Compsci 201
Recurrences, Recursion, Big-Oh

Susan Rodger
March 25, 2020
Q is for …

• Queue
  • First-in-first-out, FIFO

• Quicksort
  • So very fast. Not stable
Announcements

• Assignment P4 DNA-Link
  • Part 2 due March 26 – Code and more Analysis

• Assignment P5 will be out Friday?
  • Then P6 will be last assignment

• APT-5 is now out and due Tuesday, March 31
Plan for WASB

• Binary Trees
  • Review, Recursion, Solving Problems

• Runtime analysis of recursive algorithms
  • Recurrence relations to aid analysis
  • From intuition to understanding

• Recursion and Invariants with List Reverse
  • Thinking about recursion and iteration

• Work in 201
Syllabus Changes

• Please see:
  https://www2.cs.duke.edu/courses/spring20/compsci201/infonew.php

• Grading of exams and apt quiz – max counts more

• Participation not required for Discussions, but strongly recommended

• Lecture now taped and recorded before lecture time, includes WOTOs
  • Lecture now for questions and go over WOTOs,
  • Participation for WOTOs has been cut back, 65%

• Try to turn things in on time
  • Need a few more days, fill out the extension form
Motivation for Trees

• HashSet and HashMap are $O(1)$ average
  • Astonishing! Search, insert, delete
  • No order for keys, sometimes order matters
  • Worst-case? Everything in same locker/bucket
    • Just in case? Use a tree in that locker/bucket

• Search Trees: TreeSet and TreeMap
  • $O(\log N)$ no matter what, average and worst
  • Alphabetical or other order and range queries
    • Find all keys in range $[low, high]$ efficiently
Why Trees are O(log N)

- With each query: eliminate half of tree
  - 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1
- Can ensure trees are balanced: TreeSet/TreeMap
  - Re-balance on add or delete
Review: Inorder Traversal

• Think of search trees as recursive/hierarchical
  • Empty OR Root/Node with two subtrees
  • Root, leaves, paths, oh my!

```
30  public void print(TreeNode root) {
31      if (root != null) {
32          print(root.left);
33          System.out.println(root.info);
34          print(root.right);
35      }
36  }
```
Constructing and Printing Tree

- Code and visualize: constructor has 3 parameters

```java
public static void main(String[] args) {
    TreeNode root =
    new TreeNode("mango",
    new TreeNode("durian",
    new TreeNode("apple",null,null),
    new TreeNode("grapefruit",null,null)),
    new TreeNode("pear",
    new TreeNode("orange",null,null),
    new TreeNode("tangerine",null,null)));

    print(root);
}
```
Review: Java Comparable Interface
Java-isms for comparing

• We can compare int, double, char
  • Using ==, and !=, and <, <=, >, >=
  • Primitives use conventional symbols

• *Cannot* write "apple" < "zebra"
  • Must compare objects using specific method
  • Objects must be *comparable*, that is they must implement the *Comparable* interface
Strings are Comparable

• Compare strings lexicographically, natural ordering, dictionary order
  • “zebra” > “aardvark” but “Zebra” < “aardvark”
  • Conceptual, cannot use < or > or ==
    • We had to use `s.equals(t)` for strings/objects

• "yak".compareTo(s) returns < 0, == 0, > 0
  • s is “zebra”, “yak”, and “toad”, respectively

• The int convention also used in C++, C, others
Comparables Elements

- TreeSet<String>, TreeMap<String, Anything>
  - Tree elements must be comparable
    - Must implement Comparable<..>
  - It's possible to supply a Comparator, later

- Arrays.sort, Collections.sort
  - What algorithm is used in sorting?
  - Can change order of sort: Comparator, later
From Recursion to Recurrence

• How do we analyze runtime: recursive algorithms
  • Difficult to see how many recursive calls made sometimes
  • Difficult to account for diminishing size in recursion using intuition

• Concepts grounded in mathematical induction
  • Analysis of algorithms matches structure of recursive methods
Determine height of a tree

- Longest root-to-leaf path (# nodes):
  - Balanced trees can use height to re-balance
Determine height of a tree

- Longest root-to-leaf path (# nodes):
  - Balanced trees can use height to re-balance
Determine height of a tree

- Longest root-to-leaf path (# nodes):
  - Balanced trees can use height to re-balance
Determine height of a tree

