PFF, Plan for Friday

- Coding examples with linked lists as low level implementation technique
  - Practicing with Practice-It
  - Foundation for DNA, linked-list assignment

- More examples of recursion and alternatives
  - Blob counting and applications of neighbor-search/Flood Fill
  - Using Queue to solve without recursion
Programming with Linked Lists

- In some ways, linked lists are inherently recursive
  - What is a list? Empty, or one element and another list: [] or [6,........]
  - null pointer or a Node with data and pointer to another Node

- Recursion can be a good choice, but not always
  - Reminder: tool in programmer tool kit
Tools: Solving Computational Problems

● **Algorithmic techniques and paradigms**
  - Brute-force/exhaustive, greedy algorithms, dynamic programming, divide-and-conquer, ...
  - Transcend a particular language
  - Designing *algorithms*, may change when turned into code

● **Programming techniques and paradigms**
  - Recursion, memo-izing, compute-once/lookup, tables, ...
  - Transcend a particular language
  - Help in making code work
    - Cope with correctness and maintenance
    - Cope with performance problems
Solving Problems Recursively

- **Recursion: indispensable in programmer's toolkit**
  - Solve a problem by creating smaller, very similar sub-problems, solve them, combine results
  - Must have a base case, no sub-problems, just do it!

- The basic idea is to get help solving a problem from coworkers (clones) who work and act like you do
  - Clone solves a simpler/smaller, but similar problem
  - Use clone's result to put together your answer

- Make recursive, clone-call and use the result
Standard list processing (recursive)

- Visit all nodes once, e.g., count them

```java
def recsize(Node list):
    if list is None:
        return 0
    return 1 + recsize(list.next)
```

- Base case is almost always empty list: null pointer
  - Must return correct value, perform correct action
  - Recursive calls use this value/state to anchor recursion
  - Sometimes one node list also used, two “base” cases

- Recursive calls make progress towards base case
  - Almost always using list.next as argument
Recursion with pictures

- Counting recursively

```java
int recsize(Node list){
    if (list == null)
        return 0;
    return 1 +
        recsize(list.next);
}

ptr
```

```java
int result = recsize(ptr);
System.out.println(result);
```
Linked list Practice

● What is a list? Empty or not: mirrored in code

```java
public Node copy(Node list) {
    if (null == list) return null;
    Node first = new Node(list.info, null);
    first.next = copy(list.next);
    return first;
}
```

● How can we replace last three lines with one?
  - return new Node(list.info, copy(list.next));
  - When constructing a list, make sure to assign to .next field!

● Can do this iteratively too, more variables/code
  - Need to add to last node, but remember first node
Recursion: adding a parameter

```java
public static int sumit(int[] a, int index){
    if (index < a.length) {
        return a[index] + sumit(a,index+1);
    }
    return 0;
}
// original call: int v = sumit(a,0);
```

- What is base case, what value is returned?
- How is progress towards base case realized?
- How is recursive call used to return a value?
- What if we sum values in a linked list?

Recursive methods sometimes use extra parameters; helper methods set this up
Two, two, two ways to reverse a list!

- We want to turn list = ['A', 'B', 'C'] into
  - rev = ['C', 'B', 'A']
  - We will move one node at a time
  - Iteratively and recursively
Iteration Step One A

- We moved ['A'] from list to the front of rev
  - rev = ['A'] and list = ['B', 'C']
  - What's the next step?
  - Invariant: rev points to reversed of nodes visited so far

```
rev  A
```
```
list  B  C
```
Iteration Step One B

- How do we make progress and maintain invariant?
  - Invariant: \( \text{rev} \) points to reversed of nodes visited so far
  - What should \('[B]'\) or \(\text{list}.\text{next}\) point to?
    - What happens if we write \(\text{list}.\text{next} = \text{rev}\)
  - What should \(\text{rev}\) point to? And \(\text{list}\) point to?

```
<table>
<thead>
<tr>
<th>rev</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>B</td>
</tr>
</tbody>
</table>
```
Iteration Step One C

- How do we make progress and maintain invariant?
  - `temp = list.next` (so we don't lose ['C'])
  - `list.next = rev` (add to front point to ['A'])
  - `rev = list` (rev points to front, ['B'])
  - `list = temp` (list updated)
Loop and method finished

- Establish invariant before loop
- Update and re-establish within loop

```java
public void reverse() {
    ListNode rev = null;
    ListNode list = front;
    while (list != null) {
        ListNode temp = list.next;
        list.next = rev;
        rev = list;
        list = temp;
    }
    front = rev;  // update state!
}
```
Reverse Recursively

- This is harder to visualize, shorter to write
  - Which method is preferred? Decide yourself

- Base case: zero or one node list, nothing to do
  - Reversing a one node list: done

- Believe in the recursion, if you reverse everything after the first node, just need to connect first node
  - Pictures are important!

