Programming Assignments Guidelines

Data Structures

General Guidelines

There will be four assigned programming assignments (labs), each constituting 10% of the total grade. Programming assignments will consist of Java, Python modules, or C++, coded, and executed only by the student.

Late Policy

Programming assignments must be submitted to the course website by Tuesday midnight of the specified module. Assignments received after that are considered late and will be penalized 5 points for each day late. Assignments more than one week late cannot be accepted, except by prior arrangement with the instructor. Problems with your system do not constitute a legitimate excuse for lateness, so make plans to deal with the unavailability of your system. Programs must compile and produce legitimate output.

Style

Your code must incorporate a consistent, well-documented style. Required points of style include:

- Include clear, concise, and adequate inline comments
- Write substantive, descriptive blocks for each module
- Write a substantive, descriptive block for the entire program.
- Comments should describe the function performed, not restate the code in English.
- Comments should explain the purpose of a function or method, its inputs, and outputs.
- Comments should explain the algorithm being applied, a particular approach to a problem, or restrictions in using the code
- Use white space liberally and consistently.
- Write one driver that executes your entire code.
- Consistent, well-delineated use of both upper and lower case is encouraged.
- Using indentation to show nesting of control statements is encouraged.
- Set tabs to only 2-4 spaces and keep line lengths to about 80 columns to reduce wrap around.
- Keep code modular. One page is a good rough guide to module size.
- Do not use GOTOs or global variables
- Use include files (C++ users) so that a link step is not necessary.

Modularity

Organize your code in files so that each piece is cohesive, and functional. For example, if you write code that operates on a binary tree then collecting the methods that actually insert, update, and retrieve information from the tree into a single module might be effective. Keep the following in mind:

- If you cannot write a simple coherent unifying description to document the module, it may not be cohesive or functional.
- If it is longer than a page or two it may not be cohesive or functional
- If you have only one file for a very small program, it may be okay; but for a larger program one file is not likely to be cohesive or functional.
- Not everyone will divide things the same way and that is fine. What you do just needs to make sense.

Input

You must correctly handle any required input, turning in output to show that it does.

- **Generate your own test cases.** You will lose points for not providing adequate additional input.
- Generate input that checks extreme cases
- Generate input with errors that might reasonably occur due to typos or a novice user. Assume if it is possible to make a stupid mistake, someone will do so.
- When testing, approach your code as a total novice, then on a second pass, as though you are an experienced end-user.
- Show that your code does everything it is supposed to do
- Show all reasonable error cases are handled.
- **Use named files to handle I/O.** There will be a penalty for hardcoded file names.
- **Enter file names as command line prompts.**
- **Do NOT use GUI/console input.**
- It may make sense to parse the input as you read it. For example, if you are reading integers, you may want to check and discard input, using an appropriate error message, if it contains characters or is out of the specified range of values.
- If you want credit for something, then it is important to demonstrate it with an appropriate I/O set.

Output

Output is not part of the analysis. Output files must:

- Echo the input as well as contain answers to the required input.
- Be user friendly with additional labels, lines, and white space
- Have statistical information as needed.

Source Code

You are expected to write your own code except as provided in a specific assignment. Do NOT use code from the internet or other sources to be part of your assignment, except Lab 4. Use of standard libraries is restricted to standard I/O calls and standard math functions, etc. In other words, you cannot use the library stack code. You may not Vectors, ArrayLists, etc.

Analysis

The emphasis in the analysis document is on points such as correct application of concepts, technique, analysis of algorithmic efficiency, and lessons learned.

The analysis should discuss the following points

- Description of your data structures
- Justification of your data structure choices and implementation
- Discussion of the appropriateness to the application
- Description and justification of your design decisions
- Efficiency with respect to both time and space
- What you learned,
- What you might do differently next time
- Specific requirements in the lab handout
- Discussion of anything you did as an enhancement

- Provide supporting details of your discussion points
- Do not reiterate the requirements of the assignment
- Do not include pseudocode, code or output
- **Use Times New Roman Size 12 font, single spaced, with .75-inch margins on all sides.**
- Include your name inside the file.

A Complete Assignment

A complete lab assignment consists of the following, in a single zip file attachment, with your name as part of the file name: **SmithJLab1.zip.** Include your name inside each file. Do not paste content into the boxes in the assignment item.

1. **A written, analysis of the project, submitted as a PDF, with your name as part of the file name.**
2. **The source code.** Include all your source files, except standard libraries.
3. **The compiled code.** Include your compiled Java, Python, or C++ code
4. **Copies of all your input data sets** (as text files/s: .txt), required and student generated test input
5. **Copies of all your output** (as text files/s: .txt)
6. **A README file** which specifies the version of Java, Python or C++ and the IDE used, along with anything else we need to know to properly compile and execute your code.

