Reading #2

Proving the Correctness of a Simple Recursive Algorithm

How Do We Specify a Computational Problem?

Definition: A ***specification of requirements for a computational problem*** (something that we want to solve using a computer program) includes the following:

- **Precondition:** A condition (property) that is either `true` or `false`, and that is satisfied by any well formed instance (i.e., set of inputs) for this problem.
- **Postcondition:** A condition that be satisfied if this problem is solved.

The **postcondition** might include relationships between inputs as well as outputs — as well as initial and final values of global data that are accessed and modified.

Examples

Consider the problem of computing the largest element of an integer array A, with positive length n, and whose elements are $A[0], A[1], \dots, A[n-1]$. This problem can be specified as follows.

Maximal Element of Integer Array

Precondition: An integer array A, with positive length n, is given as input.

Postcondition: The largest element in the set

$$\{A[0], A[1], \dots, A[n-1]\}$$

is returned as output.

A related problem is to compute the largest element in a *part* of an integer array:

Maximal Element in Part of an Integer Array

Precondition: An integer array A, with positive length n, and integers `low` and `high` such that $0 \leq \texttt{low} \leq \texttt{high} \leq n-1$, are given as input.

Postcondition: The largest element in the set

$$\{A[\texttt{low}], A[\texttt{low}+1], \dots, A[\texttt{high}]\}$$

is returned as output.

How Do We Specify an Algorithm?

According to the Wikipedia page about them, an ***algorithm*** is “an effective method expressed as a finite list of well-defined instructions for calculating a function.” If you replace the last few words, “calculating a function,” with the words “solving a computational problem” instead, then this definition will serve for these notes.
Algorithms can be presented in a variety of ways, including using (clear and precise) English, using pseudocode, or using executable code.

Example

An algorithm that solves the “Maximal Element in Part of an Integer Array” problem is as follows. It is described using pseudocode.

```
integer maxInRange (integer[] A, integer low, integer high) {
1.  if (low == high) {
2.    return A[low]
    } else {
3.    integer mid := floor((low + high)/2)
4.    return max(maxInRange(A, low, mid), maxInRange(A, mid+1, high))
    }
}
```

Correctness

Definition: An algorithm, for a given computational problem, is ***correct*** if the following property is satisfied: Whenever the problem’s precondition is satisfied and the algorithm is executed, then the execution of the algorithm eventually ends, and the problem’s postcondition is satisfied when this happens.

No changes to the system state, that are not documented in the precondition and postcondition, should be caused by the execution of this algorithm if the precondition holds when the execution begins. Thus the only undocumented variables whose values might change should be local variables — which did not exist before execution began and do not exist again after execution ends.
Note: This does not say (or promise) anything about what happens if the problem's precondition is *not* satisfied and the algorithm is executed.

A Proof of the Correctness of an Algorithm

The following is a ***proof*** that the `maxInRange` algorithm is correct — that is, it correctly solves the "Maximal Element in Part of an Integer Array" problem.

- It is assumed (or used as an ***axiom***) here that the `floor` function always receives a real number `x` as input and returns, as output, the largest integer that is less than or equal to `x`.
- It is also assumed that the `max` function always receives a pair of integers, `a` and `b` as input, and returns as output whichever of `a` or `b` is larger.

Theorem: Suppose that `A` is an integer array with positive length n, `low` and `high` are integers such that $0 \le \texttt{low} \le \texttt{high} \le n-1$, and that the algorithm `maxInRange` is executed, given `A`, `low` and `high` as inputs.
Then this execution of the algorithm ends, and the largest value in the set

$$\{\texttt{A}[\texttt{low}], \texttt{A}[\texttt{low}+1], \dots, \texttt{A}[\texttt{high}]\}$$

is returned as output when this happens.

Proof: This will be proved by induction on $\texttt{high} - \texttt{low}$. The strong form of mathematical induction will be used, and the (single) case that $\texttt{high} - \texttt{low} = 0$ will be considered in the basis.

