Debugging

I see many of you wasting large blocks of time on debugging. As you already know, the requirements of this class are such that you cannot afford to waste any time. Below are my suggestions for debugging. I have been programming for 35 years! While I don't pretend to have all the answers, these techniques work! Give them a try.

1) If you are asking someone for help getting started, show them your attempts at a solution. You may be totally wrong, but at least the tutor can see quickly where you are at.
2) After you finish entering your code, get a printed copy and **read it**. You are not looking for syntax errors, but for logic errors. You want to see the logical functioning at a higher level. This simple practice can save you an hour of debugging per program. You can find several mistakes in a few minutes.
3) Program defensively. Write an assert routine such as

```
public class BadData extends Exception
{
   public BadData(String msg)
   { super(msg);
   }
   }
   public void assert(boolean b,String msg)
   {
   try{
   if (!b)
   {
   throw new BadData(msg+"Assert Violated");
   }
   } catch (Exception e) {e.printStackTrace(); System.exit(1); }
}
```

   Using assert statements, place a variety of tests in your code. For example, verify input files were opened correctly, verify input data is correct, verify array bounds are within range. Simple assert statements take but a few minutes to insert, yet can save you hours. When I include asserts, I often think I am doing someone else a favor, but they end up helping me.

4) Learn to use a debugger.
5) However, do not try to do all debugging from the debugger. Tracing through the code one line at a time can waste HOURS of time as you focus on the low level details. You need to get a higher level perspective. Insert print statements at key points to tell you - what parameters were passed, what values were read, what is the content of the data structure at a given point in time. These print statements help you to get the global picture of what is happening so you can narrow down your range of search. You must break the problem apart so that you can verify certain components work properly. You wouldn't try to understand Shakespeare by examining each word separately. You would need to come up with your own meanings, circumstances,

character biographies, objectives and units of actions. You would need to look at the play in more advanced ways than an individual word analysis.

I put such debug statements in my code preceded by the conditional ``if (debug)" Then I set debug to true or false depending on what I want to see printed. I leave these debug statements in the final version. When you ask for help, these printed statements are much better than a verbal description of what you saw in the debugging window.

Many of you are too impatient to spend the time approaching the problem of debugging in a methodical manner. Some of you are ``too busy sawing to sharpen the saw." Take some time to insert strategic print statements. This practice will save you hours of debugging time.

6) You should have a way of printing **every** class of interest. Writing a tree manipulation routine without being able to see the tree at points of interest is horrible. That is exactly why I often make you code routines like ``print prettily" (a routine that prints trees like we draw them) and ``print array". These routines should be called whenever you have a question about what the data structure really contains. Don't debug in the dark. Sometimes people say to me, ``I know this works. I've done it on paper five times, but when I run it I don't get the right answer." Just get debug information so you verify that what you think is happening *is* happening.

   Lots of times when I suggest putting in extra debug statements, I meet with hostility, ``Well, I can see what the statement does. Printing out key values won't do any good. That won't solve the problem." The simple fact is this: if the code were doing what you *"know"* it is doing, it would be working. Since it isn't working, you need to discover the difference in what should be happening and what is happening. If you are getting integer division instead of real division or if you are doing assignment (a=1) instead of comparison (a==1) or if you have inadvertently put a semicolon after your while condition, you can read the code dozens of times without picking up on the error. If you insert ``almost ridiculous" debug statements, the errors will quickly become apparent.

7) Get up close and personal. There is almost a disease in debugging among students. They come to me saying, ``This isn't working correctly. See, the output isn't right. I've tried several things, but the output still isn't right. Tell me what is wrong." When I ask them, ``Is the data read in properly? What are the values of the variables at this point? What is stored in the array? Does this statement ever get executed?" the answer is always, ``I don't know." There is a fundamental law of debugging: **You cannot fix the code by merely observing the output. You must create information which tells you what is going on inside the code.**

   What you are doing is analogous to the following scenario: You are a mechanic. Your car won't start. You say to yourself. Gee the car won't start. I'll try giving it more gas. Nope, still doesn't start. I'll try putting in oil. Nope, still doesn't start. I'll try a new

starter. Nope, still doesn't start. I'll try a new battery. Nope, still doesn't start. I'll try cleaning the engine. Nope, still doesn't start. I'll try washing it. Nope, still doesn't start. This you continue for days systematically replacing everything that could possibly be wrong. This is called hacking. You can't afford to do it. You must discover more information about the problem. If you were a lay person and not a mechanic, replacing every conceivable piece may be a reasonable approach, but a mechanic must know how to test individual components to see if they are functioning properly. He must know which components are even suspect. The trial and error (clueless) approach is not efficient.

When you try to drag your instructor into this hacking loop, she may become quite agitated. You debug by gathering information about the problem, not by just staring at it until the problem is revealed by inspiration or hacking at it mindlessly.

