Multiple-Choice Questions:

1. True or false? Generally, in practice, developers exhaustively test software.
    a. True
    b. False

2. True or false? All "real" software contains bugs.
    a. True
    b. False

3. Which of the following is not a desirable quality of a unit test?
    a. No I/O
    b. Fast
    c. Non-deterministic
    d. Tests one property
    e. None of the above

4. Which of the following is true of *exhaustive testing*?
    a. Generally infeasible in practice
    b. Tests all possible inputs
    c. Typically results in an intractably large set of test cases even for small programs
    d. All of the above
    e. None of the above

Solutions:

1. b
2. a
3. c
4. d

Problem: Following test-driven development (TDD), fill in the blanks/cells in the following description of steps typically associated with TDD.

Step name	Short description of the step
#1 Red	
#2 Green	
#3 _____	

Solution:

Step name	Short description of the step
#1 Red	Write a test for the functionality you're building. Of course, the test fails ("red") because you haven't implemented the functionality yet.
#2 Green	Implement the functionality to make the test pass.
#3 Refactor	Clean up any duplication, old code, ugliness, etc.

Multiple-Choice Questions:

1. Which type(s) of tests do you typically write when doing test-driven development?

    a. Automated tests

    b. Unit tests

    c. Blackbox tests

    d. All of the above

    e. None of the above

Solution:

1. d

Problem: Following test-driven development (TDD), describe in plain English how you would go about adding a new method `index2br` to `RentalsController` along with an associated ERB. The purpose of this functionality would be to display an index of all rentals with 2 bedrooms. Describe the sequence of steps you would perform and all the tests you would write. Cover all aspects of TDD.

```ruby
class RentalsController < ApplicationController
  def index
    @rentals = Rental.all
  end

  def show
    # YOUR ANSWER HERE
  end

  def new
    @rental = Rental.new
  end

  def edit
    @rental = Rental.find(params[:id])
  end

  def create
    @rental = Rental.new(rental_params)
    respond_to do |format|
      if @rental.save
        format.html { redirect_to @rental, notice: 'Rental was successfully created.' }
        format.json { render action: 'show', status: :created, location: @rental }
      else
        format.html { render action: 'new' }
        format.json { render json: @rental.errors, status: :unprocessable_entity }
      end
    end
  end
```

... and so on ...

Solution:

Red: Following TDD, I would first write tests for the new functionality. Since these would be black box tests, I would have one or more "normal" cases (e.g., a mix of 2-bedroom and non-2-bedroom rentals). I would also have some boundary cases: no rentals at all, no 2-bedroom rentals, only 2-bedroom rentals, etc. Having written the tests, I would run them to see that they fail.

Green: I would then write just enough code to make the test pass. (the index2br method and ERB)

Refactor: Finally, if I saw an opportunity to improve my design after implementing the code, I would refactor the code, improving the design.

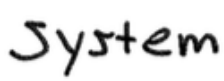
Short-Answer Questions:

1. In _____ testing, you hook everything together and treat the system like a black box.

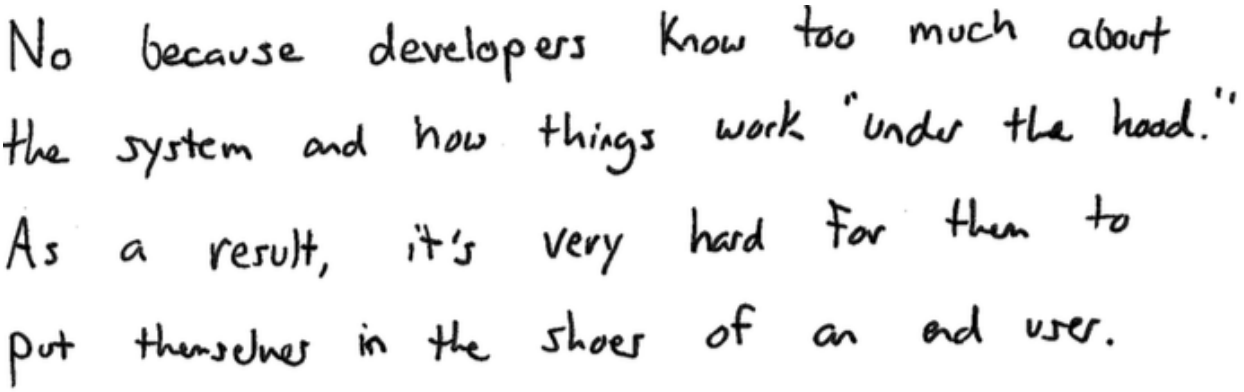
2. When it comes to the type of testing from the previous question, should the software developers who wrote the code perform the testing? Why?

