CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

- Grading update:
  - PS2 in progress
- Expect PS3 out in next couple of days
  - Due no sooner than 1 week after it's assigned
- Project 2
  - First part due already
  - Other parts due Nov. 3
  - Please note some restrictions on my schedule:
    - I will not be available for dress rehearsals Tuesday or later
    - I expect to be traveling on Nov. 3 and probably won't be on email or able to answer questions after noon on that day, so please plan accordingly

Business:
Some notes on grading of projects

- Grading on projects is in part, as might be expected, similar to grading of other papers in other, non-CS courses
- Overall, for both presentations and write-ups, more credit will be given to submissions that demonstrate:
  - Greater scope of work completed
  - Greater depth of insight in the work completed
  - Greater command of relevant concepts
  - Greater clarity, completeness, and effective communication of the work

(This is related to the idea that not all improvements to your exhaustive search algo are of equal value, as noted on the project assignment)

This is meant to be intuitive—what you'd expect to get credit for—but please let me know if there are any questions about grading criteria!

Business:
Some notes on grading of projects

- Grading on projects is in part, as might be expected, similar to grading of other papers in other, non-CS courses
- As expected, credit will be given for work *your team* has done
  - ***As always, be sure to cite / credit every source of assistance, including your Prof., TAs, textbook, and online sources***
  - Be specific about contributions from other sources, so your audience can be certain about what work your team has done (i.e., as opposed to work taken from other sources)
  - As expected, credit is given for work done by your team—presenting others' work or ideas earns credit for the presentation, but not for creating the work / ideas of others
- As always, please feel free to ask me grading-related questions as you do your work on projects!

It's best to direct project-oriented questions to me, rather than to TAs—TAs do not always have the context that will best help you complete the project as intended / expected

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 our *IntBinTree* Data Structure

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

- 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

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

```
Definition IntBinTree: Empty, or…
    int val # the int value; not empty
    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 often implicit in many specifications

Definition of our *IntBinTree* Data Structure

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

- 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
- To be unambiguous (*and consistent with functional programming*) about how we work with IBTs, these will be the primitive functions defined on IBTs:
  - val(*T*): returns the *val* element of an IBT *T*
  - left(*T*): returns the *left* subtree of an IBT *T*
  - right(*T*): returns the *right* subtree of an IBT *T*

We'll call them *IBTs*, for short

For CS375, these are the only ways to access *val, left, right*. So, something like *T.val* is not permitted.

Important note: *val(T), left(T),* and *right(T)* are *functions* that return values; they are not fields of an object. Because of this, we cannot assign values to them—e.g., val(T) = 3 is not permitted.

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(left(T)), Levels(right(T))} + 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(left(T)), Levels(right(T))} + 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?

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

- 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

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 val(T) == i
     return True
   else
     return IBTSearch(i,left(T)) or IBTSearch(i,right(T))
```

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 *IBTSearch*? Inductively…
  - Base case: If *T* is empty…

```
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 val(T) == i
     return True
   else
     return IBTSearch(i,left(T)) or IBTSearch(i,right(T))
```

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…

```
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 val(T) == i
   return True
  else
   return IBTSearch(i,left(T)) or IBTSearch(i,right(T))
```

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…
  - Does the algo terminate? I.e., does input get closer to a base case with each recursive call? (Hint: Yes. It's always a subtree, …)

```
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 val(T) == i
   return True
  else
   return IBTSearch(i,left(T)) or IBTSearch(i,right(T))
```

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.

Break It Down Again

- In general, different ways of breaking down a problem into subproblems can lead to different algorithms

e.g., Mergesort vs. …
any other sort, basically

- Different data structures, by their definitions, suggest different natural ways to break problems into subproblems
  - How would a binary tree suggest breaking a problem into subproblems?
  - How would a node-based linked list suggest breaking a problem into subproblems?
  - How about for an array?

This isn't to say that, for any given data structure, some approach is *always* applied!

This is just looking for common approaches, and what makes them natural in context.

Break It Down Again

- In general, different ways of breaking down a problem into subproblems can lead to different algorithms

  e.g., Mergesort vs. … any other sort, basically

- Different data structures, by their definitions, suggest different natural ways to break problems into subproblems
  - How would a binary tree suggest breaking a problem into subproblems? [subproblems on sub-trees—kinda one half at a time]
  - How would a node-based linked list suggest breaking a problem into subproblems? [subproblems on lists one element shorter]
  - How about for an array? [subproblems involve changing indices and iterating over indexed ranges—index access is central to arrays!]