Basis: Suppose that $\texttt{high} - \texttt{low} = 0$, Then

$$0 \le \texttt{low} = \texttt{high} \le n-1.$$

In this case, when the execution of the algorithm begins, the test at line 1 is successful, so that step 2 is then executed. The execution of the algorithm then ends and, since

$$\{\texttt{A}[\texttt{low}], \texttt{A}[\texttt{low}+1], \dots, \texttt{A}[\texttt{high}]\} = \{\texttt{A}[\texttt{low}]\}$$

in this case, the largest element of this set is then returned as output, as required.

Inductive Step: Let k be an integer such that $k \geq 0$. It is necessary and sufficient to use the following *inductive hypothesis* to prove the following *inductive claim*.

> Inductive Hypothesis: Suppose that `A` is an integer array with positive length n, `low` and `high` are integers such that $0 \leq \texttt{low} \leq \texttt{high} \leq n-1$ and, furthermore, that $0 \leq \texttt{high} - \texttt{low} \leq k$. If the algorithm `maxInRange` is executed given `A`, `low` and `high` as inputs then this execution of the algorithm ends, and the largest value in the set
> $$\{\texttt{A}[\texttt{low}], \texttt{A}[\texttt{low}+1], \dots, \texttt{A}[\texttt{high}]\}$$
> is returned as output when this happens.
>
> Inductive Claim: Suppose that `A` is an integer array with positive length n, `low` and `high` are integers such that $0 \leq \texttt{low} \leq \texttt{high} \leq n-1$ and, furthermore, that $0 \leq \texttt{high} - \texttt{low} = k+1$. If the algorithm `maxInRange` is executed given `A`, `low` and `high` as inputs then this execution of the algorithm ends, and the largest value in the set
> $$\{\texttt{A}[\texttt{low}], \texttt{A}[\texttt{low}+1], \dots, \texttt{A}[\texttt{high}]\}$$
> is returned as output when this happens.

With that noted, let us suppose that

- `A` is an integer array with positive length n as input,
- `low` and `high` are integers such that $0 \leq \texttt{low} \leq \texttt{high} \leq n-1$ and, furthermore, $\texttt{high} - \texttt{low} = k+1$.

Suppose that the algorithm `maxInRange` is executed given A, `low` and `high` as input.

- In this case, $\texttt{low} > \texttt{high}$ so that the test at line 1 fails and execution of the algorithm continues with line 3.
- Since $\texttt{high} = \texttt{low} + k + 1$,
$$\texttt{mid} = \left\lfloor \frac{2\texttt{low} + k + 1}{2} \right\rfloor = \texttt{low} + \left\lfloor \frac{k+1}{2} \right\rfloor . \tag{1}$$
The integer k must either be even or odd.
- If k is even then $k = 2h$ for some integer $h \geq 0$.
- In this case $\texttt{mid} = \texttt{low} + h$, so that $0 \leq \texttt{low} \leq \texttt{mid} \leq \texttt{high} \leq n-1$ and, furthermore, $0 \leq \texttt{mid} - \texttt{low} = h \leq k$.

- It follows by the inductive hypothesis that the execution of the algorithm `maxInRange` with inputs A, `low` and `mid` at line 4 eventually halts, with the largest element of the set

  $$\{A[\texttt{low}], A[\texttt{low} = 1], \dots, A[\texttt{mid}]\}$$

  returned as output.

- Since $k = 2h$, $\texttt{mid} + 1 = \texttt{low} + h + 1$ and $\texttt{high} = \texttt{low} + 2h + 1$, $0 \leq \texttt{high} - (\texttt{mid} + 1) = h \leq k$ as well. It follows by the inductive hypothesis that the execution of the algorithm `maxInRange` with inputs A, $\texttt{mid} + 1$ and `high` at line 4 also eventually halts, with the largest element of the set

  $$\{A[\texttt{mid} + 1], A[\texttt{mid} + 2], \dots, A[\texttt{high}]\}$$

  returned as output.

- One can see, by inspection of the code at line 4, that this execution of the algorithm also ends and that the largest element in the set

  $$\{A[\texttt{low}], A[\texttt{low} + 1], \dots, A[\texttt{high}]\}$$

  is returned as output in this case — as required.

