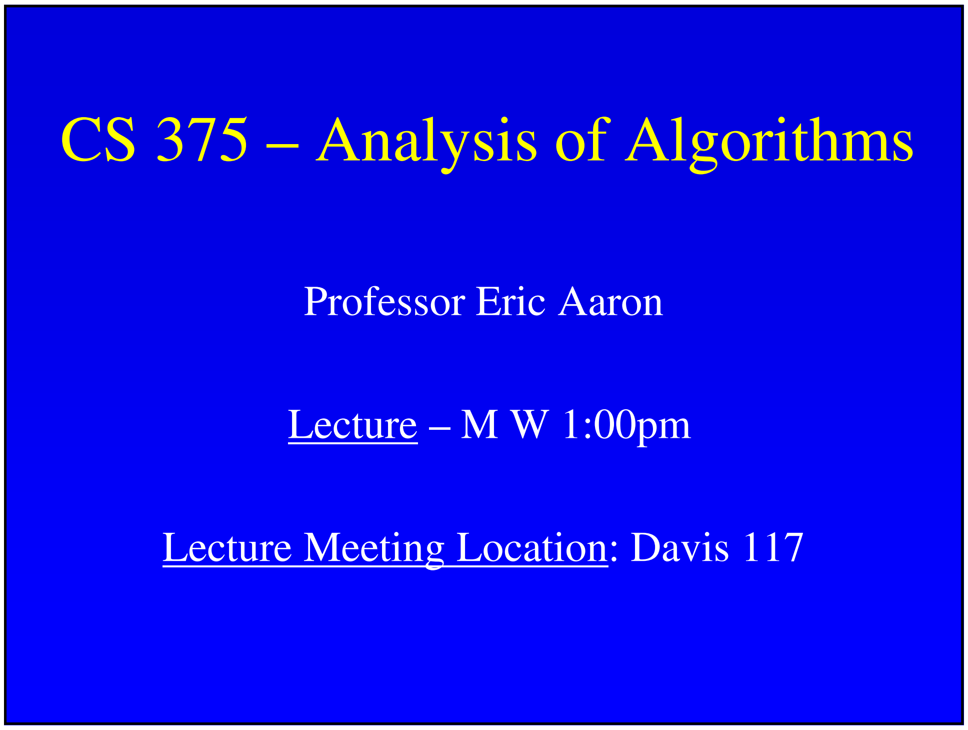
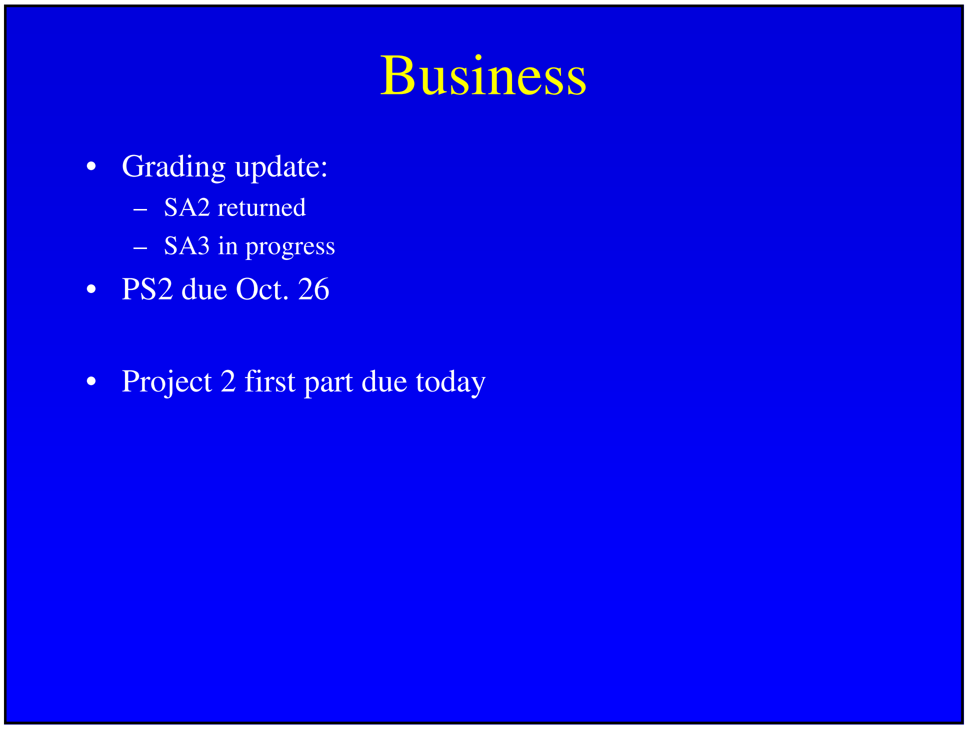


CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117



Business

- Grading update:
  - SA2 returned
  - SA3 in progress
- PS2 due Oct. 26
- Project 2 first part due today

Business: Project 2

- Project 2 out, due Nov. 3

Have you read through Proj2 yet?

Would you have preferred the assignment sheet to be broken up into separate documents?

- Parts of Project 2:
  1. **Design Exhaustive Search Algorithms**: Your team will collectively design exhaustive search algorithms for 8 problems.
  2. **Improve Time Efficiency**: Your team will pick one of the problems and make your exhaustive search algorithm more efficient.
  3. ***Reduction***: For the same problem chosen for part 2 above, you will *reduce* that problem to one of the other seven problems from part 1.
  4. **Create and Give a Presentation**: Your team will present work from the previous three parts of the assignment, *using loop invariants* where appropriate to explain correctness.

Hint: Your team may want to be strategic about which of the 8 problems you choose to focus on for your improvements, reduction, and presentation. Pick a problem for which you can do good work!

The green-shaded ones are examples of *polynomial time* classes—upper bounded by n^k for some constant *k*. Problems solvable in polynomial time are considered *tractable.* (More about this later in the semester!)

Time Complexity Classes Illustrated!

Complexity Class	*What we call it*	*Example algorithms / objects*
O(1)	Constant	Print "Hello, World!"; stack operations [and much, much more—be careful!]
O(lg n)	Log time	Binary search
O(n)	Linear	Exhaustive search of an array (linear search); Merge (as used in Mergesort)
O(n lg n)	n lg n	Mergesort; Heapsort [**Recall: sorting can be done in θ(n lg n)**]
O(n^2)	n-squared; quadratic	Insertion / selection / bubble sort; several graph algos
O(n^3)	n-cubed; cubic	*My favorite algorithm*! (a graph algo)
O(2^n)	Exponential	Number of *subsets* of a set of size n
O(n!)	Factorial	Number of orderings / permutations of elements of a list of length n