- Longest root-to-leaf path (# nodes):
  - Balanced trees can use height to re-balance

Max is 4
Add root: +1
Height is 5
Tree function: Tree height in TreeDemo.java

- Compute tree height (longest root-to-leaf path)
  [Link](https://coursework.cs.duke.edu/201spring20/classcode/~/blob/master/src/TreeDemo.java)

```java
186 187 188 189
public int height(TreeNode root) {
    if (root == null) return 0;
    return 1 + Math.max(height(root.left), height(root.right));
}
```

- Find height of left subtree, height of right subtree
  - Combine results to determine height of tree
  - *Always use results of recursive calls*
Complexity/Efficiency

Tree has n nodes

• Intuitively: visit every node once for height: $O(n)$
  • How can we analyze this mathematically?

• Write a recurrence relation describing runtime
  • Eventually we will solve, but for now? Just write

• Let $T(n)$ be time for height to run on n-node tree
  • Then $T(0) = O(1)$
  • Then $T(n) = O(1) + 2*T(n/2)$ balanced tree
Annotated Tree height

- Compute tree height (longest root-to-leaf path)

```java
public int height(TreeNode root) {
    if (root == null) return 0;
    return 1 + Math.max(height(root.left), height(root.right));
}
```

- Base case? $O(1)$ – Recursive? $2T(n/2)$, why?
  - Work to combine results? $O(1)$
  - $T(n) = 2T(n/2) + O(1)$
- Unbalanced? $T(n) = T(n-1) + O(1)$
Annotated Tree height

• Compute tree height (longest root-to-leaf path)

```java
public int height(TreeNode root) {
    if (root == null) return 0;
    return 1 + Math.max(height(root.left), height(root.right));
}
```

• Base case? $\mathcal{O}(1)$ – Recursive? $2T(n/2)$, why?
  • Work to combine results? $\mathcal{O}(1)$
  • $T(n) = 2T(n/2) + \mathcal{O}(1)$
• Unbalanced? $T(n) = T(n-1) + \mathcal{O}(1)$

$T(0) = \mathcal{O}(1)$
Recurrence Relations

No need to derive, remember or look up. Derive?
Once here, more in 230 and 330

<table>
<thead>
<tr>
<th>Recurrence</th>
<th>Algorithm</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = T(n/2) + O(1)$</td>
<td>binary search</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(1)$</td>
<td>sequential search</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(1)$</td>
<td>tree traversal</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T(n) = T(n/2) + O(n)$</td>
<td>quicksort partition</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>mergesort, quicksort</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(n)$</td>
<td>selection or bubble sort</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>
Balanced Trees and Complexity

• A tree is height-balanced if
  • Left and right subtrees are height-balanced
  • Left and right heights differ by at most one

• Which trees are balanced?
Balanced Trees and Complexity

- A tree is height-balanced if
  - Left and right subtrees are height-balanced
  - Left and right heights differ by at most one
- Which trees are balanced?
Balanced Trees and Complexity

• A tree is height-balanced if
  • Left and right subtrees are height-balanced
  • Left and right heights differ by at most one

```java
public boolean isBalanced(Tree root) {
    if (root == null) return true;

    int lh = height(root.left);
    int rh = height(root.right);

    return isBalanced(root.left) &&
           isBalanced(root.right) &&
           Math.abs(lh - rh) <= 1;
}
```
Balanced Trees and Complexity

• A tree is height-balanced if
  • Left and right subtrees are height-balanced
  • Left and right heights differ by at most one

```java
public boolean isBalanced(Tree root) {
    if (root == null) return true;
    int lh = height(root.left);
    int rh = height(root.right);
    return isBalanced(root.left) &&
           isBalanced(root.right) &&
           Math.abs(lh - rh) <= 1;
}
```
Complexity of isBalanced

• We know that height(root) is $O(N)$ for $N$-node tree
  • Recurrence was $T(N) = 2T(N/2) + O(1)$

• Recurrence for isBalanced (average case)
  • $T(N)$ is time for isBalanced on $N$-node tree
  • Call height twice: $O(N)$, each tree $N/2$ nodes
  • Make two recursive calls $2T(N/2)$
  • Recurrence: $T(N) = 2T(N/2) + O(N)$
Annotating every line

• Let $T(n)$ be *run time* for `isBalanced` for roughly-balanced $N$-node tree: $2T(n/2) + O(n)$

• Why roughly balanced?