- On the other hand, if k is odd then $k = 2h + 1$ for some integer $h \geq 0$ In this case, $\texttt{mid} = h + 1$.

- ***Exercise:*** Modify the argument for the case that k is even, as needed, to prove that this execution of the algorithm would also halt, with the largest element of the set

  $$\{A[\texttt{low}], A[\texttt{low} + 1], \dots, A[\texttt{high}]\}$$

  returned as output in this case too.

- Since both cases have been considered, it now follows that an execution of the algorithm with inputs A, `low` and `high` such that $0 \leq \texttt{low} \leq \texttt{high} \leq n - 1$ and $\texttt{high} - \texttt{low}k + 1$ always halts, with the largest element in the set

  $$A[\texttt{low}], A[\texttt{low} + 1], \dots, A[\texttt{high}]\}$$

  being returned as output — as needed to establish the inductive claim and complete the inductive step.

- The result now follows by induction on $\texttt{high} - \texttt{low}$. $\square$

Things To Notice About This Proof

- This makes use of a recognized proof technique — ***mathematical induction*** — specifically, the strong form of mathematical induction, using induction on `high` − `low`.
- This is written in the style used in the previous set of notes — and avoids the mistakes and shortcuts discussed there.
- This ***traces the execution of the algorithm*** on various kinds of inputs in order to establish the basis and complete the inductive step.

Checking for Undocumented Side-Effects

One can see by inspection of the code of the recursive algorithm `maxInRange` that

- The only inputs accessed are the inputs `A`, `low` and `high` that are mentioned in the "Maximal Element in Part of an Integer Array" problem's precondition, and no global data is accessed.
- The algorithm does not modify any input or global data and the only output returned is the value mentioned in the above problem's postcondition.

Thus it has no undocumented side-effects and really *can* be considered to be "correct."

Correctness of Algorithms: Questions and Answers

At this point some people have questions like the following, or conclusions about this material, based on assumed answers:

1. ***Question:*** Why, on earth, should anyone care about the ***correctness*** of an algorithm, or computer program, anyway?

   Answer: There are certainly situations where this kind of analysis of an algorithm is infeasible or unrealistic. For example, any problem in the domain of ***artificial intelligence*** cannot be handled in this way — the problems are too big.

   On the other hand, problems that are ***safety-critical*** should (at least arguably) be handled in this way. Widely used software like ***library software*** — the kind of software found in the "Java Collections Framework" or the "Standard Template Library" should (at least arguably) be treated in this way too — because so many people make use of it in ways that cannot be anticipated or controlled.

2. ***Question:*** Why can we not rely on ***testing and debugging*** instead of all this mathematical stuff?

   Answer: These approaches are actually ***complementary*** rather than ***competitive:*** It is a good idea to know about and be able to apply *both* approaches:

- In this course we will generally be considering algorithms given as ***pseudocode*** rather than ***executable code***. *Dynamic testing* allows working code to be run and checked — making it more effective in the identification of coding errors as well as the effects of assumptions that tend to get made in proofs of correctness — like "integer arithmetic is exact".
- On the other hand, ***testing can almost never be exhaustive or complete:*** There will virtually always be missed situations or cases, not considered in test suites, that will only be noticed years or even decades after software — again, especially software like library software — is in widespread use. Understanding a proof of the ***correctness*** of an algorithm can be helpful for more effective and extensive ***test design*** and more effective ***debugging***.

3. ***Question:*** Why is it important that there are no undocumented side-effects when the algorithm is executed?

   Answer: Once again, ***library software*** should be considered.

   - This will be used in larger systems in unpredictable and uncontrollable ways. Undocumented side-effects will, on occasion, cause some of these larger systems to fail.

     If this happens, it might be difficult or impossible to figure out why.

4. ***Question:*** Oh, come on, now. It is ***obvious*** that the algorithm shown here is correct. Why is such a trivial example being presented?

   Answer: The decision to do this can be explained on both academic and workplace-related grounds.

