CS 375 – Analysis of Algorithms

Professor Eric Aaron

**Lecture** – M W 1:00pm

**Lecture Meeting Location:** Davis 117

---

**Business**

- HW1 due already
- HW2 out tonight, due Monday, Feb. 17
Zen and The Art Of Algorithm Design

• A couple of Generally Good Ideas (principles) to help you design your algorithms (and their implementations)

1. The foundations—i.e., relevant definitions and data structures—should be as simple as possible while still providing all needed functionality

2. Let the foundations guide the development and analysis of algorithms based on them

   - I might restate principle 1 as “Keep your foundations simple”
   - I might restate principle 2 as “Let your definitions tell you what to do”

• Let’s apply this to binary trees…

Definition of an \textit{IntBinTree} Data Structure

• Throughout CS375, we will sometimes refer to an \textit{IntBinTree} data structure, representing a binary tree of integers

• In English, we’d say an IntBinTree is:
  – Either empty,
  – Or
  • an int, called \textit{val}
  • and two \textit{subtrees}, called \textit{left} and \textit{right}, that are also IntBinTrees

• Programmers might be used to seeing it more like this

\texttt{Definition IntBinTree: Empty, or... int \textit{val} \# the int value; not empty intBinTree \textit{left} \# the left subtree intBinTree \textit{right} \# the right subtree}

\texttt{Is this a good definition? Consider Principle 1: Keep your foundations simple...}

\texttt{Is this definition equivalent to the English one above?}

\texttt{The fact that a tree could be empty is implicit here, as in many implementations}
IntBinTrees: An Exercise

• Design an algorithm to return the number of levels in an IntBinTree

• Because the definition of IntBinTree is recursive, it makes sense that algorithms over it would be recursive

A Review of Recursive Design (Divide and Conquer)

• Every recursive algorithm has the following components
  – Base case(s): One or more small case(s) for which it is easy to identify (or compute) and return a solution
  – Recursive case(s): One or more cases in which the algorithm calls itself on a smaller instance of its input
    • Divide: The algorithm must break the original problem (input) down into smaller sub-problems (sub-inputs) on which the algo can be called recursively
    • “Conquer”: The algorithm must solve each of the sub-problems
    • Combine: The algorithm must combine / employ the solutions of sub-problems into a solution of the original problem
  – Recursive cases bring input closer to terminating in a base case

It’s really important that recursions terminate
IntBinTrees: An Exercise

- Design an algorithm to return the number of levels in an IntBinTree
  - Reminder: In English, an IntBinTree is either empty or an int and two subtrees
- For our recursive algorithm to compute the number of levels...
  - What are the input and output?
  - What input does the base case check for?
  - What’s the intended output for the base case?
  - How many recursive calls will the algorithm make?
  - How do we use output from recursive call(s) to compute the output on the given input?

Algorithm: Levels(T)
// Input: IntBinTree T
// Output: integer, number of levels in T
if T is empty
  return 0
else
  return max(Levels(T.left), Levels(T.right)) + 1

How would we explain this algorithm’s correctness?
Correctness of Recursive Algorithms: Inductive Arguments

• When arguing the correctness of a recursive algorithm, the general form is that of an inductive argument
• The explanation follows the structure of the algorithm
  – Show that the algorithm’s base case returns correct output
  – Show that the recursive cases return correct output…
    under the assumption that all recursive calls return correct output

Just like when creating recursive code—we assume recursive calls work in the recursive case!

• Here, how would that work?
• How would we explain the base case? The recursive case?

Algorithm: Levels(T)
//Input: IntBinTree T
//Output: integer, number of levels in T
if T is empty
  return 0
else
  return max(Levels(T.left), Levels(T.right)) + 1

Correctness of Recursive Algorithms: Inductive Arguments

• When arguing the correctness of a recursive algorithm, the general form is that of an inductive argument
• The explanation follows the structure of the algorithm
  – Show that the algorithm’s base case returns correct output
  – Show that the recursive cases return correct output…
    under the assumption that all recursive calls return correct output

Just like when creating recursive code—we assume recursive calls work in the recursive case!