This isn't Mergesort! It's just the Merge algo used in Mergesort

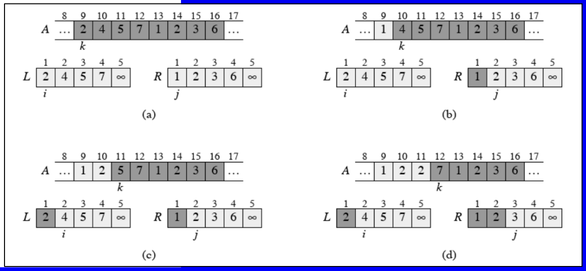
See CLRS, pg. 31

O(n): Merge

- See CLRS page 34 for specfications / loop invariant
  - In the function call *Merge(A,p,q,r)*, what are *A, p, q,* and *r*?

```
MERGE(A, p, q, r)
  n_1 = q - p + 1
  n_2 = r - q
  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
  for i = 1 to n_1
      L[i] = A[p + i - 1]
  for j = 1 to n_2
      R[j] = A[q + j]
  L[n_1 + 1] = ∞
  R[n_2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```

To me, one cool thing about this algo is how effective it can be to do a constant amount of work for each element in the input! (Sorta like with stack operations—a powerful data structure with fast operations!)



This isn't Mergesort! It's just the Merge algo used in Mergesort

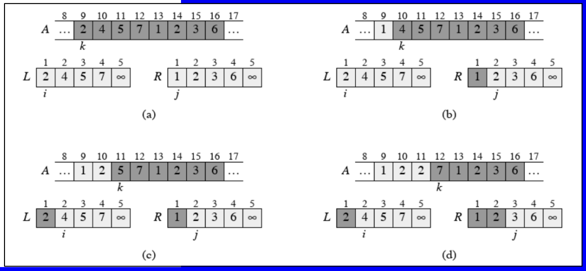
See CLRS, pg. 31

O(n): Merge

- See CLRS page 34 for specfications / loop invariant
  - In the function call *Merge(A,p,q,r)*, what are *A, p, q,* and *r*?

```
MERGE(A, p, q, r)
  n_1 = q - p + 1
  n_2 = r - q
  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
  for i = 1 to n_1
      L[i] = A[p + i - 1]
  for j = 1 to n_2
      R[j] = A[q + j]
  L[n_1 + 1] = ∞
  R[n_2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```

- How would you explain the time complexity of this algorithm?
- What is the algorithm's space complexity?

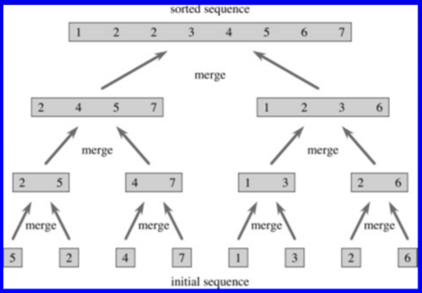


See CLRS, pg. 34

O(n lg n): Mergesort

- Mergesort is a *classic* example of an *n lg n* algorithm
  - Algo idea: Repeatedly split input list in half, sort each half separately, then Merge the two halves together
  - Uses the *Merge* function as a subroutine
  - Pretty fast algo, because *Merge* is O(n)
- Pseudocode, from CLRS:

```
MERGE-SORT(A, p, r)
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```



See CLRS, pg. 34

O(n lg n): Mergesort

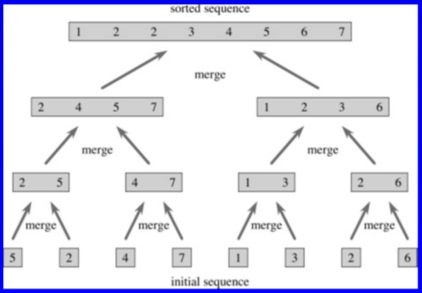
- Mergesort is a *classic* example of an *n lg n* algorithm
  - Algo idea: Repeatedly split input list in half, sort each half separately, then Merge the two halves together

How would we explain the complexity of this recursive algorithm?

(We'll get to correctness later in the semester)

- Pseudocode, from CLRS:

```
MERGE-SORT(A, p, r)
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```



See CLRS, pg. 34

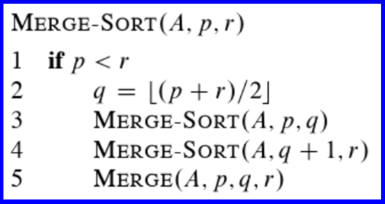
O(n lg n): Mergesort

- Mergesort is a *classic* example of an n lg n algorithm
  - Algo idea: Repeatedly split input list in half, sort each half separately, then Merge the two halves together

How would we explain the complexity of this recursive algorithm?