  • Recurrence for unbalanced? All but one node in right subtree: $T(n) = O(n) + T(n-1)$
Annotating every line

• Let $T(n)$ be *run time* for `isBalanced` for roughly-balanced $N$-node tree: $2T(n/2) + O(n)$

• Why roughly balanced?
  • Recurrence for unbalanced? All but one node in right subtree: $T(n) = O(n) + T(n-1)$

```java
public boolean isBalanced(TreeNode root) {
    if (root == null) return true;

    int lh = height(root.left);          // O(n)
    int rh = height(root.right);         // O(n)

    return isBalanced(root.left) &&      // T(n/2)
           isBalanced(root.right) &&      // T(n/2)
           Math.abs(lh - rh) <= 1;        // O(1)
}
```
Recurrence Relations

No need to derive, remember or look up. Derive?
Once here, more in 230 and 330

<table>
<thead>
<tr>
<th>Recurrence</th>
<th>Algorithm</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = T(n/2) + O(1)$</td>
<td>binary search</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(1)$</td>
<td>sequential search</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(1)$</td>
<td>tree traversal</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T(n) = T(n/2) + O(n)$</td>
<td>quicksort partition</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>mergesort, quicksort</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(n)$</td>
<td>selection or bubble sort</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>
Solve recurrence

• T(n) is time for isBalanced to run on n-node tree
  • T(n) = 2T(n/2) + n

Eureka, pattern!
• Holds n = 1, 2, … Let n = 2^k, so k=log_2 n

• nT(1) + n log(n)

We now have solution to recurrence!
• O(n log n)  -- base 2, but base doesn’t matter
Solve recurrence

- $T(n)$ is time for `isBalanced` to run on $n$-node tree
  - $T(n) = 2T(n/2) + n$
  - $= 2 \left[ 2(T(n/4) + n/2) \right] + n$
  - $= 4T(n/4) + 2n$
  - $= 8T(n/8) + 3n$
  - $= 2^kT(n/2^k) + kn$

Eureka, pattern!

We plugged $n/2$ into $T(n) = 2T(n/2) + n$

Note: $T(n/2) = 2T((n/2)/2) + n/2$
Solve recurrence

- T(n) is time for `isBalanced` to run on n-node tree
  - T(n) = 2T(n/2) + n
  - = 2 [2 (T(n/4) + n/2) + n
  - = 4T(n/4) + n + n = 4T(n/4) + 2n

Simplify
Solve recurrence

• $T(n)$ is time for `isBalanced` to run on $n$-node tree
  • $T(n) = 2T(n/2) + n$
  • $= 2 \left[ 2 \left( T(n/4) + n/2 \right) \right] + n$
  • $= 4T(n/4) + n + n = 4T(n/4) + 2n$
  • $= 4 \left[ 2T(n/8) + n/4 \right] + 2n$

We plugged $n/4$ into $T(n) = 2T(n/2) + n$
Solve recurrence

• $T(n)$ is time for `isBalanced` to run on $n$-node tree
  • $T(n) = 2T(n/2) + n$
  • $= 2 [2(T(n/4) + n/2)] + n$
  • $= 4T(n/4) + n + n = 4T(n/4) + 2n$
  • $= 4 [2T(n/8) + n/4] + 2n$
  • $= 8T(n/8) + 3n$

Simplify

$O(n \log n)$ — base 2, but base doesn’t matter
Solve recurrence

- \( T(n) \) is time for isBalanced to run on \( n \)-node tree
  - \( T(n) = 2T(n/2) + n \)
  - \( = 2 \left[ 2 \left( T(n/4) + n/2 \right) \right] + n \)
  - \( = 4T(n/4) + n + n = 4T(n/4) + 2n \)
  - \( = 4 \left[ 2T(n/8) + n/4 \right] + 2n \)
  - \( = 8T(n/8) + 3n \)
  - \( = 2^kT(n/2^k) + kn \)  
    - Eureka, pattern!