Solutions:

1.



2.



Problem: Give two reasons why it's bad for developers to system test their own code.

Solution:

Many possible answers...

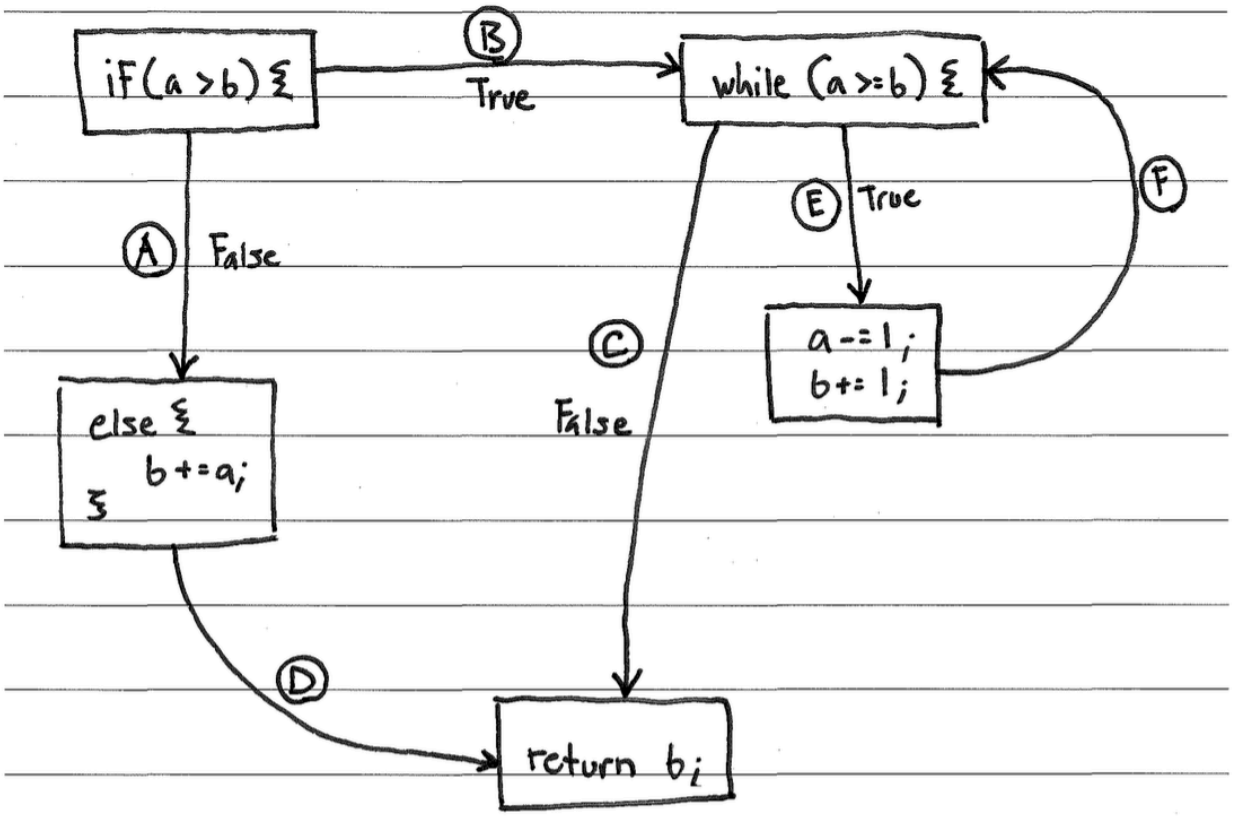
Here [redacted] two good ones:

- Developer can't see system like a user does (he/she knows too much about its inner workings)
- Because developer made it, he/she may have disincentive to find problems

Problem: Draw a control flow diagram for this function. Label each edge with an uppercase letter.

```
int funWithNumbers(int a, int b) {
    if (a > b) {
        while (a >= b) {
            a -= 1;
            b += 1;
        }
    } else {
        b += a;
    }
    return b;
}
```

Solution:


iF(a >b) {
B
True
while (a >=b) {
A
False
E
True
F
a -=1;
b+= 1;
C
False
else {
b+=a;
}
D
return b;

Problem: Fill in the table below with a test suite that provides *path coverage* of the code from the previous question. Cover no more than 2 iterations of the loop. In the covers column, list the relevant labeled items in your CFG that each test case covers. If there is some part of the coverage that is impossible to cover, then list it in the covers column, and put "N/A" in the associated x and y cells. Some cells in the table may be left blank.