(We'll get to correctness later in the semester)

- Pseudocode, from CLRS:



```
MERGE-SORT(A, p, r)
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Runtime T(n) as a *recurrence:*
- ***$T(n) = 2T(n/2) + \theta(n)$***
- ***$T(1) = \theta(1)$***

Solves to T(n) = *$\theta(n \lg n)$*

We'll do *much* more of this later in the semester!

For now, focus more on where the recurrence came from and how it represents this algo's runtime, rather than on how to solve it

"Hey, Prof., weren't you gonna talk about what the fastest sorting algorithm is?"

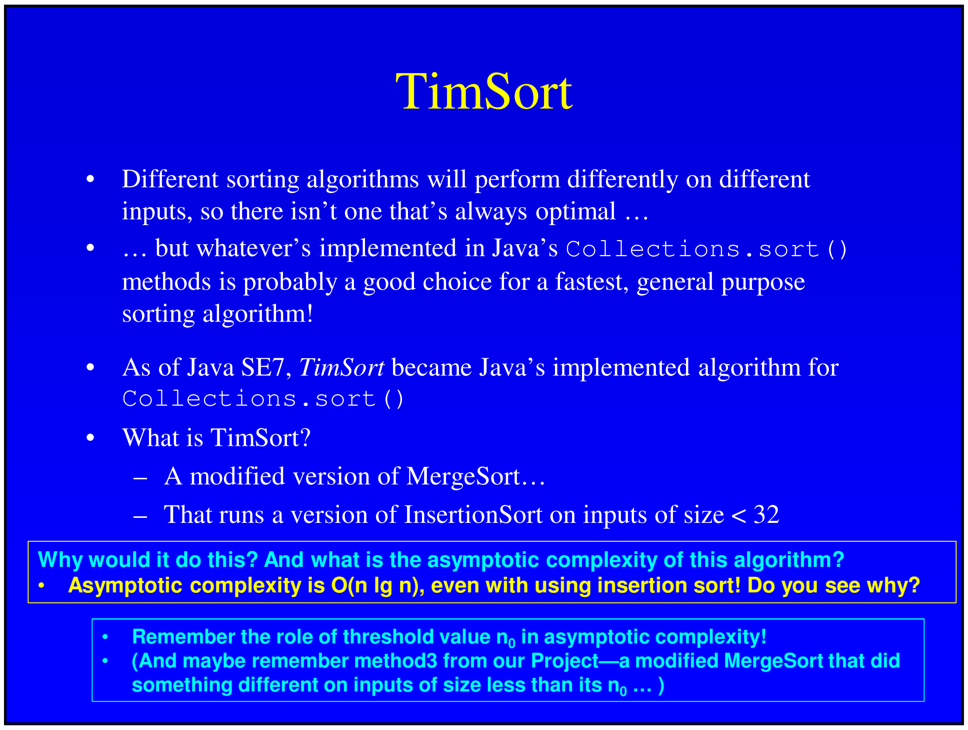
Yes!

- Different sorting algorithms will perform differently on different inputs, so there isn't one that's always optimal …
- … but whatever's implemented in Java's `Collections.sort()` methods is probably a good choice for a fastest, general purpose sorting algorithm!

TimSort

- Different sorting algorithms will perform differently on different inputs, so there isn't one that's always optimal …
- … but whatever's implemented in Java's `Collections.sort()` methods is probably a good choice for a fastest, general purpose sorting algorithm!
- As of Java SE7, *TimSort* became Java's implemented algorithm for `Collections.sort()`
- What is TimSort?
  - A modified version of MergeSort…
  - That runs a version of InsertionSort on inputs of size < 32

Why would it do this? And what is the asymptotic complexity of this algorithm?

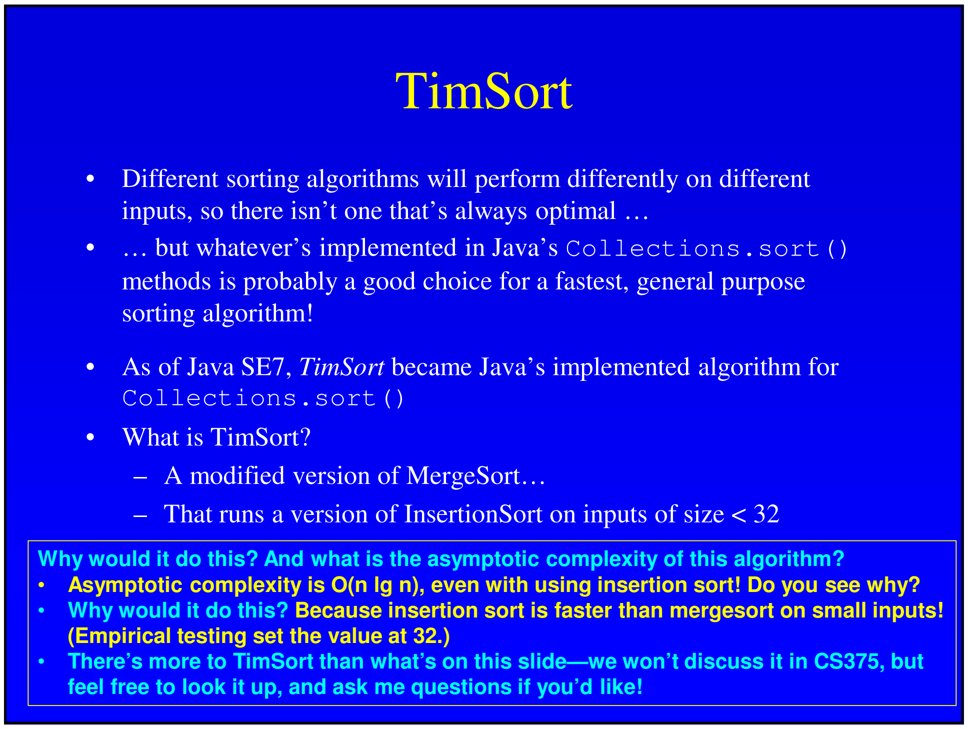


TimSort

- Different sorting algorithms will perform differently on different inputs, so there isn't one that's always optimal …
- … but whatever's implemented in Java's `Collections.sort()` methods is probably a good choice for a fastest, general purpose sorting algorithm!
- As of Java SE7, *TimSort* became Java's implemented algorithm for `Collections.sort()`