   - ***Academic Justification:*** This involves the application of material from an indirect prerequisite in discrete mathematics — MATH 271 or 273. Some students passed this prerequisite but did not excel in it. Others have also forgot its contents. The application of this material to problems in computer science might not have been discussed very much (if at all).

     Starting with something simple and small — that does not provide additional sources of confusion — makes sense, for anyone trying to help someone to try to learn to do this stuff, under these circumstances.
   - ***Workplace Justification:*** The ***software library*** scenario is relevant again: You are contributing to something that others will use in a way that you can neither predict nor control. You cannot anticipate the way your work will be used. Nor will you necessarily be able to change it later. You are adding to ***bedrock*** that others will depend on, when you are not around to explain or repair.

     Doing this kind of thing gradually, and from the bottom up, makes sense, in support of what you are to provide to other software professionals.

Documenting Correctness

Helpful ***inline documentation*** that you should consider including at the beginning of any program that implements a nontrivial algorithm includes

- the ***precondition*** and ***postcondition*** for the problem that can be solved using this algorithm;
- a reference to a readable ***proof of the correctness of this algorithm***, if one has been published;
- ***bounds on the time and space*** required for the execution of this algorithm on a given input, if these have also been proved.

Additional inline documentation, described below, also helps a reader of the code to see *why* it is correct.

Bound Functions for Recursive Algorithms

Definition: A ***bound function for a recursive algorithm*** is a mathematical function that is defined on the inputs and global data that are accessed and modified when the recursive algorithm is executed — specifically, on those inputs and global data satisfying the precondition for the problem being solved — and that also satisfies the following properties — assuming, as usual, that the problem's precondition is satisfied when the algorithm is executed.

1. This is an *integer-valued* function.
2. Whenever the algorithm is applied recursively, the value of the function has been *decreased* by *at least* one.
3. If the function's value is less than or equal to zero when the algorithm is applied then the algorithm does not call itself recursively during this execution.

Example: Consider the recursive algorithm `maxInRange`, and the function

$$f(\mathtt{A}, \mathtt{low}, \mathtt{high}) = \mathtt{high} - \mathtt{low}.$$

1. Since `low` and `high` are both integer inputs, this is certainly an *integer-valued* function.
2. As noted in the proof of the correctness of this algorithm, the value of this function is decreased by at least one every time the algorithm is replied recursively — see the "inductive step" in the above proof of correctness of the algorithm for details.

3. As noted in the proof of correctness of this algorithm, $\texttt{low} \le \texttt{high}$ whenever this algorithm is recursively applied. It follows that is only possible that

   $$f(\texttt{A}, \texttt{low}, \texttt{high}) = \texttt{high} - \texttt{low} \le 0$$

   under these conditions if $\texttt{high} - \texttt{low} = 0$, that is, $\texttt{low} = \texttt{high}$.

   In this case, the test at line 1 passes, execution continues with line 2 and execution ends immediately after that, without the algorithm having called itself recursively.

What To Notice About This Argument:

- A ***bound function*** was identified — and established, just by checking that all of the properties listed in the ***definition*** of a "bound function for a recursive algorithm" were satisfied.
- This only required an examination of the pseudocode of the algorithm. Establishing a bound function for a recursive algorithm will *often* (but not always) be as easy as this.

Why are Bound Functions Useful?

- It is often possible to prove properties of recursive algorithms, using mathematical induction — specifically, using induction on the *initial value of the bound function*.

  Indeed, this is how the ***correctness*** of recursive algorithms will generally be proved.

Assertions

Definition: An ***assertion*** is...

- a ***boolean condition*** (or "predicate")
- involving an algorithm's ***inputs, local variables, outputs,*** and (possibly) ***global data***...
- that is assumed to be satisfied at ***a specific point in a computation,*** namely, either immediately before the execution of the algorithm begins, or immediately before or after a specific instruction in the algorithm has been executed — assuming, again, that the problem's precondition was satisfied when execution began.

Why are Assertions Useful?

