CSSE 220 Day 29

Analysis of Algorithms continued

Recursion

Questions?

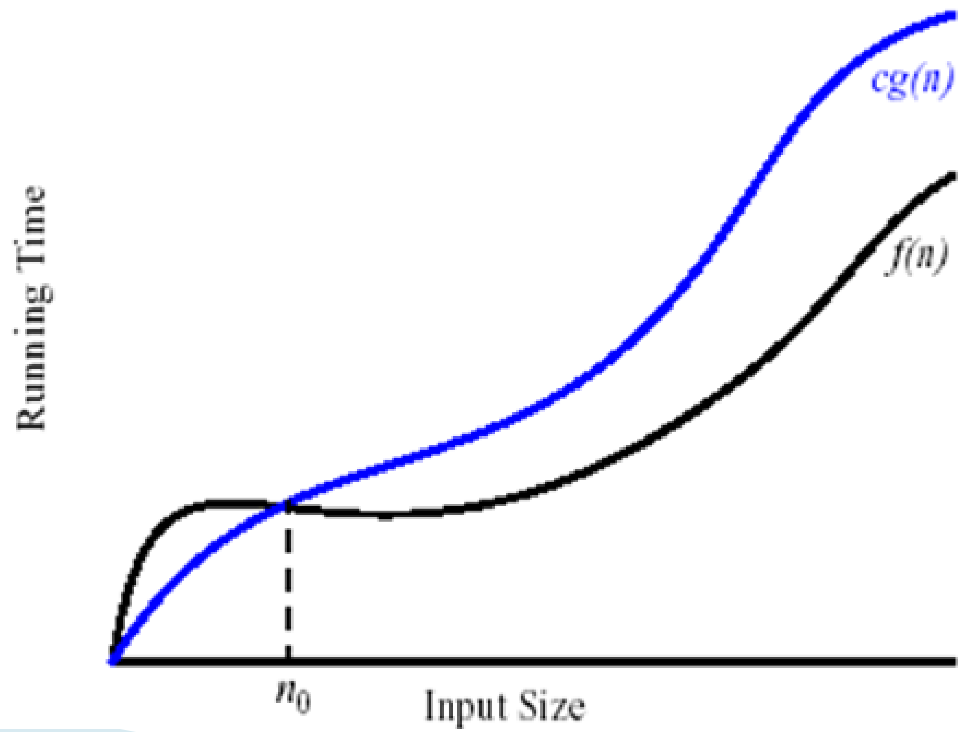
- On Capstone Project?
  - Automatic extension to Monday morning
  - If a team member does not wish to join the team in its extension-decision, see me to work it out
  - Final reflection is open – do it when you are done with project!!!
- On Exam 2?
  - Complete by now unless you have made/make arrangements with me
- On grading of Exam 1:
  - Earn back points!
  - Fix FIXME's (but keep FIXME in comment) and recommit.
  - Complete before the final exam.
- Final exam:
  - Take it either (your choice):
    - Tuesday 1 p.m. in F-231 (CSSE conference room), or
    - Friday 1 p.m. in G-313 or G-315 (your choice)
  - Open everything, HALF paper and pencil, about 90 minutes
  - Covers last few days
- Questions on anything else?

Outline of today's session

- Algorithm analysis, review
- Recursion, review
- Recursion, making it efficient
- Data structures, how to choose
- Implementation of Linked Lists
- Work on Capstone

Definition of big-Oh

- Formal:
  - We say that f(n) is O(g(n)) if and only if
  - there exist constants c and n_0 such that
  - for every $n \geq n_0$ we have
  - $f(n) \leq c \times g(n)$
- Informal:
  - f(n) is roughly proportional to g(n), for large n



Recursive Functions

- Factorial:

Base Case

$$n! = \begin{cases} 1 & \text{if } n \leq 1 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

Recursive step

- Ackermann function:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{otherwise} \end{cases}$$

Q4

Key Rules to Using Recursion

- Always have a base case that doesn't recurse
- Make sure recursive case always makes progress, by solving a smaller problem
- You gotta believe
  - Trust in the recursive solution
  - Just consider one step at a time

Course Goals for Searching and Sorting: You should be able to …

- Describe basic searching & sorting algorithms:
  - Search
    - Linear search of an UNsorted array
    - Linear seach of a sorted array (silly, but good example)
    - Binary search of a sorted array
  - Sort
    - Selection sort
    - Insertion sort
    - Merge sort
- Determine the best and worst case inputs for each
- Derive the run-time efficiency of each, for best and worst-case

Recap: Search, unorganized data

- For an ***unsorted*** / unorganized array:
  - *Linear search* is as good as anything:
    - Go through the elements of the array, one by one
    - Quit when you find the element (best-case = early) or you get to the end of the array (worst-case)
  - We'll see *mapping* techniques for unsorted but *organized* data
  - Best-case: O(1)
  - Worst-case: O(n)

Recap: Search, sorted data

- For a ***sorted*** array:
  - Linear search of a SORTED array:
    - Go through the elements starting at the beginning
    - Stop when either:
      - You find the sought-for number, or
      - You get past where the sought-for number would be

