CSE 3101 Design and Analysis of Algorithms
Meta Steps for Unit 1

Jeff Edmonds

This contains the **most important** concepts in this unit. You will not be able to pass this course without knowing and understanding these. The steps provided **must** be followed on all assignments and tests in this course. Do **not** believe that because you know the material, you can answer the questions in your own way. Though this material is necessary, it does not contain everything that you need. You must read the book, go to class, review the slides, and ask lots of question.

Chapters 1,2,4,5,&6: Loop Invariants and Iterative Algorithms

If you are designing, describing, and proving correct an algorithm using loop invariants, the following is the list the things must consider. This step order is a nice one for presenting them because it is the order in which they are often logically developed. You might prefer to present them in the order that the computation proceeds. It is a judgment call. Each step should be contained in its own paragraph with the heading given.

Specification: Define the Problem (pre and post conditions): Likely defined by the problem.

*** Define Loop Invariant:** Define Loop Invariant. It needs to be a picture of what is true every time the algorithm is at the top of the loop. Also specify which of the following it is:

More of the Input: A prefix of the input has been read so far. Pretending that this prefix is the entire input, I have a complete solution. I also have any additional information stored so that I can solve the problem for the entire input without needing to go back and reread the parts of the input that I have already read.

More of the Output: The first so many of the objects in the required output have been correctly produced.

Narrowing The Search Space: A narrowed search space has been specified and if the thing being searched for is anywhere, then then it is in this narrowed space.

Shrinking Instance: An instance has been produced that is smaller than the actual instance and these two instances require the same output. Hence, it is sufficient for me to forget about the actual instance and simply give the output for the new smaller one.

Fairy God Mother: I have arrived from Mars, some amount of progress has been made already and my Fairy God Mother has set the state of the computation to be just what I want so that I feel as comfortable as I could be given the amount of progress that has been made.

*** Step:** Define the step $code_{loop}$ that is executed each iteration and is guaranteed to *maintain the loop invariant* while *making progress*. Though this step, when actually coding, may require a nested loop or other fancy code, here it suffices to give a quick description of what actions take place. (Though you can't say that miracles occurs.)

- One common way decide what actions should be taken in an iteration is to first change the data structure is some way that makes progress. Perhaps you increase i by one, consider the next item, or cut the value of x in half.
- You know that the loop invariant had been true when you were at that top of the loop. However, the change you made to make progress messed up the loop invariant being true.
- Now you have to make a change to the data structure that brings it back to the loop invariant, without undoing the progress that you just made. Breath and move back onto the path.
- On the other hand, sometimes something else moves messes up the loop invariant. Maybe the monster runs around the river after you. Your initial and main focus then might be to maintain the loop invariant. After you can give some thought to making progress.

*** Maintaining the Loop Invariant:**
$\langle loop{-}invariant' \rangle$ & *not* $\langle exit{-}cond \rangle$ & $code_{loop} \Rightarrow \langle loop{-}invariant'' \rangle$:

- Imagine that you have just flown in from Mars and are at the top of the loop. All you know about the state of the data structure is that the loop invariant is true and the exit condition is not. (Ok you have not actually defined the exit condition, but you know that you are not done.)
- Let x' be the current values of the variables and x'' after going around the loop again.
- State what you know about x' from the fact that $\langle loop{-}invariant' \rangle$ holds for these values.
- The step $code_{loop}$ constructs the values x'' from the values x'. State what this tells you about x''.
- State how it then follows that $\langle loop{-}invariant'' \rangle$ holds for the values x''.

*** Exit:** Define the Exit Condition.

Not Post Condition: It would be natural to exit when you are done and hence it is natural to make the exit condition be the same as the post condition. But this just passes the problem of will the post condition ever be met onto whether your loop will ever exit. Do not do this.

Sufficient Progress: Instead, define a measure of progress and have your exit condition be that sufficient progress has been made. If it is clear that progress is made each iteration, then it will be obvious that the loop will eventually exit. In this case, you can skip the Termination step.

Panic: Another good exit condition is the reason that that you can no longer maintain the loop invariant. Note that defining such an exit condition makes it easier to do the previous step of proving that if the exit condition is not true then the step taken maintains the loop invariant. You may have to revisit this previous step.

*** Establishing the Loop Invariant:** $\langle pre{-}cond \rangle$ & $code_{pre\text{–}loop} \Rightarrow \langle loop{-}invariant \rangle$:
Don't work hard. Do the minimum work to make the loop invariant true.

- State what you know about the input instance because of $\langle pre{-}cond \rangle$.
- State the effect of the code before the loop.
- State how it then follows that $\langle loop{-}invariant \rangle$ has been established.

*** Obtaining the Postcondition:** $\langle loop{-}invariant \rangle$ & $\langle exit{-}cond \rangle$ & $code_{post\text{–}loop} \Rightarrow \langle post{-}cond \rangle$:

- State what you know because of $\langle loop{-}invariant \rangle$.
- State what you know because of $\langle exit{-}cond \rangle$.
- State what you know because of $code_{post\text{–}loop}$.
- State how it then follows that $\langle post{-}cond \rangle$.

Termination: As stated above, if it is obvious that the exit condition will be met you do not need to do these step.

- Define measure of progress.
- Initially finite progress required.
- Prove that each iteration you make progress.
- Proof that with sufficient progress, the exit condition will be met.