Grading on Programming Assignments

The grade for each lab assignment is broken down as follows:

- **40% - Correctness** - In problem solution and results.
- **20% - Style/Proper Coding** - Following a reasonable, consistent style with STRONG documentation, with appropriate use of structures, modularity, error checking, etc.
- **10% - Input/Output** - Labeled, formatted, correct use of prompts, correctly handles specified inputs and outputs as well as additional cases provided by the student, is user friendly.
- **20% - Analysis**
- **10% - Enhancements** - Recognition of superior work on one of the required aspects of the assignments (everyone can do this) or work above and beyond the requirements. If you do something extra, please make sure it is reflected in the I/O set so you get proper credit. Discuss it in your analysis. We can't give you credit for something unless we are aware of it. If you add an extra feature to your code, make sure it is "in addition to", not "in place of" a required component of the problem.

This grading policy reflects the expectation that you can already write minimal, working code. If the unexpected comes up, please let me know. We are happy to discuss your grade with you anytime

Examples of Programming Assignment Grades:

94-100% - This is a very strong lab that correctly implements all the required elements, and includes corresponding example I/O cases, and uses a reasonable and consistent style. Each module has an introductory comment block giving an extensive, high level description of the module, detailed descriptive comments in the declarations sections and occasional detailed comments throughout the code. Error checking is strong and covers boundary conditions and additional cases beyond the minimum specified input. Output contains all the required elements, is user friendly, formatted in a visually pleasing manner, and contains useful descriptive statistics about the results. Extra features may have been added. The analysis addresses the design decisions incorporated into the code development and explicitly justifies the specific data structures used and the specific

implementation used. It considers both the theoretical and observed efficiency and explains any discrepancy that may exist, using Big Oh and theta notation appropriately. It

summarizes what the user learned in the lab, from language and structure specific experiences through big-picture concepts. It displays the writer's command of the topic in the assignment by identifying additional work that addresses known problems and/or continues to improve and expand the scope. It contains graphs or tables to enabling easier assimilation of the results. It is formatted as required.

86-94% - This is good solid lab. Correctness errors, if any, are minimal. Documentation, style and error checking are good. Error checking may have some small omissions. There is some extra input, but less than might reasonably be expected. Output looks fine and can be followed without any problem. There are probably no extra features, or just small, easy ones. The analysis is good, mostly formatted as required, but is less comprehensive. Some areas may be treated only superficially or omitted altogether.

80-86% - This is a good lab, with room for improvement. There may be small correctness errors, sometimes due to a misunderstanding of the requirements. Error checking, style, and documentation are standard. Output minimally covers what it needs to. There are few, if any, extra input cases. The analysis does not do a very good job of justifying the design decisions and structure choices and may not be formatted as required. Other aspects of the analysis, especially algorithmic efficiency, may be omitted.

70-80% - This is a weak lab. It may do everything correctly but minimally, with no extras thrown in, minimal documentation, and a very sketchy, superficial analysis, probably not formatted as required. Alternatively, this may be a well-done lab aimed for the categories above, with a moderately serious correctness error or a significant omission of a required component.

<70% - This is a lab with major errors in correctness and minimal efforts on style, and error checking. Documentation may be non- existent or extremely minimal. There are probably no additional I/O cases, no extra features. The analysis is probably under a page in length, not formatted as required, and addresses only one or two required topic areas.

Academic Integrity

IMPORTANT: You are expected to do your own work. Help from other sources must be acknowledged. It is okay to discuss the problem with others for perspective or to make sure you understand it correctly, but the code you write must be your own.

Downloading code from other sources for the programming assignments, while strongly discouraged, should absolutely be properly accredited. It is prohibited except for the Lab on Sorting.

Practical Points

- Reread this before turning in lab.
- Reread the handout of the assignment before turning in assignment.
- Check that your analysis is correctly formatted.
- Turn in a README file, noting any special requirements.
- **Compile and execute your code to verify that it works properly.**
- Exercise the features of your code with an extensive I/O set. Create test input containing errors/
- Fully document your code.
- Provide input files with the required input and your supplemental input.
- DO NOT turn in other's work.

- Provide output to all the required input cases and your supplemental cases.
- **Do NOT turn in an assignment that does not compile.**
- Remember that your code will be compiled and executed in a different environment than the one you are using.
- Consolidate everything in a single zip file, with your name as part of the file name: JSmithLab1.zip. I/O files should be text. The Analysis should be a PDF.
- If you need to resubmit, just go ahead and resubmit to the course website. It should be accepted, even after the due date.

Advice on Programming

- The single most important thing you can do to smooth the code development process and reduce the overall time required is to work out the entire problem on paper, including debugging. Resist the temptation to jump into the editor and type just to make yourself feel productive.
- Once you actually start to implement your code on the computer, adopt either a top-down or bottom-up approach, and stick with it. Implement a single module and fully debug it before you go onto the next module. Then when you add a new module, if there are problems you can more easily locate them (in other words, if you combine two working pieces of code, any problems occurring have to be in the interface between them).
- Remember that a debugger is a piece of software and is subject to bugs. If you cannot find a problem, then consider debugging the old-fashioned way with write statements.
- Back it up. Disk space is cheap, and you cannot back up your code too often. It is very easy to lose track of all the changes and iterations, so consider a version numbering system like V2.1 where changing the 1st number represents a big change and changing the second number represents a minor change.
- The final piece of advice: "If it ain't broke, don't fix it". Less experienced programmers sometimes start changing things willy-nilly in a frustrated effort to address code problems they cannot figure out. Get help, or at least save a copy, first.