Recurrence Relations

(or: The Big-O of Recursive Functions)

A Big-O Warmup

```
// Assume strings has size n.
// Assume "robot" appears in strings.
public int findRobot(String[] strings) {
   int current = 0;
   while (!strings[current].equals("robot")) {
      current++;
   }
   return current;
}
```

A Big-O Warmup

```
// Assume strings has size n.
// Assume "robot" appears in strings.
public int findRobot(String[] strings) {
   int current = 0;
   while (!strings[current].equals("robot")) {
      current++;
   }
   return current;
}
```

1. $O(n^2)$ 2. $O(1)$ 3. $O(n)$ 4. $O(\log n)$ 5. $O(n \log n)$

A (harder!) Big-O Warmup

```
// Return the index of v in the sorted array, or -1 if it's not there.
// Search between the indices low and and high.
public int findInSorted(int[] sorted, int v, int low, int high) {
   while (low < high) {
      int midpoint = (low + high) / 2;
      if (v < sorted[midpoint]) {
         high = midpoint;  // Search in the lower half.
      } else if (v > sorted[midpoint]) {
         low = midpoint + 1;  // Search in the upper half.
      } else {
         return midpoint;
      }
   }
   return -1;
}
```

A (harder!) Big-O Warmup

```
// Return the index of v in the sorted array, or -1 if it's not there.
// Search between the indices low and and high.
public int findInSorted(int[] sorted, int v, int low, int high) {
   while (low < high) {
      int midpoint = (low + high) / 2;
      if (v < sorted[midpoint]) {
         high = midpoint;  // Search in the lower half.
      } else if (v > sorted[midpoint]) {
         low = midpoint + 1;  // Search in the upper half.
      } else {
         return midpoint;
      }
   }
   return -1;
}
```

1. $O(n^2)$ 2. $O(1)$ 3. $O(n)$ 4. $O(\log n)$ 5. $O(n \log n)$

A problem

```
// Is the value v contained in the binary search tree rooted at node?
public boolean BSTcontainsValue(int v, TreeNode node) {
   if (node == null) {
      return false;
   }
   if (node.value == v) {
      return true;
   }

   if (v < node.value) {
      return BSTcontainsValue(v, node.left);
   } else {
      return BSTcontainsValue(v, node.right);
   }
}
```

A problem

```
// Is the value v contained in the binary search tree rooted at node?
public boolean BSTcontainsValue(int v, TreeNode node) {
   if (node == null) {
      return false;
   }
   if (node.value == v) {
      return true;
   }

   if (v < node.value) {
      return BSTcontainsValue(v, node.left);
   } else {
      return BSTcontainsValue(v, node.right);
   }
}
```

1. What's the Big-O if the tree is balanced?
2. What's the Big-O if it isn't?

What you'll need to do (or not)

1. Write down the recurrence relation.

2. *Solve the recurrence relation.* *You won't need to do this in this class*

3. Compute the Big-O from that solution.

A problem

```
// Is the value v contained in the binary search tree rooted at node?
public boolean BSTcontainsValue(int v, TreeNode node) {
   if (node == null) {
      return false;
   }
   if (node.value == v) {
      return true;
   }

   if (v < node.value) {
      return BSTcontainsValue(v, node.left);
   } else {
      return BSTcontainsValue(v, node.right);
   }
}
```

1. What's the Big-O if the tree is balanced?
2. What's the Big-O if it isn't?

A problem

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

http://goo.gl/UwqvK

A problem

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

http://goo.gl/UwqvK

http://goo.gl/8V6kS

A problem

```
public boolean isBalanced(TreeNode node) {
   int left = computeHeight(node.left);
   int right = computeHeight(node.right);
   if (Math.abs(left - right) > 1) {
      return false;
   }

   return (isBalanced(node.left) &&
           isBalanced(node.right));
}
```

The Table to end all Tables

Recurrence	An algorithm with this recurrence	Running Time
$T(n) = T(n/2) + O(1)$	Binary Search	$O(\log n)$
$T(n) = T(n-1) + O(1)$	Linear Search	$O(n)$
$T(n) = 2T(n/2) + O(1)$	Tree traversal	$O(n)$
$T(n) = 2T(n/2) + O(n)$	QuickSort	$O(n \log n)$
$T(n) = T(n-1) + O(n)$	BubbleSort	$O(n^2)$

Two more

```
public int maximum(int[] values, int low, int high) {
   if (low == high) {
      return values[low];
   }
   int mid = (low + high) / 2;
   return Math.max(maximum(values, low, mid),
                   maximum(values, mid+1, high));
}
```

```
   // Reverse the array values, between the indices low and high.
   public static void reverse(int[] values, int low, int high) {
      if (low >= high) {
         return;
      }
      int temp = values[low];
      values[low] = values[high];
      values[high] = temp;
      reverse(values, low+1, high-1);
   }
```

http://goo.gl/YxiK4