*** Running Time:** Give an upper bound on the number of times that this algorithm iterates. Also estimate the total running time.

==========

System Invariants (Quite straight forward but we dont cover.)

Like *loop invariants*, another extremely useful tool are *system invariants*. They are similar, but have the following differences.

Goal: Loop invariants are used to prove the correctness of an iterative algorithm computing a function from the given input to the required output. In contrast, system invariants are used to maintain the integrity of a system though out time.

A Function: Given an input meeting a pre condition, an iterative algorithm computing a function repeated takes the *step* within the loop in order to compute an output meeting the post condition.

A System: A system/class/ADT/object has an internal data structure that stores its current state and has a set of operations that can be executed by the user/client (or users) of the system.

Relationship to Loop Invariants:

"Loop" Through Time: The system does not have an internal loop like an iterative algorithm that repeatedly executes its step until the exit condition is met. Instead, the system's "loop" is time. Though out time, the external user of the system specifies a sequence of the systems operations. The goal of the system is to perform each of these operations when it arises in time.

More of the Input Loop Invariant: A system invariant can be viewed as a more of the input loop invariant, if the input $I = \langle R_1, R_2, \ldots, R_n \rangle$ is the sequence of requests that will be made by the client of the system. Each R_i specifies which operation needs to be executed and on what inputs. Then the system invariant says that a prefix of these requests have been handled and the state of the system meets the system invariants.

Invariants: System Invariants might be *public* or *private*.

Public Invariants: Public system invariants are specified in the description of the abstract data type or system so that the user can better understand how it works.

Bank: The system invariant of a bank account is that the amount is not negative.

Stack: The system invariant of a stack is that it contains an ordered list of objects which cannot be changed except at the front end via the explicit push and pop operations.

Drug Allocation: A system that keeps track of which drugs has have been allocated by various doctors for various conditions to various patients would have the invariant that a patient is not simultaneously prescribed a set of drugs that interact poorly with each other.

Private Invariants: A particular implementation of an ADT or system likely will have its own invariants that are not revealed to its users. They are used to ensure that the algorithm for each operation works correctly and efficiently.

Balanced Binary Search Tree: A balanced binary search tree is an implementation of the ADT *Set*, that allows the client to store a set of object. These objects can be searched for, can be added, and can be deleted. The private invariants will ensure that the objects are stored at the nodes of a binary tree such that for each node all the values in its left subtree are at most that at this node and all in the right are at least that. To ensure $\mathcal{O}(\log n)$ search times, an additional invariant might impose some tree balancing requirement.

Drug Allocation: A private invariant of an implementation of a drug system might be that each patient's record contains a list of the patient's doctors, his ailments and the drugs he is and has been on.

Maintaining the System Invariants: No matter what sequence of operation occurs on a ADT/system, their invariants must be maintained.
$\langle system{-}invariant' \rangle \;\&\; code_{op} \Rightarrow \langle system{-}invariant'' \rangle$:

- Imagine that you have just flown in from Mars. All you know about the state of the system is that its invariants are true. Then some operation gets executed.
- Let x' be the current values of the variables and x'' after executing the operation.
- State what you know about x' from the fact that $\langle system{-}invariant' \rangle$ holds for these values.

- The operation $code_{op}$ constructs the values x'' from the values x'. State what this tells you about x''.
- State how it then follows that $\langle system{-}invariant'' \rangle$ holds for the values x''.

Operation Specification: Each operation allowed on the ADT/system will have an input, an output, and a change to the state of the system. It will be specified using pre and post conditions.

As a Function: The search & delete operation on at binary search tree can be thought of as executing the following function whose input is $\langle key, tree \rangle$ and whose output is $\langle object, tree' \rangle$. Given key meets the pre-condition of a valid key and $tree$ is the current state of the ADT meeting all public and private system invariants, then $object$ as specified by the post condition will be the object in the data structure $tree$ with key key (assuming it exists) and $tree'$ is the same as $tree$ except this object has been deleted. As always the operation must guarantee that $tree'$ still meets all the system requirements.

As a System Operation: The only different when this operation is thought of as a system operation instead of as a function is that $tree$ and $tree'$ are no longer inputs and outputs, but are global variables specifying the internal state of the system. These variable are hidden from the client so cannot be accessed or changed except through these operations.

Proof of Correctness:

$\langle system{-}invariant' \rangle$ & $\langle pre{-}cond_{op} \rangle$ & $code_{op} \Rightarrow \langle post{-}cond_{op} \rangle$ & $\langle system{-}invariant'' \rangle$

Obtaining the Postcondition: Unlike iterative algorithm, a system may continue forever or might get terminated by one of its operations. But as seen above, after each operation is executed, the prover must prove that the system invariants ensure that the post conditions of the operation are full filled correctly.

Initializing the System: We have seen how an iterative algorithm must establish the loop invariants from the pre conditions on its inputs before the first iteration of the loop. Similarly, each system/ADT will have operation that create a new instance of the system/ADT. Such operations will have inputs that must meet given preconditions They must establish the initial state of the system so that all the system requirements are true.

$\langle pre{-}cond_{InitializingOp} \rangle$ & $code_{InitializingOp} \Rightarrow \langle system{-}invariant \rangle$