Input		**Covers**
x	**y**	

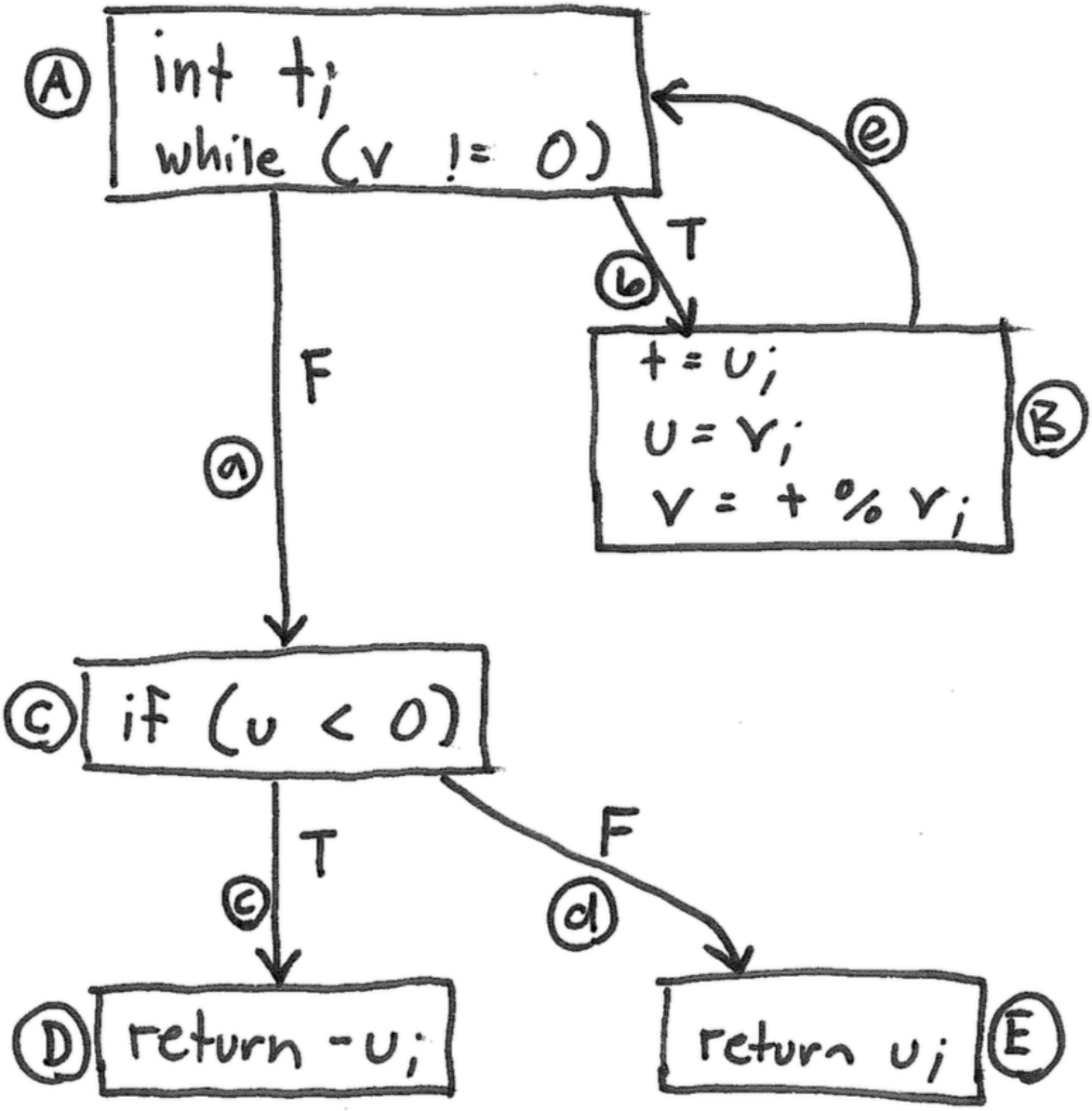
Solution:

Input		Covers
x	y	
1	2	AD
N/A	N/A	BC
1	0	BEFC
4	2	BEFEFC

Problem: Draw a control flow diagram for this function. Label each node in the graph with a capital letter, and label each edge with a lowercase letter.

```
int blammo(int u, int v) {
  int t;
  while (v != 0) {
    t = u;
    u = v;
    v = t % v; // Recall that % computes remainder of t/v
  }
  if (u < 0) { return -u; }
  return u;
}
```

Solution:


A
int t;
while (v != 0)
e
T
b
t = u;
u = v;
v = t % v;
B
F
a
c
if (u < 0)
T
F
c
d
D
return -u;
return u;
E

Problems:

1. Fill in the table below with a test suite that provides statement coverage of the "blammo" code. In the covers column, list the relevant labeled items in your CFG that each test case covers. Some cells in the table may be left blank.