- Next pattern is:
  - \( 16T(n/16) + 4n \)
Solve recurrence

- $T(n)$ is time for `isBalanced` to run on $n$-node tree
  - $T(n) = 2T(n/2) + n$
  - $= 2 \left[ 2(T(n/4) + n/2) \right] + n$
  - $= 4T(n/4) + n + n = 4T(n/4) + 2n$
  - $= 4 \left[ 2T(n/8) + n/4 \right] + 2n$
  - $= 8T(n/8) + 3n$
  - $= 2^kT(n/2^k) + kn$  \textbf{Eureka, pattern!}
  - Holds $n = 1, 2, \ldots$ Let $n = 2^k$, so $k = \log_2 n$
Solve recurrence

• \( T(n) \) is time for isBalanced to run on \( n \)-node tree
  • \( T(n) = 2T(n/2) + n \)
  • \( = 2 \left[ 2(T(n/4) + n/2) \right] + n \)
  • \( = 4T(n/4) + n + n = 4T(n/4) + 2n \)
  • \( = 4 \left[ 2T(n/8) + n/4 \right] + 2n \)
  • \( = 8T(n/8) + 3n \)
  • \( = 2^kT(n/2^k) + kn \)  \( \text{Eureka, pattern!} \)
• Holds \( n = 1, 2, \ldots \) Let \( n = 2^k \), so \( k = \log_2 n \)
  • \( = nT(1) + n \log(n) \)
Solve recurrence

- $T(n)$ is time for `isBalanced` to run on $n$-node tree
  - $T(n) = 2T(n/2) + n$
  - $= 2 \left[ 2(T(n/4) + n/2) \right] + n$
  - $= 4T(n/4) + n + n = 4T(n/4) + 2n$
  - $= 4 \left[ 2T(n/8) + n/4 \right] + 2n$
  - $= 8T(n/8) + 3n$
  - $= 2^kT(n/2^k) + kn$  
    
    **Eureka, pattern!**

- Holds $n = 1, 2, \ldots$. Let $n = 2^k$, so $k = \log_2 n$
  - $= nT(1) + n \log(n)$

- We now have solution to recurrence!
  - $O(n \log n)$ -- base 2, but base doesn't matter
Recurrence relations

• You will solve recurrence relations if you take CompSci 230 or 330
• After solving you would use proof by induction to prove the recurrence relation you came up with is correct.
  
• In this class just solved one so you could see.
• We will not solve or prove by induction
• We will just form recurrence relations and look up!
Recurrence Relations

No need to derive, remember or look up

<table>
<thead>
<tr>
<th>Recurrence</th>
<th>Algorithm</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(n) = T(n/2) + O(1)</td>
<td>binary search</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>T(n) = T(n-1) + O(1)</td>
<td>sequential search</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>T(n) = 2T(n/2) + O(1)</td>
<td>tree traversal</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>T(n) = T(n/2) + O(n)</td>
<td>quicksort partition</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>T(n) = 2T(n/2) + O(n)</td>
<td>mergesort, quicksort</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>T(n) = T(n-1) + O(n)</td>
<td>selection or bubble sort</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>
Problem Statement

Write method rewire that returns a new tree, isomorphic (same shape) as the tree parameter, but in which each node's info field is equal to that node's height in the tree.

Class

```java
public class HeightLabel {
    public TreeNode rewire(TreeNode t) {
        // replace with working code
        return null;
    }
}
```

Must use TreeNode class for Tree APTs

Not nested in file!

Must be a class in your folder of APTs
Tree APTs: Thinking and Analyzing with HeightLabel

• Create a new tree from a tree parameter
  • [https://www2.cs.duke.edu/csed/newapt/heightlabel.html](https://www2.cs.duke.edu/csed/newapt/heightlabel.html)
  • Same shape, but nodes labeled with tree-size
  • Must call `new TreeNode`. With what values …
public TreeNode rewire(TreeNode t) {
    if (t == null) return null;
    return new TreeNode(height(t),
                        rewire(t.left),
                        rewire(t.right));
}

private int height(TreeNode t) {
    if (t == null) return 0;
    return 1 + Math.max(height(t.left),
                         height(t.right));
}
Creating a New Tree Recursively

• root == null: Not interesting, but necessary
  • Easy to make a copy of nothing?