  - But binary search (next slide) is MUCH better

  - Best-case: O(1)
  - Worst-case: O(n)

Recap: Search, sorted data

```
search(Comparable[] a, int start, int stop, Comparable sought) {
    if (start > stop) {
        return NOT_FOUND;
    }

    int middle = (left + right) / 2;
    int comparison = a[middle].compareTo(sought);

    if (comparison == 0) {
        return middle;
    } else if (comparison > 0) {
        return search(a, 0, middle – 1, sought);
    } else {
        return search(a, middle + 1, stop, sought);
    }
}
```

Best-case: O(1)
Worst-case: O(log n)

Recap: Selection Sort

- Basic idea:
  - Think of the list as having a sorted part (at the beginning) and an unsorted part (the rest)
  - Find the smallest number in the unsorted part
  - Exchange it with the element at the beginning of the unsorted part (making the sorted part bigger and the unsorted part smaller)

Repeat until unsorted part is empty

Best-case: $O(n^2)$
Worst-case: $O(n^2)$

Recap: Insertion Sort

- Basic idea:
  - Think of the list as having a sorted part (at the beginning) and an unsorted part (the rest)

  - Get the first number in the unsorted part
  - Insert it into the correct location in the sorted part, moving larger values up in the array to make room

Repeat until unsorted part is empty

Best-case: O(n)
Worst-case: O(n^2)

Merge Sort

- Basic recursive idea:
  - If list is length 0 or 1, then it's already sorted
  - Otherwise:
    - Divide list into two halves
    - Recursively sort the two halves
    - **Merge** the sorted halves back together
- Analysis: use tree-based sketch…

Best-case: O(n log n)
Worst-case: O(n log n)

Outline of today's session

- Algorithm analysis, review
- Recursion, review

- Recursion, making it efficient

- Data structures, how to choose

- Implementation of Linked Lists

- Work on Capstone

What the Fib?

A more careful analysis yields a smaller base but it is still exponential.

- Why does recursive Fibonacci take so long?!?
  - Answer: it recomputes subproblems repeatedly: $O(2^n)$
- Can we fix it? Yes! Just:

1. "Memorize" every solution we find to subproblems, and

2. Before you recursively compute a solution to a subproblem, look it up in the "memory table" to see if you have already computed it

This is a classic ***time-space tradeoff***

- A deep discovery of computer science
- Tune the solution by varying the amount of storage space used and the amount of computation performed
- Studied by "Complexity Theorists"
- Used everyday by software engineers

Outline of today's session

- Algorithm analysis, review
- Recursion, review

- Recursion, making it efficient

- Data structures, how to choose

- Implementation of Linked Lists

- Work on Capstone

Data Structures

- Understanding the engineering trade-offs when storing data

Data Structures Recap

- Efficient ways to store data based on how we'll use it

- So far we've seen `ArrayLists`
  - Fast addition **to end of list**
  - Fast access to any existing position
  - Slow inserts to and deletes from middle of list

Another List Data Structure

- What if we have to add/remove data from a list frequently?

- `LinkedList`s support this:
  - Fast insertion and removal of elements
    - Once we know where they go
  - Slow access to arbitrary elements

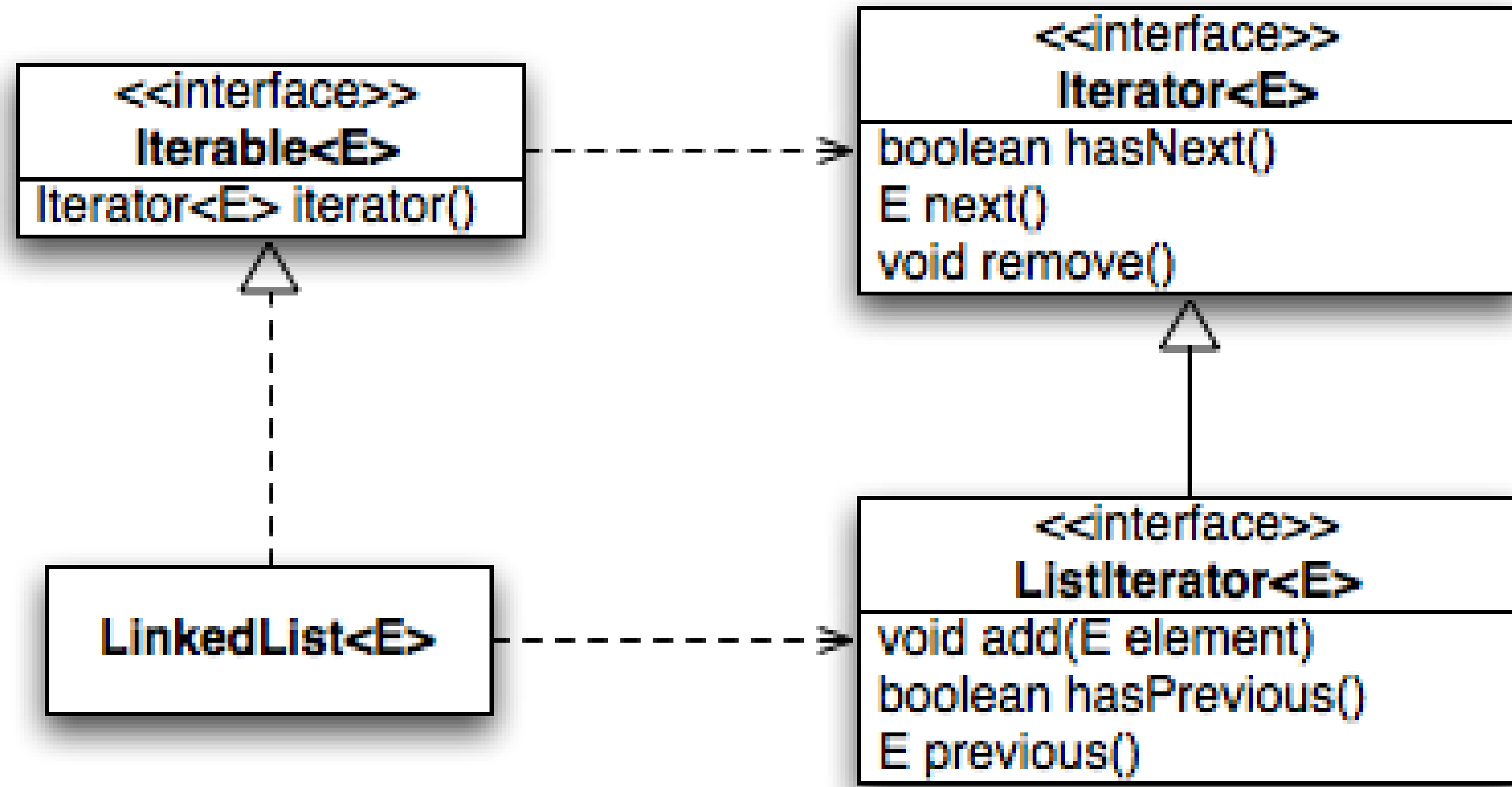
“random access”

