Definitions

1. Explain the meaning of each of the following.
    a. a syntax error
        - aka “format errors”
        - Syntax error occurs when the programmer fails to obey one of the grammar rules of the language, eg. wrong case, placement of punctuation, etc. (Richard Baldwin 2007)
    b. a run-time error
        - Runtime error occurs when a syntactically-correct program is executed by the computer, but the computer is unable to complete the task. Common examples include dividing by zero, or trying to access an index that is out of bounds
    c. a logic error
        - Usually the most difficult to debug
        - The program may execute without and clear indication of an error, because the code is syntactically correct and there are no runtime errors.
        - Examples include implementing mathematical operations incorrectly, or branching incorrectly in conditionals.
    d. a test plan
        - aka “Testing Strategy”
        - Is a systematic approach to testing software
        - Refer to Q2 or notes for lectures 5-6 for more details.
    e. a unit test
        - Each “module” (class or function) is tested individually
        - Goal is to show that each module meets its specifications
        - Ignores interaction between modules
        - *First* state of software testing
            - Later stages consider groups of modules, and are simpler if we can be confident that each module works correctly by itself
    f. a regression test
        - If an error is found and corrected then testing of the affected modules and subsystems should be **repeated**, to be sure no new errors were introduced!
        - This is one reason why it is important to *document* tests - you may need to use them more than once!
    g. integration testing
        - Is performed after unit testing
        - Individual modules (that separately seem to be acceptable) are combined to form and test progressively larger subsystems
        - Multiple methods of an object might be tested in combination as part of this process

h. validation testing
    - aka Acceptance Testing
    - Involves users to ensure that specifications are met
i. system testing
    - Testing the integration of multiple software systems
j. static testing
    - aka Structured Walkthrough
    - Involves examination of source code without execution
    - Often first stage of *unit testing*
    - Is a “reality-check” on code before proceeding to more detailed or complicated testing
    - Two types:
        - Desk checking: read through code, look for errors
        - Hand executions: trace code execution on small inputs with known outputs by handi
k. dynamic testing
    - Tests the behaviour of a module or program during execution
    - Two types:
        - **Black Box Testing** (also called **Functional Testing**)
        - **White Box Testing** (also called **Structural Testing**)
    - *Both* black box and white box testing are useful for all phases of testing
l. a black box test
    - Includes tests designed using only the problem specification (not the code)
    - Tests both valid and invalid input
    - Tests typical cases and boundary conditions (special, rarely-occurring cases)
    - Is useful for finding
        - Incorrect or missing functions
        - Interface errors (involving functions)
        - Interface errors for data structions or external databases
        - initialization and termination errors
        - Is generally used in later testing states, but certainly can and should be used during unit testing too
    - Refer to notes for lectures 5-6 for full details.
m. a white box test
    - Includes tests designed using the internal workings of a module (including source code)
        - Goal is to test every line of code and every execution path
    - Tests typically try to ensure that:
        - Every statement in code is executed in one or more tests
        - Each “if” and “Else” branch of every conditional statement is tested

        - Each loop is iterated zero, one, several, and as many times as possible (if these situations are feasible)
        - Each exit condition causing a loop or function to terminate is executed
        - All exception handling is tested
    - Refer to notes for lectures 5-6 for full details.
n. functional testing
    - Another name for black box testing
o. structural testing
    - Another name for white box testing
p. defensive programming
    - A style of programming intended to ensure that software continues to function (or, at least, does not cause harm) in spite of unforeseeable use of the software
    - Includes the use of code that detects unexpected or invalid input data values - one way of "preparing for testing" as you write your code
q. debugging
    - A methodical process of finding and removing defects in a program
    - General process:
        - Recognize that a bug exists (eg. ideally, via testing)
        - Isolate the source of the bug
        - Identify the cause of the bug
        - Determine a fix for the bug
        - Apply the fix and test it

2. What should a test plan include?

    A Test Plan (or "Testing Strategy") includes
    - Deciding how software will be tested
    - Deciding when tests will occur
    - Deciding who will do the testing
    - Deciding what test data will be used - and what the expected output should be for each input

Testing Principles

1. When (during software development) should testing begin? Why?

    Testing should begin early on in software development. It is extremely unlikely that long and complex software is free of errors, and it is generally cheaper and easier to correct an error if it is detected early in software development.

2. Who should test your software? Why?

    It is frequently a good idea to have someone else test the software you have designed and implemented (if possible). We all have "blind spots." Frequently, other people can more easily see problems with our work that we don't notice ourselves. It is easy (and human) for us to be overly "protective" of our own work

- we'd like to think it is perfect! This is not helpful, considering that "the goal of testing is to prove that your software is correct."

3. Explain why tests should be thoroughly documented.

   Other people must be able to follow what was tested and why.

4. Is it possible to test, thoroughly and effectively, if a specification of the problem to be solved is not available? Why (or why not)?

5. Why are proofs of correctness and testing of software both useful?

   You can't "test in" quality or use testing to repair a method based on an incorrect algorithm - or debug code effectively unless you know what is is supposed to do. We need to test because proofs of correctness tend to be "sketched" instead of developed in detail, or skipped altogether, if correctness seems "obvious". Sometimes the proofs are faulty, and they tend to rely on idealistic and unrealistic assumptions (e.g. arithmetic is exact); testing provides a "reality check". A variety of errors can be introduced during the coding phase, even if you are starting with an algorithm that really is "correct".

6. It was claimed in class that the objective of testing is to prove that software is incorrect, and that a test is successful if it finds evidence of an error. Why is this the case?

   The purpose of testing is to identify errors in the program or algorithm. If a test is unable to identify any evidence of an error, then it is not doing its job. Poorly designed tests often do not cover cases most likely to result in an error (edge cases), and may work "most of the time" without discovering the errors we have missed.

7. If your experience as a programmer has been limited to the course you've taken here and at schools you've attended before, then it might seem like far more time and effort is now being devoted to testing than is necessary.

   Why is testing worthwhile anyway? In particular, when (or for what kind of software development) is this worthwhile?

   Testing is particularly useful when developing large and complex software. Testing can help identify errors quickly and effectively, in a systematic way. While it doesn't prove that your program is correct, it helps to reduce the possible errors. It can also save lots of time down the road, when you are trying to debug an error. This is especially true with logic errors, which are difficult to catch with just executing the program, but much easier with a well thought-out set of tests.