In all of these cases, the foundations—the definitions of the underlying structure—suggest that approach to breaking into subproblems

Break It Down Again

- In general, different ways of breaking down a problem into subproblems can lead to different algorithms

  e.g., Mergesort vs. … any other sort, basically

- Different data structures, by their definitions, suggest different natural ways to break problems into subproblems
  - How would a binary tree suggest breaking a problem into subproblems? [subproblems on sub-trees—kinda one half at a time]
  - How would a node-based linked list suggest breaking a problem into subproblems? [subproblems on lists one element shorter]

In a very broad sense, these represent *the two major ways* of thinking about recursion: Size of subproblems either divided from [tree] or subtracted from [list] original input size

In all of these cases, the foundations—the definitions of the underlying structure—suggest that approach to breaking into subproblems

List Algorithms

- We've seen how the definition of a binary tree can guide the design of algorithms on binary trees…
- Many common algorithms are written on lists
  - How does the definition of a list guide the designs of those algorithms?

What are some difference between arrays and lists, in this context?

List Algorithms

- We’ve seen how the definition of a binary tree can guide the design of algorithms on binary trees…
- Many common algorithms are written on lists
  - How does the definition of a list guide the designs of those algorithms?

What are some difference between arrays and lists, in this context?

- Among the significant differences between an array and a list:
  - Arrays are stored as contiguous blocks of memory; lists, when not simply extensions of arrays, are node-based (*linked lists*)
  - Arrays have direct, constant-time *indexed* access to any element; lists require traversing a list to reach an element, which is not constant-time
  - Because lists are node-based, it can be constant-time to access the sub-list of all but the first element *as a distinct object*

List Algorithms

- We’ve seen how the definition of a binary tree can guide the design of algorithms on binary trees…
- Many common algorithms are written on lists
  - How does the definition of a list guide the designs of those algorithms?

What are some difference between arrays and lists, in this context?

- For example, consider the *search* problem on lists

Input: item *i* and list L

Output: True if *i* is an element of L,
False otherwise

- There are multiple ways to approach designing an algorithm for this… how might you design one?
  - What can you say about the complexity of your algorithm?

List Algorithms and Recursion

- Lists, as opposed to arrays, can have node-based defintitions
- As part of that, a List type is commonly defined recursively!
- How would you write a recursive algorithm to solve the search problem on lists?
  - One possibility is shown here:
  - How would we argue its correctness?
  - (Do you believe that it works correctly?)
  - What can we say about its complexity?

Input: item *i* and list L

Output: True if *i* is an element of L, False otherwise

```
Algorithm: recListSearch(i, L)
// see specification immediately above
  if L = [ ]
    return False
  else
    if i is L[0]
      return True
    else
      return recListSearch(i,L[1:])
```

This uses Python-like list slicing syntax to refer to “all but the first element of L”

List Algorithms and Recursion

- Lists, as opposed to arrays, can have node-based defintitions
- As part of that, a List type is commonly defined recursively!
- How would you write a recursive algorithm to solve the search problem on lists?
  - One possibility is shown here:
  - How would we argue its correctness?
  - (Do you believe that it works correctly?)
  - What can we say about its complexity?
  - List slicing can’t be assumed to be constant time!

Input: item *i* and list L

Output: True if *i* is an element of L, False otherwise

```
Algorithm: recListSearch(i, L)
// see specification immediately above
  if L = [ ]
    return False
  else
    if i is L[0]
      return True
    else
      return recListSearch(i,L[1:])
```

This uses Python-like list slicing syntax to refer to “all but the first element of L”

List Algorithms:

Remove (first occurrence of an element)

- Consider the problem of removing the first occurrence of an element from a sequence, specified here for a list

Input: item *i* and list L = [x_0, …, x_n]

Output: If *i* = x_k and k is the smallest value for which *i* = x_k, return [x_0, …, x_{k-1}, x_{k+1}, … x_n]
Otherwise—i.e., when there is no k such that *i* = x_k—return L

- How would you design an algorithm to solve this problem…
  - Iteratively, on array-based lists?
  - Recursively, on node-based lists?
  - How would the complexity of this be different on a list (i.e., a linked list) than on an array?

Definition of our *LList* Data Structure

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

- Throughout CS375, we will sometimes refer to an *LList* data structure, representing a list of elements
- In English, we'd say an LList is:
  - Either empty,
  - Or
    - an element, called *first*
    - and an LList, called *rest,* representing all the elements after *first*

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

Is this consistent with your understanding of list structures—that is, *linked list* structures (which are typically node-based in implementation)?

Definition of our *LList* Data Structure

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

- In English, we'd say an LList is:
  - Either the empty list,
  - Or
    - an element, called *first*
    - and an LList, called *rest,* representing all the elements after *first*
- To be unambiguous about how we work with LLists, these will be the primitive functions defined on Llists:
  - first(L): returns value of the *first* element of an LList L
  - rest(L): returns value of the *rest* sublist of an LList L
  - cons(v,L): a ***constructor*** function that takes an item v and an LList L and returns a new LList L' such that…?
    - (What do you think it might be?)

What do you think the complexities of these functions are?