- What is TimSort?
  - A modified version of MergeSort…
  - That runs a version of InsertionSort on inputs of size < 32

Why would it do this? And what is the asymptotic complexity of this algorithm?
- **Asymptotic complexity is O(n lg n), even with using insertion sort! Do you see why?**

- **Remember the role of threshold value n_0 in asymptotic complexity!**
- **(And maybe remember method3 from our Project—a modified MergeSort that did something different on inputs of size less than its n_0 …)**



TimSort

- Different sorting algorithms will perform differently on different inputs, so there isn't one that's always optimal …
- … but whatever's implemented in Java's `Collections.sort()` methods is probably a good choice for a fastest, general purpose sorting algorithm!
- As of Java SE7, *TimSort* became Java's implemented algorithm for `Collections.sort()`
- What is TimSort?
  - A modified version of MergeSort…
  - That runs a version of InsertionSort on inputs of size < 32

Why would it do this? And what is the asymptotic complexity of this algorithm?
- **Asymptotic complexity is O(n lg n), even with using insertion sort! Do you see why?**
- **Why would it do this? Because insertion sort is faster than mergesort on small inputs! (Empirical testing set the value at 32.)**
- **There's more to TimSort than what's on this slide—we won't discuss it in CS375, but feel free to look it up, and ask me questions if you'd like!**

Common myths about recursion, debunked!

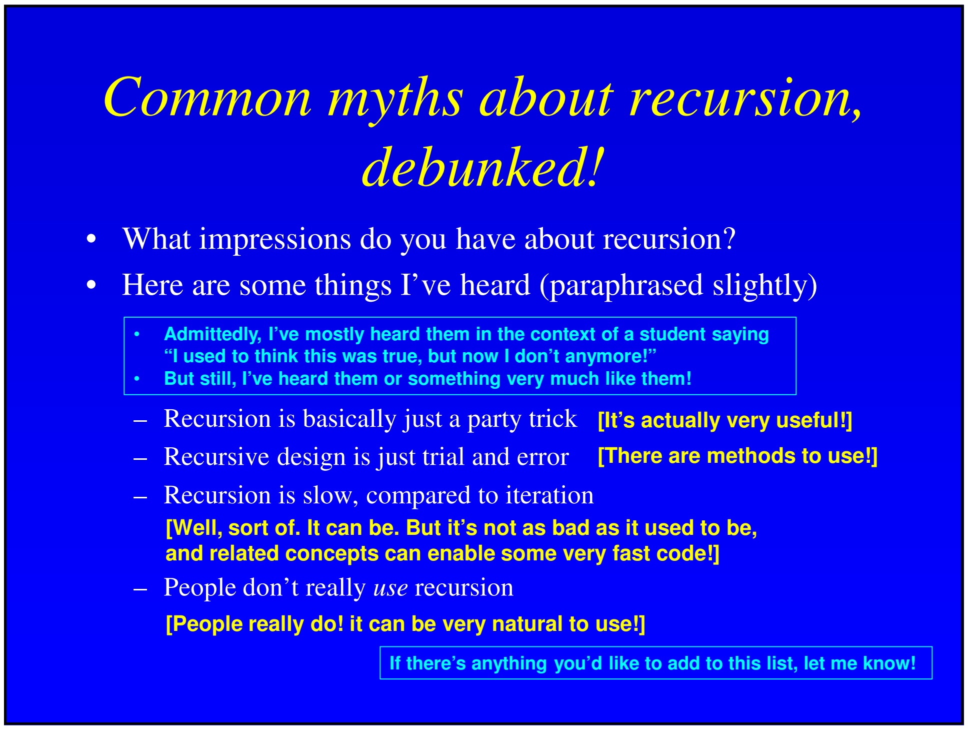
- What impressions do you have about recursion?
- Here are some things I've heard (paraphrased slightly)

  - **Admittedly, I've mostly heard them in the context of a student saying "I used to think this was true, but now I don't anymore!"**
  - **But still, I've heard them or something very much like them!**