Input		Covers
u	v	

2. Fill in the table below with a test suite that provides path coverage of the "blammo" code. Cover no more than 1 iteration of the loop. In the covers column, list the relevant labeled items in your CFG that each test case covers. Some cells in the table may be left blank.

Input		Covers
u	v	

Solutions:

1.

Input		Covers
u	v	
2	2	A, B, C, E
-1	0	A, C, D

2.

Input		Covers
u	v	
-1	0	a, c
0	0	a, d
-2	-2	b, e, a, c
2	2	b, e, a, d

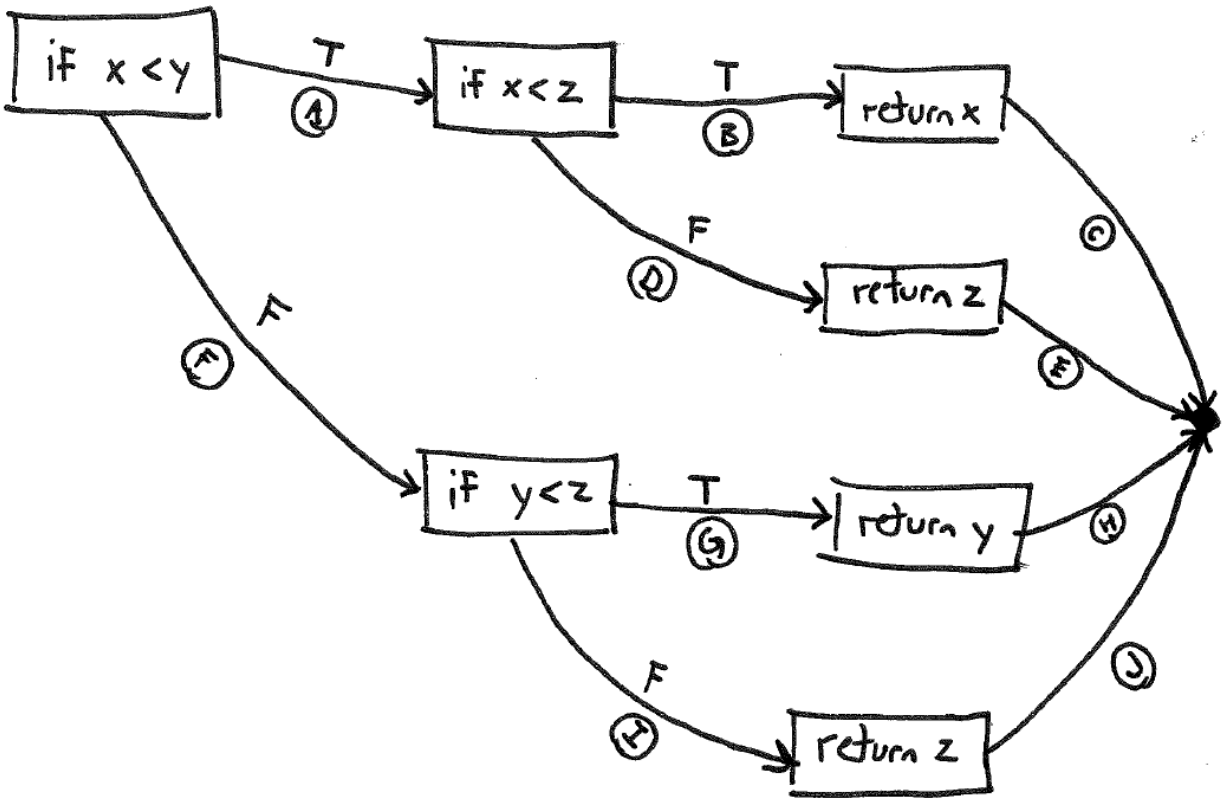
Paths:

a, c
a, d
b, e, a, c
b, e, a, d

Problem: Draw a control-flow graph for the following function. Label each edge in the graph with an uppercase letter.

```
def min_of_three(x, y, z)
  if x < y then
    if x < z then
      return x
    else
      return z
    end
  else
    if y < z then
      return y
    else
      return z
    end
  end
end
```

Solution:


if x<y
T
1
if x<z
T
B
return x
C
F
D
return z
E
F
if y<z
T
G
return y
H
F
I
return z
J

Problem: Fill in the table below with a test suite that provides path coverage of the `min_of_three` function from the previous question. In the covers column, list the relevant labeled edges in your CFG that each test case covers. Some cells in the table may be left blank.

Input			Expected Output	Covers
x	y	z		

Solution:

Input			Expected Output	Covers
x	y	z		
1	2	2	1	A, B, C
2	3	1	1	A, D, E
2	1	2	1	F, G, H
3	2	1	1	F, I, J