CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

• HW1 due Wednesday, Feb. 12
• Any questions on the reading from Appendices, covering summations and sets?
• Thank you to those who emailed me (from the assignment of last week!):
  – I’ve replied to all emails I’ve received (by 11am today)—if you sent one but haven’t heard from me, please let me know!
  – If you haven’t emailed me, be sure to do so—please look at the first day’s lecture notes to find the assignment

• What is a problem, in a useful, computational sense?
  – Informal definition: In a relevant sense, a problem is an input/output relationship

• What’s an algorithm? Informal definition, from CLRS:
  
  An algorithm is a well-defined computational procedure that takes input and produces output.

• What does it mean to solve a problem?
  – Informal definition: In this computational sense, a solution to a problem is an algorithm…
  – We say an algorithm correctly solves a problem when it transforms every input to its related, correct output

A tiny bit about the course, the Remix:
Introduction to some Main Ideas

• Some important elements for any course on algorithms:
  – Algorithm design techniques and paradigms
  – Analyzing and explaining an algorithm’s correctness
  – Analyzing and explaining an algorithm’s complexity

• We’ll spend a lot of the semester on these important elements

<table>
<thead>
<tr>
<th>Design Paradigm</th>
<th>Analysis</th>
<th>Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Complexity (Efficiency)</td>
<td>Loop invariants</td>
</tr>
<tr>
<td>Iterative</td>
<td>Counting (Exact count of operations / space used)</td>
<td></td>
</tr>
<tr>
<td>Recursive</td>
<td>Solving recurrences</td>
<td>Induction</td>
</tr>
</tbody>
</table>
To Start, To Sum It Up…

• There are a couple of especially important summation formulas for our purposes, both of which are (in some form) in the appendix reading in CLRS

\[ \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \]

\[ \sum_{i=0}^{\infty} \frac{1}{2^i} = \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 2 \]

• Do you know how to derive these formulas?

Sum Algorithm Design

• Programming from specifications—designing algorithms or code based only on problem specifications—is an essential skill

• Design an algorithm to solve this problem

Input: non-negative integer \( n \)
Output: the sum of the integers from 0 to \( n \)

• What would an iterative (i.e., using loops, not recursion) algorithm be?

Questions to keep in mind for later:
• What would a recursive algorithm be?
• What is the time complexity of each algorithm?
• How could we explain the correctness of each algorithm?
Sum Algorithm Design

- Programming from specifications—designing algorithms or code based only on problem specifications—is an essential skill
- Design an algorithm to solve this problem

**Input:** non-negative integer \( n \)
**Output:** the sum of the integers from 0 to \( n \)

- What would an iterative (i.e., using loops, not recursion) algorithm be?
- 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 \( n \)?

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…
Binary Trees: A Review

• Remember binary trees from CS231?
  – Application: Binary search trees (BST’s)

Note: This isn’t asking about the definition of a BST, but about the more general data structure of a binary tree
One Possible Definition of Binary Tree

- Often, an implementation of a binary tree is based on two classes: a Node class, and a Tree class (as well as a type T of data to store)
- Node<T> class has fields:
  - item: T – the data stored at the node; a value of type T
  - left: Node – the left sub-tree, represented by its root node
  - right: Node – the right sub-tree, represented by its root node
  - … and perhaps others …
- Tree<T> class has fields:
  - root: Node<T> – the root node, which represents the tree
  - … and perhaps others, such as …
  - size: int – the number of nodes in the tree

If we wanted a data structure just to be a binary tree of integers, for example, would we need all of this structure?

Definition of an IntBinTree Data Structure

- Throughout CS375, we will sometimes refer to an IntBinTree data structure, representing a binary tree of integers
- In English, we’d say an IntBinTree is:
  - Either empty,
  - Or
    - an int, called val
    - and two subtrees, called left and right, that are also IntBinTrees
- Programmers might be used to seeing it more like this

```c
Definition IntBinTree:
int val # the int value
intBinTree left # the left subtree
intBinTree right # the right subtree
```

Is this definition equivalent to the English one above?
The fact that a tree could be empty is implicit here, as in many implementations.

NOTE: This definition may show up on HW, too!
IntBinTrees: An Exercise

- Design an algorithm to return the number of *levels* in an IntBinTree

  As IntBinTrees, we would say:
  - tree (a) has 3 levels, and
  - tree (b) has 5 levels

- What design paradigm will we use for this algorithm?
  - Will it be iterative or recursive?

Consider principle 2, “Let your definitions tell you what to do.” How does our definition of an IntBinTree tell us what to do here?

Definition IntBinTree:

- int val # int value
- IntBinTree left # left subtree
- IntBinTree right # right subtree

Recall, it could be empty, too

IntBinTrees: An Exercise

- Design an algorithm to return the number of *levels* in an IntBinTree

  As IntBinTrees, we would say:
  - tree (a) has 3 levels, and
  - tree (b) has 5 levels

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

In fact, because an IntBinTree is defined recursively in terms of two IntBinTrees, it might make sense for an algo over IntBinTree to have two recursive calls! (That's an example of Principle 2—letting definitions tell us what to do.)

Definition IntBinTree:

- int val # int value
- IntBinTree left # left subtree
- IntBinTree right # right subtree

Recall, it could be empty, too
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?

```
Definition IntBinTree:
int val  # int value
intBinTree left  # left subtree
intBinTree right  # right subtree
```

It can be helpful to write down a problem specification with input and output types....
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

- Our recursive algorithm to compute the number of levels:
  
  ```
  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*

- 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
```