

Algorithm Design and Analysis, Colby College CS 375

Prof. Eric Aaron

Example Recursive Algorithms and Inductive Arguments for CS375

- These are intended as examples of recursive algorithms and inductive arguments of correctness for CS375, for you to use as a reference. Please note that there is no single right way to express an inductive correctness argument, so if yours isn't like these, that doesn't necessarily mean yours is any less correct!
- These example algorithms and inductive arguments use our *IBT* and *LList* datatypes—in particular, for algorithms that find the maximum value in an *IBT* with all positive integers, and in an *LList* of all positive integers. (We went through these algorithms in our Nov. 2 class meeting; they are not in our lecture notes, but they are presented here.) Please see our lecture notes for details about the definitions of the *IBT* and *LList* datatypes, and the functions that can be applied to them; those details are not included here.

1 General Notes about Inductive Arguments for CS375

An inductive argument should show that on any input, a recursive algorithm meets its specifications. As mentioned in the lecture notes of Oct. 31, there are three components to an inductive correctness argument of a recursive algorithm in CS375:

1. Explain how the algorithm's **base case returns correct output**
2. Explain how the **recursive cases return correct output, under the assumption that all recursive calls return correct output**
3. Explain how we know the algorithm **terminates**—show that arguments in recursive calls get smaller and closer to the base case

In the above components, showing that output is correct requires referring to the output specifications for the problem being solved, as demonstrated below.

In addition, as with correctness arguments using loop invariants, inductive arguments should refer to specific lines of pseudocode if possible—this can make an explanation substantially clearer and easier to follow.

2 Example: Max value in an LList

Here's pseudocode for an algorithm to find the maximum value in an *LList* that contains only positive integers:

```
# Input: LList L containing only positive integers
# Output: Largest value occurring in L, or 0 if L is empty
```

```
LLMAX(L)
```

1. **if** *L* is empty # base case: *L* is empty
2. **return** 0
3. **else** # recursive case: *L* is non-empty
4. **return** *max*(*first*(*L*), LLMAX(*rest*(*L*)))

2.1 An inductive correctness argument for *LLMax*

We can explain the correctness of the *LLMax* algorithm inductively, following the three parts mentioned in Section 1 above.

Note that by the definition of LList, every LList is either empty or a combination of a *first* and a *rest*. This inductive argument covers both of those cases, thus showing the algorithm correct on all possible inputs.

Base case The base case for the definition of LLists, and for the algorithm *LLMax*, is when LList L is empty. The specifications for the problem say that for an empty list, the algorithm should return 0, and on lines 1–2, this algorithm returns 0 when the input LList is empty, meeting those specifications.

Recursive case For the recursive case, the specifications say that we need to return the largest value in non-empty LList L . There are only two possibilities for the largest value in L : it's either the first element, or it's the largest value in L that isn't the first element, whichever of those two is bigger; no element could be the largest in L unless it was one of those two possibilities. Assuming that recursive call *LLMax*(*rest*(L)) correctly returns the largest value in *rest*(L) on line 4, then the calculation *max*(*first*(L), *LLMax*(*rest*(L))) computes the largest value in L , so the value returned on line 4 meets specifications.

Termination The base case is a list with no elements. The recursive call *LLMax*(*rest*(L)) is on an input of size one less than input L —*rest*(L) is always one element shorter than L , by the definition of the LList datatype—so each recursive call is on smaller than the input, getting closer to the base case, and the algorithm will eventually terminate on its input.

3 Example: Max value in an IBT

Here's pseudocode for an algorithm to find the maximum value in an *IBT* that contains only positive integers:

```
# Input: IBT T containing only positive integers
# Output: Largest value occurring in T, or 0 if T is empty
```

```
IBTMAX( $T$ )
```

1. **if** T is empty # base case: T is empty
2. **return** 0
3. **else** # recursive case: T is non-empty
4. **return** *max*(*val*(T), IBTMax(*left*(T)), IBTMax(*right*(T)))

3.1 An inductive correctness argument for *IBTMax*

We can explain the correctness of the *IBTMax* algorithm inductively, following the three parts mentioned in Section 1 above.

Note that by the definition of IBT, every IBT is either empty or a combination of a *val*, a *first*, and a *rest*. This inductive argument covers both of those cases, thus showing the algorithm correct on all possible inputs.

Base case The base case for the definition of IBTs, and for the algorithm IBTMax, is when IBT T is empty. The specifications for the problem say that for an empty tree, the algorithm should return 0, and on lines 1–2, this algorithm returns 0 when the input IBT is empty, meeting those specifications.

Recursive case For the recursive case, the specifications say that we need to return the largest value in non-empty IBT T . There are three possibilities for the largest value in T : it's either the root of the tree, or it's the largest value in the left subtree, or it's the largest value in the right subtree—the largest value in T is the largest of those three values. No element could be the largest in L unless it was one of those three possibilities. If a value is in the left subtree but isn't the largest value in the left subtree, for instance, it couldn't possibly be the largest value in the entire tree.

Assuming that recursive calls $\text{IBTMax}(\text{left}(T))$ and $\text{IBTMax}(\text{right}(T))$ correctly return the largest value in $\text{left}(T)$ and $\text{right}(T)$ (respectively) on line 4, then the calculation $\text{max}(\text{val}(T), \text{IBTMax}(\text{left}(T)), \text{IBTMax}(\text{right}(T)))$ computes the largest of those three values, which must be the largest value in T . So, the value returned on line 4 meets specifications.

Termination The base case is a tree with no elements. The recursive call $\text{IBTMax}(\text{left}(T))$ is on an input of size *at least* one less than input T —even if the right subtree is empty, $\text{left}(T)$ doesn't include the root, so it is always at least one element smaller than T ; by the same reasoning, the recursive call $\text{IBTMax}(\text{right}(T))$ is on an input of size *at least* one less than input T . So, every recursive call is on a tree that's smaller than the input, getting closer to the base case of an empty tree, and the algorithm will eventually terminate on its input.