CSCE 747 - Final
Name:

This is a 150-minute exam. On all essay type questions, you will receive points based on the quality of the answer - not the quantity.

Make an effort to write legibly. Illegible answers will not be graded and awarded 0 points.

There are a total of 10 questions and 100 points available on the test.

Question 1

1. A program may be correct, yet not reliable.
    a. **True**
    b. False

2. If a system is on an average down for a total 30 minutes during any 24-hour period:
    a. **Its availability is about 98% (approximated to the nearest integer)**
    b. Its reliability is about 98% (approximated to the nearest integer)
    c. Its mean time between failures is 23.5 hours
    d. Its maintenance window is 30 minutes

3. In general, we need either stubs or drivers but not both, when testing a module.
    a. True
    b. **False**

4. Which of the following may be logically inferred from the post-condition of a sorting routine, sort(array, size) that sorts elements in ascending order?
    a. size > 0
    b. ∃ i, j, 0 <= i < j < size : array[i] = array[j]
    c. ∀ i, j, 0 <= i < j < size : array[i] < array[j]
    d. **∀ i, j, 0 <= i < j < size : array[i] <= array[j]**

5. If a temporal property holds for a finite-state model of a system, it holds for any implementation that conforms to the model.
    a. **True**
    b. False

6. Self-check oracles do not require the expected output for judging whether a program passed or failed a test.
    a. **True**
    b. False

7. Object-oriented design and implementation typically have an impact on verification such that OO specific approaches are required for:
    a. **Unit Testing**
    b. **Integration Testing**
    c. System Testing
    d. Acceptance Testing

Question 2

Metaheuristic search techniques can be divided into local and global search techniques. Define what a “local” search and a “global” search is. Contrast the two approaches. What are the strengths and weaknesses of each?

Local search techniques formulate a solution, and attempt to improve that solution by making small changes (looking for a better solution in the “local neighborhood” - the possible solutions formed by making one small change). Global searches typically form more than one solution at a time, and freely change those solutions (moving to any spot in the search space).

Local searches are often very fast, easy to implement, and easy to understand conceptually. However, they depend strongly on the choice of initial guess. They can easily get stuck in local optima - where they find the best solution possible given the neighborhood, but not the best for the whole search space. This weakness can be partially overcome by allowing restarts.

Global searches are harder to implement and are often slower, but have no problems with becoming stuck, as they try more than one solution at once. However, because they are slower, they may not find as good of a solution given the same time budget.

Question 3

Consider the following function:

```
bSearch(A, value, start, end) {
        if (end <= start)
                return -1;
        mid = (start + end) / 2;
        if (A[mid] > value) {
                return bSearch(A, value, start, mid);
        } else if (value > A[mid]) {
                return bSearch(A, value, mid+1, end);
        } else {
                return mid;
        }
```

```
}
```

Give an example, with a brief justification, for each of the following kinds of mutants that may be derived from the code by applying mutation operators of your choice. Do not reuse a mutation, even if it fits multiple categories.

1. Equivalent Mutant

```
bSearch(A, value, start, end) {
        if (end <= start)
                return -1;
        mid = (start + end) / 2;
        if (A[mid] > value) {
                return bSearch(A, value, start, mid);
        } else if (value > A[mid]) {
                return bSearch(A, value, mid+1, end);
        } else {
        }
        return mid;
}
```

SES - End Block Shift

2. Invalid Mutant

```
bSearch(A, value, start, end) {
        if (end <= start)
                return -1;
        mid = (start + end) / 2;
        if (A[mid] > value) {
                return bSearch(A, value, start, mid);
        } else if (value > A[mid]) {
                return bSearch(A, value, mid+1, end);
        } else {
                return mid;
        }
}
```

SDL - Statement Deletion

3. Valid, but not Useful

```
bSearch(A, value, start, end) {
        if (end > start)
                return -1;
        mid = (start + end) / 2;
        if (A[mid] > value) {
                return bSearch(A, value, start, mid);
        } else if (value > A[mid]) {
                return bSearch(A, value, mid+1, end);
        } else {
                return mid;
        }

}
```

ROR - Relational Operator Replacement

4. Useful Mutant

```
bSearch(A, value, start, end) {
        if (end <= start)
                return -1;
        mid = (start + end) / 2;
        if (A[mid] > value) {
                return bSearch(A, value, start, mid);
        } else if (value > A[mid]) {
                return bSearch(A, value, mid+2, end);
        } else {
                return mid;
        }

}
```

CRP - Constant for Constant Replacement

Question 4

Suppose that finite state verification of an abstract model of some software exposes a counter-example to a property that is expected to hold for true for the system. Briefly describe what follow-up actions would you take and why?

This tells us that a property we expect to hold is not held by the model. This implies one of the following:

- **There is an issue with the model. The model is made by interpreting the requirements, and there could be a mistake in the model (fault in the code, bad assumptions, incorrect interpretation).**
- **There is an issue with the property. The property may not say what you intended it to say.**
- **There is an issue with your requirements. Two requirements may contradict, or a requirement may be written incorrectly.**

The action you take depends on which is true. You should look at each angle, and find the source of the problem.