• Create one new node, recursion creates all others
  • Constructor: value, left subtree, right subtree
  • What value stored in node? Call height
  • What values stored in subtrees? Recurse
Tree APTs

• Must include TreeNode class in project src
  • Similar to ListNode, not submitted as code

• Helper method often useful: add a parameter
  • Can be ArrayList<.> or String or …
  • Also useful to add instance variables

• For running/testing outside of APT system?
  • Create tree in main to test (see prev slide)
public TreeNode rewire(TreeNode t) {
    if (t == null) return null;
    return new TreeNode(height(t),
        rewire(t.left),
        rewire(t.right));
}

private int height(TreeNode t) {
    if (t == null) return 0;
    return 1 + Math.max(height(t.left),
        height(t.right));
}

When balanced: \( T(n) = 2T(n/2) + O(n) \)

When not balanced?
public TreeNode rewire(TreeNode t) {
    if (t == null) return null;
    return new TreeNode(height(t),
        rewire(t.left),
        rewire(t.right));
}

private int height(TreeNode t) {
    if (t == null) return 0;
    return 1 + Math.max(height(t.left),
        height(t.right));
}

When balanced: \( T(n) = 2T(n/2) + O(n) \)
When not balanced?
Turing Award 2017: John Hennessy and David Patterson


Patterson: “Computer science is becoming an extraordinarily popular major,” Patterson said. “Berkeley is a diverse campus, and what’s happening with popularity is that the field is getting more diverse. All of us in the field who have been here a long time think that’s just wonderful.”

In response to James Damore memo

Computer Architecture
RISC processors
Comp sci 201
Recurrences, Recursion, Big-Oh
Lecture Part 2

Susan Rodger
March 25, 2020
Is Recursion Important?

- Solving problems is important
- Scaling solutions is important
- Knowledge of tools to solve and scale are important
- You'll be expected to understand recursion!
  - Interviews, later courses, fundamental Compsci
  - You can always solve a problem iteratively
    - May be less elegant, may be harder to code

- But, the tool itself? ....
LeafSum APT

- [https://www2.cs.duke.edu/csed/newapt/leafsum.html](https://www2.cs.duke.edu/csed/newapt/leafsum.html)
- Sum all the values in leaves of tree
  - Base cases?
  - Recursive calls?

- What should big-Oh be?
  - N-node tree?
  - Visit each node ….
  - Balanced or stringy …
  - Similar to tree height
LeafSum correct?

- What do we do with null tree? Why?
  - From base-case to combining recursive calls
  - Recurrence expression?

```java
public class LeafSum {
    public int sum(TreeNode t) {
        if (t == null) return 0;
        // something is missing here!
        return sum(t.left) + sum(t.right);
    }
}
```
LeafSum Now Correct!

• What do we do with a leaf, why?
  • Why else isn’t needed, but ok
  • O(n) for all trees: two calls + O(1)

```java
public class LeafSum {
    public int sum(TreeNode t) {
        if (t == null) return 0;
        if (t.left == null && t.right == null) return t.info;
        return sum(t.left) + sum(t.right);
    }
}
```
Now solve APT Leaf Sum

- What is the Big-Oh of Leaf sum?
Now solve APT Leaf Sum

• What is the Big-Oh of Leaf sum?

• \( T(n) = 2 \ T(n/2) + O(1) \)
• What is this? Look up in the table.
Recurrence Relations

No need to derive, remember or look up

<table>
<thead>
<tr>
<th>Recurrence</th>
<th>Algorithm</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(n) = T(n/2) + O(1) )</td>
<td>binary search</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>( T(n) = T(n-1) + O(1) )</td>
<td>sequential search</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>( T(n) = 2T(n/2) + O(1) )</td>
<td>tree traversal</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>( T(n) = T(n/2) + O(n) )</td>
<td>quicksort partition</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>( T(n) = 2T(n/2) + O(n) )</td>
<td>mergesort, quicksort</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>( T(n) = T(n-1) + O(n) )</td>
<td>selection or bubble sort</td>
<td>( O(n^2) )</td>
</tr>
</tbody>
</table>
Now solve APT Leaf Sum

• What is the Big-Oh of Leaf sum?