![Diagram](image-url)
First the whole thing, then dissect it

- Helper method is sometimes ... helpful

```java
private ListNode doRev(ListNode list){
    if (list == null || list.next == null)
        return list;
    ListNode after = doRev(list.next);
    list.next.next = list;
    list.next = null;
    return after;
}

public void reverse(){
    front = doRev(front);
}
```
Establishing the base case

- Does the base case do the right thing?
  - What if first == null? Or a one node list?

```java
private ListNode doRev(ListNode list){
    if (list == null || list.next == null)
        return list;

    ListNode after = doRev(list.next);
    list.next.next = list;
    list.next = null;
    return after;
}

public void reverse(){
    front = doRev(front);
}
```
After recursive call, before ...

private ListNode doRev(ListNode list){
    if (list == null || list.next == null)
        return list;
    ListNode after = doRev(list.next);
    list.next.next = list;
    list.next = null;
    return after;
}
Programming with Nodes/Linked Lists

- When adding or removing nodes
  - Be sure you alter a .next field
  - Typically via re-assignment or call to new
    - `list.next = new Node()` OR `tmp` OR `recursiveCall`

- Using iteration: keep pointer to first AND current
  - Allow iteration over list, but must keep pointer to front
  - Sometimes call new before looping to have a Node
    - e.g., invariant add to a .next field in loop
    - Return `temp.next` as needed

- Recursion is sometimes simpler than iteration
  - Code mirrors structure of data!
Charles Isbell

[Georgia Tech created an inexpensive, online Computer Science Master's Degree program for $7K] Charles Isbell, a senior associate dean at the College of Computing, helped lead the effort. 

New York Times, 9/29/16

Computing also bears resemblance to the arts—the creation of artifacts— to humanities—the study of texts—and to the social sciences—the study of humans and societies.

For me, the differences are simple to state: Computationalists grok that models, languages and machines are equivalent.
WOTO questions


Sometimes recursion helps, sometimes not so much, but practice is good
Blob Counting, Flood Fill

- Flood a region with color
  - Erase region, make transparent
  - How do find the region?

- Finding regions, blobs, edges, ..
  - See blob counting code
  - What is a blob?

- Recursion helps, but necessary?
  - Performance, clarity, ...
  - Ease of development
BlobCount or edge detection or …

- How do we find images? Components? Paths?
  - [https://git.cs.duke.edu/201fall16/blobstuff/tree/master/src](https://git.cs.duke.edu/201fall16/blobstuff/tree/master/src)
Running Blobs and BlobModel

- Initialize with number of random pixels/points
  - Could also read a file of scanned gels from a lab!
  - Use Random with no seed to get different runs

- Count and label blobs of minimal size
  - Blobs contiguous horizontal/vertical connection
  - Large blobs not as plentiful as small blobs
  - Run example and look for different sizes

- We'll dissect code to understand algorithm and implementation
Counting Blobs, Flood Fill

  - See animation on page!
- The general idea is to "color" a pixel, then visit adjacent pixels and color them
  - Repeat same process for adjacent pixels
- The core of recursion – similar task, smaller problem
  - One pixel done, many to go
Ideas behind blob fill code

- **Ask your neighbors**
  - Return blob size
  - Ensure no re-counts
  - Sum and return

- **What do neighbors do?**
  - Same thing!
  - Colors indicate calls
Details and Idioms in blob code

- **Method `blobFill` has four parameters**
  - `(row,column)` of where search starts
  - Character being searched for (initially `*` or `blob`)
  - Character to fill with on success (e.g., count ‘2’ or ‘4’)
    - Mark for visualization
    - Mark to ensure we don't search again!

- **If `(row,column)` is part of blob, count it and ask neighbors for their counts**
  - They're part of blob (if never visited before)

- **Return total of yourself and neighbors**
  - Key to recursion: do one thing and ask for help
Examine code in Eclipse/Git

● int size = blobFill(5,7,'*','8')

➢ Start at (row,col) == (5,7), look for '*', fill with '8'

```java
protected int blobFill(int row, int col,
int lookFor, int fillWith) {

    int size = 0;
    if (inRange(row, col)) {
        if (myGrid[row][col] != lookFor) return 0;

        myGrid[row][col] = fillWith; // mark pixel
        size = 1; // count this pixel, then scout for neighbors
        size += blobFill(row - 1, col, lookFor, fillWith) + blobFill(row + 1, col, lookFor, fillWith) + blobFill(row, col - 1, lookFor, fillWith) + blobFill(row, col + 1, lookFor, fillWith);
    }
    return size;
}
```
Blob questions

- **What changes if diagonal cells are adjacent?**
  - Conceptually and in code

- **How do we find blob sizes in a range?**
  - Not bigger than $X$, but between $X$ and $Y$

- **How would we number blobs by size rather than by when they're found? Bigger blobs labeled with one kind of character, smaller with different**
  - Do we have the tools to do this in existing code?

- **Can we avoid recursion and do this iteratively?**
Iterative Blob Counting

- **Conceptually:** Use a queue of elements in one blob
  - Enqueue the first element of a blob
  - Put its neighbors on the queue (one step away)
  - Dequeue element, enqueue neighbors if in blob
    - But do not enqueue element seen before

- **Programmatically:** re-use most of BlobModel
  - Change two methods, use inheritance
  - More on this later, see @Override in code
  - These methods use Iterative Version
BlobModel and IterativeBlobModel

- **Iterative** inherits some methods, use those
  - Including `findBlobs`, `addView`

- **Override methods**, see `findBlobs` calling `blobFill`
  - Where is `blobFill` method? It depends

- **Basic technique of determining which method to call at runtime**
  - Cornerstone of object oriented programming
  - Studied more in later courses
floodFill and blobCounting

● In recursive version: four recursive calls
  ➢ Use result of each, sum to get total
  ➢ If we want to include diagonals? Eight calls!

● In iterative version: four neighbors enqueued
  ➢ See the offset/delta code as short-hand, could have used this in recursive version as well

● Instance variable myQueue – should it be local?
  ➢ Is it used in more than one method?
  ➢ Does it maintain state across more than one call?
WOTO questions


Sometimes recursion helps, sometimes not so much, but practice is good