Question 5

You are building a web store that you feel will unseat Amazon as the king of online shops. Your marketing department has come back with figures stating that - to accomplish your goal - your shop will need an **availability** of at least 99%, a **probability of failure on demand** of less than 0.1, and a **rate of fault occurrence** of less than 2 failures per 8-hour work period.

You have recently finished a testing period of one week (seven full 24-hour days). During this time, 972 requests were served to the page. The product failed a total of 64 times. 37 of those resulted in a system crash, while the remaining 27 resulted in incorrect shopping cart totals. When the system crashes, it takes 2 minutes to restart it.

1. What is the rate of fault occurrence?
2. What is the probability of failure on demand?
3. What is the availability?
4. What additional information would you need to calculate the mean time between failures?
5. Is the product ready to ship? If not, why not?

1. **64/168 hours = 0.38/hour = 3.04/8 hour work day**
2. **64/972 = 0.066**
3. **It was down for (37*2) = 74 minutes out of 168 hours = 74/10089 minutes = 0.7% of the time. Availability = 99.3%**
4. **We need timestamps. We know how long it is down (on average), but not when each crash occurs.**
5. **No. Availability, POFOD are good. ROCOF is too low. How would you improve it?**

Question 6

Temporal Operators: A quick reference list.

- G p: p holds globally at every state on the path
- F p: p holds at some state on the path
- X p: p holds at the next (second) state on the path
- p U q: q holds at some state on the path and p holds at every state before the first state at which q holds.
- A: for all paths from a state
- E: for some path from a state

Consider a finite state model of a traffic-light controller similar to the one discussed in the homework, with a pedestrian crossing and a button to request right-of-way to cross the road.

State variables:

- **traffic_light: {RED, YELLOW, GREEN}**
- **pedestrian_light: {WAIT, WALK, FLASH}**
- **button: {RESET, SET}**

Initially: **traffic_light = RED, pedestrian_light = WAIT, button = RESET**

Transitions:
pedestrian_light:

- **WAIT → WALK if traffic_light = RED**
- **WAIT → WAIT otherwise**
- **WALK → {WALK, FLASH}**
- **FLASH → {FLASH, WAIT}**

traffic_light:

- **RED → GREEN if button = RESET**
- **RED → RED otherwise**
- **GREEN → {GREEN, YELLOW} if button = SET**
- **GREEN → GREEN otherwise**
- **YELLOW→ {YELLOW, RED}**

button:

- **SET → RESET if pedestrian_light = WALK**
- **SET → SET otherwise**
- **RESET → {RESET, SET} if traffic_light = GREEN**
- **RESET → RESET otherwise**

1. Briefly describe a safety-property (nothing “bad” ever happens) for this model and formulate it in CTL.
2. Briefly describe a liveness-property (something “good” eventually happens) for this model and formulate it in LTL.
3. Write a trap-property that can be used to derive a test case using the model-checker to exercise the scenario “pedestrian obtains right-of-way to cross the road after pressing the button”.

1. **AG (pedestrian_light = walk -> traffic_light != green)**
   The pedestrian light cannot indicate that I should walk when the traffic light is green. This is a safety property. We are saying that something should NEVER happen.

2. **G (traffic_light = RED & button = RESET -> F (traffic_light = green))**
   If the light is red, and the button is reset, then eventually, the light will turn green. This is a liveness property, as we assert that something will eventually happen.

3. **First, we should formulate the property in a temporal logic, than translate into a trap property:**
   G (button = SET -> F (pedestrian_light = WALK))
   This states that, no matter what happens, if the button is pressed, then eventually the pedestrian light will signal that I can cross the street. This is a liveness property.

   A trap property takes a property we know to be true (like this), then negates it. By negating it, we assert that this property is NOT true. The negated form is:
   G !(button = SET -> F (pedestrian_light = walk))

   Because it is actually true, the model checker gives us a counter-example showing one concrete scenario where the property is true. This is a test case we can use to test our real program.

Question 7

Consider the following C function:

```
int sum(int arr[], int n){
      int s = 0;
      while (n > 0){
            n = n - 1;
            s = s + arr[n];
      }
      return s;
}
```

1. What are the pre and post-conditions of this method?
2. For each line of code, derive the state predicates that would be collected by symbolic execution.