• T(n) = 2 T(n/2) + O(1)
• What is this? Look up in the table. O(n)
I’ve just done 2 APTs for you

- You should try to do them without looking at the notes!
Recurrences, Recursion, Big-Oh
Lecture Part 3
Sometimes Recursion Isn't Simple

• How do we reverse nodes in a linked list?
  • Similar to DNA Reverse, but different
  • Iterative and Recursive Solutions
Thinking with Invariants: Reverse

• We want to turn list = ['A', 'B', 'C'] into
  • rev = ['C', 'B', 'A']
• We will move one node at a time, no new nodes!
  • Iteratively and recursively
Iteration Step One A

- We moved ['A'] from list to the front of rev
  - rev = ['A'] and list = ['B', 'C']
  - Invariant: rev points to front of reversed so far
    - What was rev before this step? null
Iteration Step One B

- How to make progress and maintain invariant?
  - Invariant: \texttt{rev} points to front reversed-so-far
  - What should ['B'] or \texttt{list.next} point to?
    - What happens if we write \texttt{list.next = rev}
- What should \texttt{rev} point to? And \texttt{list} point to?
Iteration Step One C

- Making 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)
Iteration Step One C

• Making 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)

```
rev
    v
  A --> list --> B --> C
```
Loop and method finished

- Establish invariant: rev is reversed-so-far (null)
- Update and re-establish within loop

```java
public void reverse(ListNode front) {
    ListNode rev = null;
    ListNode list = front;
    while (list != null) {
        ListNode temp = list.next;
        list.next = rev;
        rev = list;
        list = temp;
    }
    front = rev;  // update state!
}
```
Look at reverse2 - ListReverse.java

https://coursework.cs.duke.edu/201spring20/classcode/-/blob/master/src/ListReverse.java

```java
private Node reverse2(Node list) {
    Node rev = null;
    while (list != null) {
        Node temp = list.next;
        rev = new Node(list.info, rev);
        list = temp;
    }
    return rev;
}
```
Look at `reverse2` - ListReverse.java

```java
private Node reverse2(Node list) {
    Node rev = null;
    while (list != null) {
        Node temp = list.next;
        Node newNode = new Node(list.info, rev);
        rev = newNode;
        // rev = new Node(list.info, rev);
        list = temp;
    }
    return rev;
}
```

Alternatively, break it up more. Replace line 56 with lines 54 and 55
Look at revrec – ListReverse.java

```java
private Node revRec(Node list) {
    if (list == null || list.next == null) {
        return list;
    }
    Node after = revRec(list.next);
    list.next.next = list;
    list.next = null;
    return after;
}
```
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
  • Reverse all nodes after first, reconnect
  • Pictures are important!
Are you kidding? What? Dissect

- This method returns the list passed, reversed

```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;
}
```
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;
}
```

```java
public void reverse(){
    front = doRev(front);
}
```
After call, value of list.next?

```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;
}
```
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;
}
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;
}
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;
}
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;
}
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;
}
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;
}
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;
}
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;
}
Back from First Recursive Call

```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;
}
```
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;
}
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;
}
This code really does reverse the list!

- This method returns the list passed, reversed

```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;
}
```
Runtime of Reverse?

• We are reversing an n-node list, likely $O(n)$?
  • Iterative version visits every node and does $O(1)$ work each time loop iterates
  • Recursive version: $T(n) = T(n-1) + O(1)$

• Define $T$ and define $n$. Or use another letter
  • Let $R(n)$ be the time for reverse to execute with an n-node list: $R(n) = R(n-1) + O(1)$
# Recurrence Relations

No need to derive, remember or look up

<table>
<thead>
<tr>
<th>Recurrence</th>
<th>Algorithm</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = T(n/2) + O(1)$</td>
<td>binary search</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(1)$</td>
<td>sequential search</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(1)$</td>
<td>tree traversal</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T(n) = T(n/2) + O(n)$</td>
<td>quicksort partition</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>mergesort, quicksort</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(n)$</td>
<td>selection or bubble sort</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>
WOTO – Correctness counts

Edsger Dijkstra

- Goto considered Harmful
- Shortest Path algorithm
- Turing Award

For me, the first challenge for computing science is to discover how to maintain order in a finite, but very large, discrete universe that is intricately intertwined. And a second, but not less important challenge is how to mold what you have achieved in solving the first problem, into a teachable discipline: it does not suffice to hone your own intellect ... you must teach others how to hone theirs. The more you concentrate on these two challenges, the clearer you will see that they are only two sides of the same coin: teaching yourself is discovering what is teachable EWD 709