- Assertions can be used to document significant parts of the proof of correctness of the algorithm. If they are reasonably complete, then a reader may see the proof of correctness just by reading the bound function and assertions!
- `Java`, `C++`, and some other languages include statements with names like `assert` that **allow assertions to be actively *checked***, when programs are being run during testing. This can make it *much* easier to find errors in software than otherwise — because this makes the code self-checking.

Assertions for the Example Algorithm

- Assertions only need to be satisfied during executions when the precondition is initially satisfied — but they must be satisfied if this is the case. Thus an assertion including the following

  1. A is an input integer array with positive length n.
  2. low and high are integer inputs such that $0 \leq \texttt{low} \leq \texttt{high} \leq \texttt{n} - 1$.

  can be listed immediately *before* line 1 of the maxInRange algorithm.
- The assertion that can be used immediately *after* this line should reflect the fact that this point is only reached if $\texttt{low} = \texttt{high}$ as well:

  1. A is an input integer array with positive length n.
  2. low and high are integer inputs such that $0 \leq \texttt{low} = \texttt{high} \leq \texttt{n} - 1$.

- The assertion that can be used immediately after step 2 should reflect the fact that the output has now be returned in this case:

  1. A is an input integer array with positive length n.
  2. low and high are integer inputs such that $0 \leq \texttt{low} = \texttt{high} \leq \texttt{n} - 1$.
  3. The value
     $$\texttt{A[low]} = \max(\texttt{A[low]}, \texttt{A[low+1]}, \ldots, \texttt{A[high]})$$
     has been returned as output.

- Assertions for later points in the (pseudo)code should also reflect what is known when each point is reached.
- A version of the algorithm that includes all such assertions is available as a separate document.

Discovering and Checking Correctness

Ideally...

- The designer of an algorithm should supply a proof of its correctness.
- Other people wishing to use this algorithm should also have access to this proof.

With that noted, the following can be helpful if you wish to do any of the following:

- Discover a proof of correctness of an algorithm, if one has not been supplied.
- Understand a proof of correctness of an algorithm, if this is not clear.
- Discover whether an *implementation* of an algorithm might be incorrect, by checking it by hand.

Traces of Execution

A ***trace of execution*** of an algorithm, on an input, is a listing of the statements (in the algorithm) that are performed, when the algorithm is executed on this input, along with a description of the effects of these statements.

Complication: If the algorithm is recursive then one execution of the algorithm will often include one or more other executions of the same algorithm on different inputs.

One way to deal with this when providing a trace of execution — that will be used in the example to follow — is

- to number (or otherwise label) the traces of all the executions that are included, and
- to give a separate trace of execution for each, providing cross-references between traces as needed.

Tracing an Execution of the Example Algorithm on a Particular Input
In the following example `A` is an integer array with length 5, in all executions such that

- $\texttt{A[0]} = 8$,
- $\texttt{A[1]} = 10$,
- $\texttt{A[2]} = 4$,
- $\texttt{A[3]} = 25$, and
- $\texttt{A[4]} = 3$.

In the initial execution $\texttt{low} = 0$ and $\texttt{high} = 4$, so that the largest integer in the entire array, 25, should eventually be returned as output.

Execution #1: The integers $\texttt{low} = 0$ and $\texttt{high} = 4$ are received as the additional inputs.

1. The test at step 1 is executed. Since $\texttt{low} \neq \texttt{high}$ this test fails, so that the execution of the algorithm continues with step 3.
2. At step 3, `mid` is set to have value $\lfloor (0+4)/2 \rfloor = 2$.
3. Step 4 is executed:
   (a) The algorithm is recursively executed with the same input integer array `A` and with $\texttt{low} = 0$ and $\texttt{high} = 2$; see the trace of "Execution #2" for details. The value $10 = \max(\texttt{A}[0], \texttt{A}[1], \texttt{A}[2])$ is returned.
   (b) The algorithm is recursively executed with the same input integer array `A` and with $\texttt{low} = 3$ and $\texttt{high} = 4$; see the trace of "Execution #7" for details. The value $25 = \max(\texttt{A}[3], \texttt{A}[4])$ is returned.
   (c) The value $25 = \max(10, 25) = \max(\texttt{A}[0], \texttt{A}[1], \texttt{A}[2], \texttt{A}[3], \texttt{A}[4])$ is returned as output.