  – Recursion is basically just a party trick
  – Recursive design is just trial and error
  – Recursion is slow, compared to iteration

  – People don't really *use* recursion

If there's anything you'd like to add to this list, let me know!



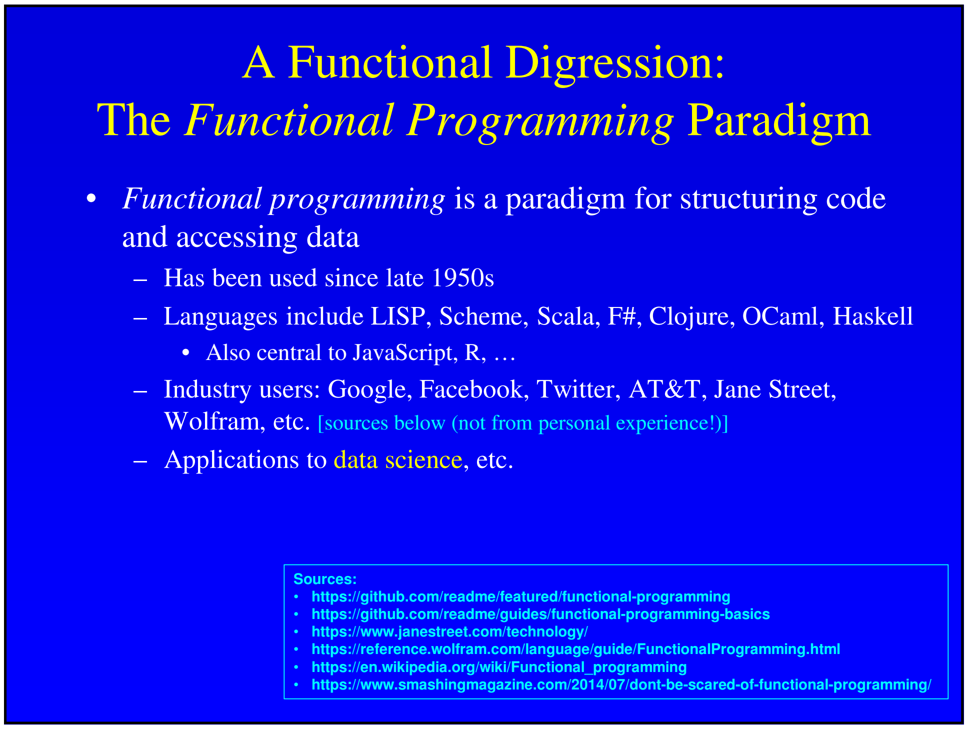
Common myths about recursion, debunked!

- What impressions do you have about recursion?
- Here are some things I've heard (paraphrased slightly)

  - **Admittedly, I've mostly heard them in the context of a student saying "I used to think this was true, but now I don't anymore!"**
  - **But still, I've heard them or something very much like them!**

  – Recursion is basically just a party trick **[It's actually very useful!]**
  – Recursive design is just trial and error **[There are methods to use!]**
  – Recursion is slow, compared to iteration
  [Well, sort of. It can be. But it's not as bad as it used to be, and related concepts can enable some very fast code!]
  – People don't really *use* recursion
  [People really do! it can be very natural to use!]

If there's anything you'd like to add to this list, let me know!



A Functional Digression: The *Functional Programming* Paradigm

- *Functional programming* is a paradigm for structuring code and accessing data
  – Has been used since late 1950s
  – Languages include LISP, Scheme, Scala, F#, Clojure, OCaml, Haskell
    - Also central to JavaScript, R, …
  – Industry users: Google, Facebook, Twitter, AT&T, Jane Street, Wolfram, etc. [sources below (not from personal experience!)]
  – Applications to data science, etc.

Sources:
- **https://github.com/readme/featured/functional-programming**
- **https://github.com/readme/guides/functional-programming-basics**
- **https://www.janestreet.com/technology/**