8) Try running your program with a *small* data set. When I give you a file of 2000 records, you should not do your initial debugging with the whole file. It is too lengthy to trace or print the data structures.
9) Start early so that when you hit a bug you can't figure out, you have time to see a tutor and won't be tempted to make random changes.
10) **When you do ask someone for help, have a CURRENT listing of your program (which is written according to the style guidelines), input, and output with you. Also bring your disk.** This is SO important. It says you value the time of the tutor (as well as your own time). I *hate* to debug someone else's code solely from an executable. I must see the whole program to understand what you are doing. It is rare that I need to run the program to find the error, if you can clearly describe what is happening. I like a listing so I can circle problems and see the whole program in a glance. If you come to me without a listing, I will send you away to get one.
11) Do not try random changes. You should try simple changes to your code to see if something logical makes a difference, but trying a wide variety of completely different approaches wastes time. You need to find out what is wrong with the original version. Sometimes people throw away a good approach to try something awkward which works. Bad idea. Make the good approach work. See it through. Don't just abandon it the first time you hit a snag.
12) Don't spend hours and hours looking at the same output. I use the ten minute rule. I will spend ten minutes looking at the code or output. If I can't figure the problem out in that much time, I GET MORE DEBUG information. Debug makes the difference clear between what you think is happening and what is really happening. Sometimes students say, ``I have spent 10 straight hours on that bug." I have to ask, ``Why?" That almost never makes sense. Get help. Get more debug. Get some rest and come back.
13) You need to learn to debug on your own, without the help of others. Because of the availability of tutors and friends who are willing to look at your code, some of you avoid doing your own debugging. As assignments get more and more difficult, others will have a harder time helping you. Some students get suggestions and then run back for more suggestions if success is not immediate. Before going back for more help, try a variety of things.

Come for help having tried all the basic things. For example:

a) If you have tried everything, you may try to simplify the program or input data. Get rid of as much input as you can and still produce the same problem. Make a test copy of your program and then get rid of as many lines as possible and still produce the error. This makes it much simpler for you and others to look at. For example, if the input routine isn't working properly, make a copy of the program which contains *only* the input routine. Get it working before combining it with other parts of the program. Some people always code this way - getting each routine working properly before going on to the next one. If you have a great deal of trouble debugging, this technique is strongly recommended.
b) Narrow down the problem. Use print statements to verify that each module works correctly. For example, if your program consists of three major phases, make sure each one has produced the correct output. Then as you debug, successively add MORE and MORE debug statement in areas where you suspect the problem lies. Let your code talk to you.
c) If a module appears to be doing something totally impossible, use detailed print statements to verify exactly what is happening. If a value isn't being passed back correctly, print it before it is returned to make sure it is computed correctly. If an if statement isn't giving the correct results, print important values to figure out why. If a certain branch doesn't appear to be executed, use a print statement before and after that section of code to print the conditional value and verify the code was reached.
d) Don't jump to the conclusion that the compiler has a fundamental error, e.g. it can't compile the *if* statement. Although compiler errors do occur infrequently, the chances of you finding a compiler errors are extremely small.

   In fact, don't jump to any conclusion. Verify any hunch you have.

These steps are critical in debugging efficiently. Many times I have students come to my office, very frustrated with a problem. Often, they have neglected most of the above steps as they have ``found a better way'' (which doesn't work). Since your programming assignments over the course of the next few semesters get very complicated, you can NO LONGER afford to debug casually. You must become experts at it. It is really a fun part of the programming process if you do it correctly. It requires sophisticated logic - ``What error would give that kind of result?'' As the semester progresses, I will become increasing persistent about your debugging skills. Give it your best effort before coming for help.

That said, the single biggest help to debugging it to program so it works the first time. This seems like obvious advice, but much is being said.

- Pseudo-code and top down refinement! Pseudo-coding allows you to ignore the gory details, thus keeping intellectual control over the whole process. This is so important in design. As programs get more and more complicated, you will NOT be able to intellectually grasp all of the details at once. You MUST use abstraction to deal with the complexity. This is what pseudo-coding encourages.

You wouldn't paint a mural by first deciding the color and placement of the left hand of the townsperson. You would plan the overall composition of the painting and then add more and more detail, in layers.

- Program carefully. Worry about the special cases. The difference between an "A" programmer and a "C" programmer is the special cases. Getting it to work for the general case is a good start, but it isn't good enough. A program that works for 99% of the cases – doesn't work!
- Don't guess. Lots of times we are tempted to think, "It is something like this." Guessing has no future. Take the time to think it through.
- Read your code from beginning to end. Debugging concentrates on a few lines at a time, rarely looking at the whole unit. Assuring the whole thing makes sense is essential BEFORE you begin the debugging process.
- The code should "feel good". If the code seems awkward, like it will never work – 99.9% of the time it won't work.