CS105 Midterm Exam Review Questions

These questions are meant to show what kinds of questions we might ask on the CS105 midterm. We won’t have time to ask detailed questions about every topic we’ve covered, so expect the actual test to have either fewer questions, easier questions, or both. Also, we generally take more care to provide clear wording and full descriptions on the actual test than on the sample, and the actual exam will have space for your answers.

1. Rewrite the program below to avoid using "and" or "or" by introducing "if's" inside other "if" statements. You do not need to make any changes to the precondition. Your new function should return the same thing this one does for any legal parameters.

```python
def strange( x, y, z):
    precondition(x != y and x != z and y != z)
    if (x < y and y < z):
        return "they are in order"
    elif (z < y and y < x):
        return "they are in reverse order"
    elif (y < z and y < x):
        return "y is the smallest"
    else:
        assert(y > z and y > x)
        return "y is the largest"
```

2. Consider the problem of determining the earliest letter alphabetically in a given string. For example, the earliest letter in "dave" is "a", while in "john" it is "h".

   a) Design an appropriate test suite for this problem, including as many different "kinds of questions" as you can.

   b) Give an appropriate pre- and post-condition for this problem.

   c) Give an recursive algorithm (looking approximately like a Python function) to solve the problem. (Note that two strings can be compared using the < operator, so that "a" < "b").

3. Consider the problem of converting a positive integer into a string of digits. For example, 157 should be turned into the string "157".

   a) Design an appropriate test suite for this problem

   b) Give an appropriate pre- and post-condition for this problem.

   c) Give a recursive algorithm for solving this problem, using something close to Python notation. You may assume that we’ve given you a function convert_one_digit that turns one digit (such as 5) into a string of length 1 (e.g. "5"). Use only convert_one_digit, string concatenation (+), "if" (which we always use with recursion), and simple integer arithmetic and comparisons (such as +, -, *, // (integer division), % (remainder), <, >, <=, >=, ==, != (not equal)). do not use the Python “str” function, which does this automatically.

4. Describe the precondition of the % operation (remainder), and give a "joint postcondition" for integer division and remainder - in other words, if we define q as x//y and r as x%y, how could we check whether q and r have the right values, without using either division or remainder in the check?
5. Given any correct function with the precondition and postcondition shown below, consider the possible impact of four possible changes in the precondition and postcondition, namely changing one < into a <= or one <= into a <. For some of these changes, we will not need to change the function body, no matter what it is — for which changes is this true, and why?

```python
def brainsample(x):
    precondition(isinstance(x, int) && 0<x && x<=10)
    result = ... some horrible complex calculation ...
    postcondition(100<result && result <= 1000)
    return result
```

6. Give a call tree for the function call `bizzare(3)`, where `bizzare` is defined like this:

```python
def bizarre(x):
    precondition(isinstance(x, int))
    if (x <= 1):
        return 1
    else:
        return bizarre(x-1)+bizzare(x-1) + bizarre(x//2)
```

7. Give a "call tree" for the call `fractions(3)`. The function `str` takes an integer and turns it into a String (i.e. it is an answer to Question 3).

```python
def all_fractions(x, y):
    precondition(isinstance(x, int) and isinstance(y, int))
    if (x > 0):
        return str(x) + "/" + str(y) + " " + all_fractions(x-1, y)
    else:
        return ""

def fractions(z):
    precondition(isinstance(z, int))
    if (z > 0):
        return all_fractions(z-1, z) + fractions(z-1)
    else:
        return ""
```

8. What is the value returned by the following function called as `Ack(2,2)`? Give a "call tree" diagram to show how this result is produced.

```python
def Ack(x, y):  # "Ackerman's function"
    precondition(isinstance(x,int) and isinstance(y,int) and x>=0 and y>=0)
    if x == 0:
        return y+1
    elif (y == 0):
        return Ack(x-1, 1)
    else:
        new_y = Ack(x, y-1)
        return Ack(x-1, new_y)
    # or, equivalently, just return Ack(x-1, Ack(x, y-1))
9. Produce the value of \texttt{bizzare(3)} [from Question 6] by substitution.

10. Produce either \texttt{Ack(1,2)} or \texttt{Ack(2,1)} [from Question 8] by substitution (you may want to think about which will be easier, based on your answer to Question 8).

11. Give an informal explanation of the progress of Ackerman’s function (Question 8) — how do we know this won’t run forever? (It is actually quite difficult to fit this into the framework from the book of “come up with a progress expression that goes down by at least 1 and upon reaching 0 ensures that there is no recursion”).

12. The course notes do not (currently) include a proof for the \texttt{power_with_hint} function of Figure 4.15 of Chapter 4.6 “Advanced Recursive Design”. Give this proof (on an actual exam, you might be asked for something shorter, such as just one of the three parts of the proof).

13. One good characteristic of a test suite is that it provides good “coverage” of an algorithm that solves the problem. “Coverage” can be measured in several ways (in fact, one of your professors helped develop one of the definitions of coverage). For the algorithms we have written so far, we may wish to achieve 100% “statement coverage”: a test suite provides 100% statement coverage for an algorithm if the process of running the algorithm with each test in the suite causes every statement in the program (i.e., every single return, if, and variable definition) to be executed. For example, in Question 7, a test suite that consisted of the tests “\texttt{bizzare(0)}” and “\texttt{bizzare(1)}” does not have 100% statement coverage, since it would never cause the execution of the 2nd return statement. However, the single test “\texttt{bizzare(2)}” gives 100% statement coverage.

   a) Give a test suite providing 100% statement coverage for the algorithm given in Question 1.

   b) Does your test suite for Question 2a give 100% statement coverage of your algorithm for Question 2?

   c) Does your test suite for Question 3a give 100% statement coverage of your algorithm for Question 3?

   d) Do you think it is more important for a test suite to provide good statement coverage, or for a test suite to give a good variety of “kinds of questions”? Explain your answer.