Execution #2: The integers $\texttt{low} = 0$ and $\texttt{high} = 2$ are received as the additional inputs.

1. The test at step 1 is executed. Since $\texttt{low} \neq \texttt{high}$ this test fails, so that the execution of the algorithm continues with step 3.
2. At step 3, `mid` is set to have value $\lfloor (0+2)/2 \rfloor = 1$.
3. Step 4 is executed:
   (a) The algorithm is recursively executed with the same input array `A` and with $\texttt{low} = 0$ and $\texttt{high} = 1$; see the trace of "Execution #3" for details. The value $10 = \max(\texttt{A}[0], \texttt{A}[1])$ is returned as output.
   (b) The algorithm is recursively executed with the same input integer array `A` and with $\texttt{low} = 2$ and $\texttt{high} = 2$; see the trace of "Execution #6" for details. The value $\texttt{A}[2] = 4$ is returned as output.
   (c) The value $10 = \max(10, 4) = \max(\texttt{A}[0], \texttt{A}[1], \texttt{A}[2])$ is returned as output.

Execution #3: The integers $\texttt{low} = 0$ and $\texttt{high} = 1$ are received as the additional inputs.

1. The test at step 1 is executed. Since $\texttt{low} \neq \texttt{high}$ this test fails, so that the execution of the algorithm continues with step 3.
2. At step 3, `mid` is set to have value $\lfloor (0+1)/2 \rfloor = 0$.
3. Step 4 is executed:
   (a) The algorithm is recursively executed with the same input integer array `A` and with $\texttt{low} = 0$ and $\texttt{high} = 0$; see the trace of "Execution #4" for details. The value $\texttt{A}[0] = 8$ is returned.
   (b) The algorithm is recursively executed with the same input integer array `A` and with $\texttt{low} = 1$ and $\texttt{high} = 1$; see the trace of "Execution #5" for details. The value $\texttt{A}[1] = 10$ is returned.
   (c) The value $10 = \max(8, 10) = \max(\texttt{A}[0], \texttt{A}[1])$ is returned as output.

Execution #4: The integers $\texttt{low} = 0$ and $\texttt{high} = 0$ are received as the additional inputs.

1. The test at step 1 is executed. Since $\texttt{low} = \texttt{high}$ this test succeeds, so that the execution of the algorithm continues with step 2.
2. The value $\texttt{A[low]} = \texttt{A}[0] = 8$ is returned as output.

Execution #5: The integers $\texttt{low} = 1$ and $\texttt{high} = 1$ are received as the additional inputs.

1. The test at step 1 is executed. Since $\texttt{low} = \texttt{high}$ this test succeeds, so that the execution of the algorithm continues with step 2.
2. The value $\texttt{A[low]} = \texttt{A}[1] = 10$ is returned as output.

Execution #6: The integers $\texttt{low} = 2$ and $\texttt{high} = 2$ are received as the additional inputs.

1. The test at step 1 is executed. Since $\texttt{low} = \texttt{high}$ this test succeeds, so that the execution of the algorithm continues with step 2.
2. The value $\texttt{A[low]} = \texttt{A}[2] = 4$ is returned as output.

Execution #7: The integers $\texttt{low} = 3$ and $\texttt{high} = 4$ are received as the additional inputs.

1. The test at step 1 is executed. Since $\texttt{low} \neq \texttt{high}$ this test fails, so that the execution of the algorithm continues with step 3.
2. At step 3, `mid` is set to have value $\lfloor (3+4) \rfloor = 3$.
3. Step 4 is executed:
   (a) The algorithm is recursively executed with the same input integer array `A` and with $\texttt{low} = 3$ and $\texttt{high} = 3$; see the trace of "Execution #8" for details. The value $\texttt{A}[3] = 25$ is returned.
   (b) The algorithm is recursively executed with the same input integer array `A` and with $\texttt{low} = 4$ and $\texttt{high} = 4$; see the trace of "Execution #9" for details. The value $\texttt{A}[4] = 3$ is returned.
   (c) The value $25 = \max(25, 3) = \max(\texttt{A}[3], \texttt{A}[4])$ is returned as output.