1. This function takes in an array, and returns the sum of the elements in that array. Therefore, the pre-conditions are:
   (n = N, N = |arr|, N >= 0)
   That is, the integer n passed in has a concrete value N. That concrete value must be equal to the length of the array to get the correct answer. Finally, the array must exist (we want to protect from trying to access an index below 0).
   Then, the post-conditions are:
   (s = S, S = $\sum_{i=0}^{N-1}$ arr[i])
   In plain terms, s will have a concrete value S, and that value is the sum of all elements of arr.
2.

```
int sum(int arr[], int n){
      int s = 0;
      while (n > 0){

            n = n - 1;

            s = s + arr[n];

      }
```

(n = N, N = |arr|, N >= 0)

(n = N, N = |arr|, N >= 0, s = 0)

On first entry: (n = N, N = |arr|, N > 0, s = 0)
Always true: (n = N, N > 0, s = S, S= $\sum_{i=N}^{|arr|}$ arr[i])

On first entry: (n = N, N = N - 1, N >= 0, s = 0)
Always true: (n = N, N = N - 1, N >= 0, s = S, S= $\sum_{i=N}^{|arr|}$ arr[i])

On first entry: (n = N, N >= 0, s = S, S = S + arr[N])
Always true: (n = N, N >= 0, s = S, S= $\sum_{i=N}^{|arr|}$ arr[i])

If loop does not execute: (n = N, N = 0, |arr| = 0, s = 0)

Always true: (n = N, N = 0, s = S, S= $\sum_{i=N=0}^{|arr|}$ arr[i])

```
        return s;
    }
```

In forming the symbolic constraints, you need to examine each line of code to determine how it transforms the existing variables and how it defines new variables. The important part here is the loop. You can't just look at what is true the first time through the loop - you need to look at what is true on every execution. In doing so, you can see how the value of s changes - it is a sum over all elements from the current value of N to |arr| of the entries in those positions of the array. This shows that the post-condition is being met.

Question 8

Search-based test generation generally does not use coverage criteria directly in generation, but instead makes use of a scoring function to determine how clost tests are to achieving the current generation goal (such as coverage of the criterion). This is called the fitness function, or objective function.

Explain the fitness function formulation used by search-based generation to achieve branch coverage of a program. It may be helpful to come up with a code example.

Normally, branch coverage is measured as the proportion of obligations covered to the total number of obligations. This does serves as a score. However, this is not an ideal fitness function, as it offers no feedback to the search process. Instead, fitness functions for branch coverage measure the distance of the current solution from the ideal solution.

Instead of raw coverage, the fitness function uses the branch distance and approach level:
fitness(s,b) = AL(s,b) + normalize(BD(s,b))

Approach level - count of the branch's control-dependent nodes not yet executed.
Branch distance - if the other branch is taken, measure how close the target branch was from being taken.

Consider the following code:

```
if(x < 10){ // Node 1
        // Do something.
}else if (x == 10){ // Node 2
        // Do something else.
}
```

If the true evaluation of Node 2 is our target (x == 10):

- **If x<10 is true, approach level = 1. If x >= 10, approach level is 0.**
- **If x < 10 is true, branch distance = 1 (the target branch predicate was never evaluated).**
- **If x==10 evaluates to true, then the branch distance = 0 (we hit the target branch).**
- **If x==10 is executed and evaluates to false, branch distance = (abs(x-10)+k).**
    - **This tells us that we got here, but that we didn't get the outcome we wanted.**
    - **Better, it tells us how close we are to the outcome we want.**

Question 9

Consider a simple microwave controller modeled as a finite state machine using the following state variables:

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press (assumes at most one at a time)
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

Formulate the following informal requirements in CTL:

1. The microwave shall never cook when the door is open.
2. The microwave shall cook only as long as there is some remaining cook time.
3. If the stop button is pressed when the microwave is not cooking, the remaining cook time shall be cleared.

1. **AG (Door = Open -> !Cooking)**
2. **AG (Cooking -> Timer > 0)**
3. **AG (Button = Stop & !Cooking -> AX (Timer = 0))**

Formulate the following informal requirements in LTL:

1. It shall never be the case that the microwave can continue cooking indefinitely.
2. The only way to initiate cooking shall be pressing the start button when the door is closed and the remaining cook time is not zero.
3. The microwave shall continue cooking when there is remaining cook time unless the stop button is pressed or the door is opened.

1. **G (Cooking -> F (!Cooking))**
2. **G (!Cooking U ((Button = Start & Door = Closed) & (Timer > 0)))**
3. **G ((Cooking & Timer > 0) -> X (((Cooking | (!Cooking & Button = Stop)) | (!Cooking & Door = Open)))**

Question 10

You are testing the following method:
`public double max(double a, double b);`

Devise four executable test cases for this method in the JUnit notation.

Examples: Please also explain your tests.

```
@Test
  public void aLarger() {
    double a = 16.0;
    double b = 10.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertTrue("should be larger", actual>b);
    assertEquals(expected, actual);
  }

@Test
  public void bLarger() {
    double a = 10.0;
    double b = 16.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertThat("b should be larger", b>a);
    assertEquals(expected, actual);
  }

@Test
  public void bothEqual() {
    double a = 16.0;
    double b = 16.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertEquals(a,b);
    assertEquals(expected, actual);
  }

@Test
```

```
public void bothNegative() {
  double a = -2.0;
  double b = -1.0;
  double expected = -1.0;
  double actual = max(a,b);
  assertTrue("should be negative",actual<0);
  assertEquals(expected, actual);
}
```