• As part of explaining recursive case(s), also explain how we know the algo terminates
  – Show arguments in recursive calls get closer to base case

Algorithm: Levels(T)
//Input: IntBinTree T
//Output: integer, number of levels in T
if T is empty
  return 0
else
  return max(Levels(T.left), Levels(T.right)) + 1
Another IntBinTrees Exercise: **Search**

- The *search problem* on IntBinTrees asks if an int is anywhere in an IntBinTree.
- How would we write the problem specification for `IBTSearch`?
  - What would be the input?
  - What would be correct output?
- How would we design an algorithm to solve it?
  - Would the algorithm be iterative or recursive?
- How would we argue its correctness?

**Definition IntBinTree:** Empty, or...
- int val # int value
- intBinTree left # left subtree
- intBinTree right # right subtree

Algorithm: `IBTSearch(i, T)`
//Input: int i, IntBinTree T
//Output: True exactly when i is in T, False otherwise
Another IntBinTrees Exercise: Search

- The search problem on IntBinTrees asks if an int is anywhere in an IntBinTree
- How would we write the problem specification for IBTSearch?
  - Input: an int \( i \), and an IntBinTree \( T \)
  - Output: True exactly when \( i \) is in \( T \), False otherwise

Definition IntBinTree: Empty, or...
- int val # int value
- intBinTree left # left subtree
- intBinTree right # right subtree

Algorithm: IBTSearch\((i, T)\)
//Input: int \( i \), IntBinTree \( T \)
//Output: True exactly when \( i \) is in \( T \), False otherwise
if \( T == \) empty
  return False
else
  if \( T.val == i \)
    return True
  else
    return IBTSearch\((i, T.left)\) or IBTSearch\((i, T.right)\)

The last line uses the Boolean operator or, which is inclusive—it is True when either or both operands are True

Some people might view this as having two base cases.
The definition of IntBinTree, however, has one base case (empty tree). I view the algo as following that definition.
Another IntBinTrees Exercise: Search

- The search problem on IntBinTrees asks if an int is anywhere in an IntBinTree
- How would we argue correctness for IBTSearch? Inductively...
  - Recursive case: For non-empty T, if i is in T, it’s either at the root, in left, or in right—(by defn, that’s all there is in a tree). So...

```
Algorithm: IBTSearch(i, T)
//Input: int I, IntBinTree T
//Output: True exactly when I is in T, False otherwise
if T == empty
  return False
else
  if T.val == i
    return True
  else
    return IBTSearch(i,T.left) or IBTSearch(i,T.right)
```

Recall: Sum Algorithm Design

- Design an algorithm to solve this problem
  
  ```
  Input: non-negative integer n
  Output: the sum of the integers from 0 to n
  ```
  
- We’ve done an iterative (i.e., using loops, not recursion) algorithm
Sum, More

• Design an algorithm to solve this problem

<table>
<thead>
<tr>
<th>Input: non-negative integer ( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: the sum of the integers from 0 to ( n )</td>
</tr>
</tbody>
</table>

• We’ve done an \textit{iterative} (i.e., using loops, not recursion) algorithm

• What would a recursive algorithm for it be?

How would we break this problem down recursively, into one or more sub-problems of the same form, on smaller inputs?

Sum, More

• Design an algorithm to solve this problem

<table>
<thead>
<tr>
<th>Input: non-negative integer ( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: the sum of the integers from 0 to ( n )</td>
</tr>
</tbody>
</table>

• We’ve done an \textit{iterative} (i.e., using loops, not recursion) algorithm

• What would a recursive algorithm for it be?

• How would you argue correctness? How could you show that it always returns \( n(n+1)/2 \)?
  
  – Testing could show it for a few values of \( n \), but how could we explain that it meets that specification for all values of \( n \)?