LinkedList<E> Methods

- `void addFirst(E element)`
- `void addLast(E element)`
- `E getFirst()`
- `E getLast()`
- `E removeFirst()`
- `E removeLast()`

- What about accessing the middle of the list?
  - `LinkedList<E> implements Iterable<E>`

Accessing the Middle of a LinkedList



An Insider's View

```java
for (String s : list) {
  // do something
}
```

```java
Iterator<String> iter =
    list.iterator();

while (iter.hasNext()) {
  String s = iter.next();
  // do something
}
```

Enhanced For Loop

What Compiler Generates

Implementing LinkedList

- A simplified version, with just the essentials
- Won't implement the java.util.List interface
- Will have the usual linked list behavior
  - Fast insertion and removal of elements
    - Once we know where they go
  - Slow random access

Abstract Data Types (ADTs)

- Boil down data types (e.g., lists) to their essential operations

- Choosing a data structure for a project then becomes:
  - Identify the operations needed
  - Identify the abstract data type that most efficient supports those operations

- Goal: that you understand several basic abstract data types and when to use them

Common ADTs

- Array List
- Linked List
- Stack
- Queue
- Set
- Map

Implementations for all of these are provided by the Java Collections Framework in the **`java.util`** package.

Array Lists and Linked Lists

Operations Provided	Array List Efficiency	Linked List Efficiency
Random access	O(1)	O(n)
Add/remove item	O(n) (do you see why?)	O(1) if you are "at" the item

Q1,2

Stacks

- A last-in, first-out (LIFO) data structure
- Real-world stacks
  - Plate dispensers in the cafeteria
  - Pancakes!
- Some uses:
  - Tracking paths through a maze
  - Providing "unlimited undo" in an application

Operations Provided	Efficiency
Push item	O(1)
Pop item	O(1)

Implemented by **`Stack`**, **`LinkedList`**, and **`ArrayDeque`** in Java

Q3

Queues

- A first-in, first-out (FIFO) data structure
- Real-world queues
  - Waiting line at the BMV
  - Character on Star Trek TNG
- Some uses:
  - Scheduling access to shared resource (e.g., printer)

Operations Provided	Efficiency
Enqueue item	O(1)
Dequeue item	O(1)

Implemented by **`LinkedList`** and **`ArrayDeque`** in Java

Q4

Sets

- **Unordered** collections **without duplicates**
- Real-world sets
  - Students
  - Collectibles
- Some uses:
  - Quickly checking if an item is in a collection

Operations	HashSet	TreeSet
Add/remove item	O(1)	O(lg n)
Contains?	O(1)	O(lg n)

Can hog space

Sorts items!

Q5

Maps

- Associate **keys** with **values**
- Real-world "maps"
  - Dictionary
  - Phone book
- Some uses:
  - Associating student ID with transcript
  - Associating name with high scores

Operations	HashMap	TreeMap
Insert key-value pair	O(1)	O(lg n)
Look up value for key	O(1)	O(lg n)

Can hog space

Sorts items by key!

Q6

Outline of today's session

- Algorithm analysis, review
- Recursion, review

- Recursion, making it efficient

- Data structures, how to choose

- Implementation of Linked Lists – part of your final exam!

- Work on Capstone