CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: It’s complicated…

Business, Feb. 15

• I’ll have at least one “Course Business” slide like this in each module’s lecture notes
  – I may have more than one! Each one will have the date on it corresponding to the class meeting in which it was presented
  – I’ll make course announcements, post updates, etc. in these slides

• Candidates for visiting positions in class today
• HW1 Extended: Now due Feb. 22
  – Submission instructions coming soon
• TA hours TBA soon
Business, Feb. 15, pt. 2

• 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 10pm yesterday—if you sent one but haven’t heard from me, please let me know!
  – If you’ve emailed me since then, I’ll reply to you soon!
  – If you haven’t emailed me, be sure to do so—please look at the first day’s lecture notes to find the assignment
• My Office Hours are slightly altered this week
  – See my email, sent last night
Business, Feb. 17

• Reminder: I’ll have at least one “Course Business” slide like this in each module’s lecture notes
  – I may have more than one! Each one will have the date on it corresponding to the class meeting in which it was presented
  – I’ll make course announcements, post updates, etc. in these slides
  – Posted notes for a course module may be modified after their initial posting, to add these “Business” slides
• Candidates for visiting positions in class today
• TA hours (all time local to Maine / Colby):
  – Thursdays, 9-10pm (except during Break)
  – Sundays, 8-9:30pm (except before / after Breaks)
  – I will email you this information soon, with Zoom links

Business, Feb. 17, pt. 2

• Thank you to those who emailed me (from the assignment of last week)!
  – I’ve replied to all emails I’ve received—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

• My Office Hours are slightly altered this week
  – See my email, sent Sunday
Business, Feb. 17, pt. 3

- HW1 Extended: Now due Feb. 22

- Problem Set / Small Assignment submission instructions for CS375 this semester:
  - Place a copy in your CS375_<userid>_SubmittedWork folder in your Google Drive filespace
    - Will be created and shared with you soon!

- Graded assignments will be placed there for you to pick up
  - I expect that I’ll also email you with Classwide Comments when graded assignments are returned (or are to be returned very soon)
Business, Feb. 22

- PS1 due already
  - Problems submitting to CS375_<userid>SubmittedWork folder?
  - Recall that graded assignments will be placed there for you to pick up
  - I'll also email you with Classwide Comments when graded assignments are returned (or are to be returned very soon)
- PS2 out today, due March 1 (see assignment sheet)
- Also, Small Assignment 1 (about logarithms) is assigned today
  - Due by beginning of class Feb. 24
  - See CS375 Lecture Notes page for the SA1 assignment sheet
    - (Look under “Slides, Readings, Etc.” for this course module 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


- Just as a problem can have many algorithms that solve it (e.g., sorting, searching problems)…
  (There are multiple sort and search algos. Which ones do you know?)
- … An algorithm can have many possible implementations in code
  - For example, every programming language would lead to a different implementation
- For CS375, we’ll consider problems and algos more than code
  - You already know enough to implement algos in at least one language!

Determining the best solutions for a problem is often better at the algorithm level than the code level—all implementations of the same algorithm will have the same time / space complexity!
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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Complexity (Efficiency)</td>
</tr>
<tr>
<td></td>
<td>Correctness</td>
</tr>
<tr>
<td>Iterative</td>
<td>Counting</td>
</tr>
<tr>
<td></td>
<td>(Exact count of operations / space used)</td>
</tr>
<tr>
<td>Recursive</td>
<td>Solving recurrences</td>
</tr>
<tr>
<td></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

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

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

This may be the most important—I think of this as “Our Favorite Summation Formula”!

This is a form of an infinite summation

• 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 Nights. fun.

What can you say, informally, about its time complexity?

- 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?

This figure from CLRS illustrates that binary trees could, in principle, contain the same data in different tree structures (parts a and b).

Part c shows **external nodes**, the **null** fields in a typical implementation of a binary tree. They can be thought of as empty sub-trees, in context. **External nodes are hidden in parts a and b.**
Binary Trees: A Review

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

  ![Binary Tree Example]

Just as a reminder of where you’ve seen binary trees before: Both of these are examples of a binary search tree—a specific kind of binary tree.

In our work over the several slides, we’ll talk not about BST’s, but about binary trees in general.

• What’s a good definition 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

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

Is this a good definition? Consider Principle 1:
Keep your foundations simple.

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 \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

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

Is this a good definition? Consider Principle 1: Keep your foundations simple....

Is this definition equivalent to the English one above? 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 \textit{levels} in an IntBinTree.
- What design paradigm will we use for this algorithm?
  - Will it be iterative or recursive?

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

\begin{verbatim}
Definition IntBinTree: Empty, or...
  int val # int value
  intBinTree left # left subtree
  intBinTree right # right subtree
\end{verbatim}

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

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

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:

  ```java
  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?
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?
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

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.
Another IntBinTrees Exercise: 

**Search**

- The *search problem* on IntBinTrees asks if an int is anywhere in an IntBinTree.
- How would we argue correctness for \texttt{IBTSearch}? Inductively...
  - Base case: If \texttt{T} is empty...

\begin{algorithm}
\textbf{Algorithm: IBTSearch(i, T)}
\begin{itemize}
  \item //Input: int \texttt{i}, IntBinTree \texttt{T}
  \item //Output: True exactly when \texttt{i} is in \texttt{T}, False otherwise
  \item if \texttt{T} == empty
    \item \hspace{1em} return False
  \item else
    \item if \texttt{T.val} == \texttt{i}
      \item \hspace{1em} return True
    \item else
      \item return IBTSearch(i, T.left) or IBTSearch(i, T.right)
\end{itemize}
\end{algorithm}

Some people might view this as having two base cases.

The definition of \texttt{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

  - 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

- 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?

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 \( n \)?
Brute Force Algorithms and Other Mathematical Functions

• How many ways are there to arrange all of the elements in a n-element list? (How about when n==0?)

This can be important for understanding the complexity of a brute force algorithm, which checks all possibilities (e.g., all arrangements of elements in a list, all subsets) to solve a problem.

• Let’s keep building our recursion muscles! How would we write recursive algorithms to compute these functions?

Brute Force Algorithms and Other Mathematical Functions

• How many ways are there to arrange all of the elements in a n-element list? (How about when n==0?)
  – n!, or n factorial, defined on natural numbers as the product
    \[ n! = 1 \times 2 \times \ldots \times (n-2) \times (n-1) \times n \]
    – 0! = 1
  • How many subsets does a set of n elements have? (How about when n==0?)

This can be important for understanding the complexity of a brute force algorithm, which checks all possibilities (e.g., all arrangements of elements in a list, all subsets) to solve a problem.

• Let’s keep building our recursion muscles! How would we write recursive algorithms to compute these functions?
Brute Force Algorithms and Other Mathematical Functions

- How many ways are there to arrange all of the elements in a n-element list? (How about when n==0?)
  - n!, or n factorial, defined on natural numbers as the product
    \[ n! = 1 \times 2 \times \ldots \times (n-2) \times (n-1) \times n \]
  - 0! = 1

- How many subsets does a set of n elements have? (How about when n==0?)
  - 2^n, an exponential function; 2^0 = 1

This can be important for understanding the complexity of a brute force algorithm, which checks all possibilities (e.g., all arrangements of elements in a list, all subsets) to solve a problem

- Let’s keep building our recursion muscles! How would we write recursive algorithms to compute these functions? (Writing the 2^n exponential function is part of our HW)