- **https://reference.wolfram.com/language/guide/FunctionalProgramming.html**
- **https://en.wikipedia.org/wiki/Functional_programming**
- **https://www.smashingmagazine.com/2014/07/dont-be-scared-of-functional-programming/**

A Functional Digression: The *Functional Programming* Paradigm

- Some important features of *pure* functional programming
  - Immutable data
  - Stateless functions [everything self-contained / no side effects!]
  - Uses *recursion* for iteration, rather than loops
  - *Supports parallel computation* [recall: applications to data science]
  - Code can be easier to show correct / debug

For those of us who like to think about programming languages at the theory and design level, it may be interesting to note that the last three of the above five features kinda follow from the first two!

This is subtle and well beyond the scope of CS375, but please feel free to talk with me more about it outside of class, if you'd like!

A Functional Digression: The *Functional Programming* Paradigm

- *Functional programming* is a paradigm for structuring code and accessing data

Why would I use functional programming?

When you think about well-structured software, you might think about software that's easy to write, easy to debug, and where a lot of it is reusable. That's functional programming in a nutshell! Granted, one might argue that it's not as easy to write, but let's touch on the other two points while you wrap your mind around the functional paradigm.

"Once you get used to it, it's self-evident. It's clear. I look at my function. What can it work with? Its arguments. Anything else? No. Are there any global variables? No. Other module data? No. It's just that." - Robert Virding, co-inventor of Erlang

[source: https://github.com/readme/guides/functional-programming-basics Author: Cassidy Williams]