Execution #8: The integers $\texttt{low} = 3$ and $\texttt{high} = 3$ are received as the additional inputs.

1. The test at step 1 is executed. Since $\texttt{low} = \texttt{high}$ this test succeeds, so that the execution of the algorithm continues with step 2.
2. The value $\texttt{A[low]} = \texttt{A}[3] = 25$ is returned as output.

Execution #9: The integers $\texttt{low} = 4$ and $\texttt{high} = 4$ are received as the additional inputs.

1. The test at step 1 is executed. Since $\texttt{low} = \texttt{high}$ this test succeeds, so that the execution of the algorithm continues with step 2.
2. The value $\texttt{A[low]} = \texttt{A}[4] = 3$ is returned as output.

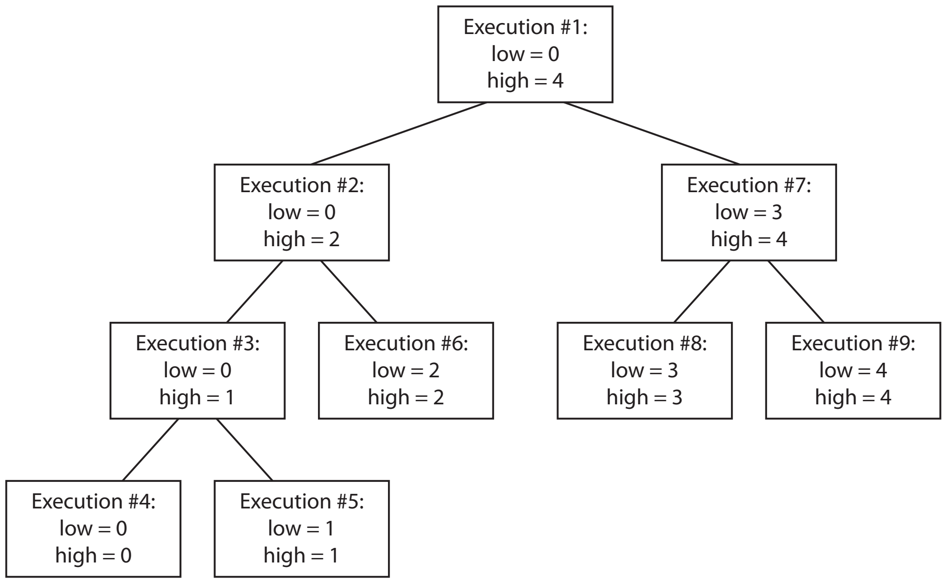
Recursion Trees

A ***recursion tree*** is a picture (actually, a "tree") that corresponds to an execution of a recursive algorithm on a fixed input and shows the relationship between all the different executions that get made along the way:

- The "root" of this tree corresponds to the original execution of the algorithm.
- If the algorithm calls itself recursively on an execution then the node for this execution has a "child" corresponding to each of the recursive calls that it makes.

- If the algorithm does not call itself recursively then the node for this execution does not have any children at all — that is, it is a "leaf" in the tree.

The recursion tree for the execution of the algorithm `maxInRange` on the inputs shown in the above trace of execution is shown below. The input array `A` is not shown in order to simplify the picture.


Execution #1: low = 0 high = 4
Execution #2: low = 0 high = 2
Execution #7: low = 3 high = 4
Execution #3: low = 0 high = 1
Execution #6: low = 2 high = 2
Execution #8: low = 3 high = 3
Execution #9: low = 4 high = 4
Execution #4: low = 0 high = 0
Execution #5: low = 1 high = 1

Recursion Trees: Why are They Useful?

- Recursion trees help to make sense of all the different "traces of execution" that are needed when one wants to consider the execution of a recursive algorithm on a particular input.
- These can also be very helpful when you are trying to prove interesting things about the **running times** and **storage requirements** for recursive algorithms.