DIT635 - Practice Examination

There are a total of 15 questions on the test (there will be fewer on the real exam - we gave you some extra questions to study with). On all essay type questions, you will receive points based on the quality of the answer - not the quantity. Write carefully - illegible answers will not be graded.

Question 1 (Warm Up)

Note - Multiple answers may be correct. Indicate all answers that apply.

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 drivers or mock objects, but not both, when testing a module.
    - a. True
    - **b. False**

4. 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

5. A test suite that meets a stronger coverage criterion will find any defects that are detected by any test suite that meets only a weaker coverage criterion
    - a. True
    - b. **False**

6. A test suite that is known to achieve Modified Condition/Decision Coverage (MC/DC) for a given program, when executed, will exercise, at least once:
    - **a. Every statement in the program.**
    - **b. Every branch in the program.**
    - c. Every combination of condition values in every decision.
    - d. Every path in the program.

7. The Category-Partition Testing technique requires identification of:
    - **a. Testing Choices**

  b. **Representative Values**
  c. Def-Use pairs
  d. Pairwise combinations

8. Validation activities can only be performed once the complete system has been built.
    a. True
    b. **False**

9. The statement coverage criterion never requires as many test cases to satisfy as branch coverage criterion.
    a. True
    b. **False**

10. Requirement specifications are not needed for selecting inputs to satisfy structural coverage of program code.
    a. **True**
    b. False

11. A system that fails to meet its user's needs may still be:
    a. **Correct with respect to its specification.**
    b. **Safe to operate.**
    c. **Robust in the presence of exceptional conditions.**
    d. **Considered to have passed verification.**

Question 2 (Quality Scenarios)

Consider the software for air-traffic control at an airport (say, GOT). Air traffic control (ATC) is a service provided by ground-based air traffic controllers (the users of this system) who direct aircraft on the ground and through controlled airspace with the help of the software. The purpose of this software is to prevent collisions, organize and expedite the flow of air traffic, and provide information and other support for pilots.

The software offers the following features:

- Monitors the location of all aircraft in a user's assigned airspace.
- Communication with the pilots by radio.
- Generation of routes for individual aircraft, intended to prevent collisions.
- Scheduling of takeoff for planes, intended to prevent potential collisions.
- Alerts of potential collisions based on current bearing of all aircraft.
    - To prevent collisions, ATC applies a set of traffic separation rules, which ensure each aircraft maintains a minimum amount of empty space around it at all times.
    - The route advice can be either of "mandatory" priority (to prevent an imminent collision, pilots should follow this command unless there is a good reason not to) or "advisory" priority (this advice is likely to result in a safe route, but a pilot can choose to ignore it).

You may add additional features or make decisions on how these features are implemented, as long as they fit the overall purpose of the system. In any case, state any assumptions that you make.

Identify one performance, one availability, and one security requirement that you think would be necessary for this software and develop a quality attribute scenario for each.

Sample Solution

Requirements should be specific and testable. Scenarios should have single stimuli and specific, measureable system responses

<u>Performance Requirement:</u> Under normal load, displayed aircraft positions shall be updated on a user's display at least every 50 ms.

<u>Performance Scenario</u>: Responsiveness

- ***Overview: Check system responsiveness for displaying updated aircraft positions***
- ***System state: System is under normal load (defined as the Deployment environment working correctly with less than 500 tracked aircraft).***
- ***Environment state: Less than 500 physical aircraft are in the airspace. All are being tracked successfully.***
- ***External stimulus: 50 Hz update of ATC system display.***

- ***Required system response: radar/sensor values are computed and fused, new position is displayed to the air traffic controller with maximum error of 5 meters.***
- ***Response measure: Fusion and display process completes in less than 45 ms 95% of the time, and in less than 50 ms 99% of the time. There is an absolute deadline of 55 ms.***

***Availability Requirement:** The system shall be able to tolerate the failure of any single server host, graphics card, display or network link.*

***Availability Scenario:** primary display card fails during screen refresh*

- ***Overview: One of the monitor display cards fails during transmission of a screen refresh***
- ***System state: System is working correctly under normal load with no failures.***
- ***Environment state: No relevant details.***
- ***External stimulus: A display card fails***
- ***Required system response: The display window manager system will detect the failure within 10 ms and route display information through a spare redundant graphics card with no user-discernable change to ATC aircraft display. The graphics card failure will be displayed as an error message at the bottom right corner of the ATC display.***
- ***Response measure: There will be no loss in continuity of visual display and failover with visual warning will complete within 1s.***

***Security Requirement:** The system shall maintain audit logs of any logins to the ATC database, containing sufficient information to identify an attacker.*

***Security Scenario:** malicious login and audit policy demonstration*

- ***Overview: A malicious agent gains access to the flight records database in the ATC.***
- ***System state: The system is working correctly under normal load.***
- ***Environment state: No relevant environmental factors.***
- ***External stimulus: A malicious agent obtains access to the flight records database through password cracking, and downloads flight plans for commercial aircraft.***
- ***Required system response: An audit log will be updated with login and download information to support future prosecution of malicious users.***
- ***Response measure: The system audit contains time, IP address, and related information for the download. This information will assist in identifying and analyzing possible attacks.***

Question 3 (Quality)

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. Is the product ready to ship? If not, why not?

Sample Solution

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. **No. Availability, POFOD are good. ROCOF is too low. How would you improve it?**

Question 4 (System Testing - Category-Partition Method)

(Note - this is a much longer question than would be on the real exam. However, it is a nice detailed example for studying.)

The airport connection check is a high-level function exposed by the API of a travel reservation system. It is intended to check the validity of a single connection between two flights in an itinerary. For example, a traveler may intend to fly from Gothenburg to Los Angeles, but there is a connection through Frankfurt. Therefore, their itinerary is Gothenburg -> Frankfurt (Flight A) and Frankfurt -> Los Angeles (Flight B).

This service will ensure that the connection through Frankfurt is a valid one. For example, if the arrival airport of Flight A differs from the departure airport of Flight B, the connection is invalid. That is, if we pass in two flights, and Flight A arrives in Frankfurt, but Flight B departs from Munich, it is not a valid connection.

Likewise, if the departure time of Flight B is too close to the arrival time of Flight A, the connection is invalid. If Flight A arrives in Frankfurt at 8:00, and Flight B departs at 8:05, there is not sufficient time to complete the customs process and board the flight.

```
validConnection(Flight flightA, Flight flightB)
                returns ValidityCode
```

A `Flight` is a data structure consisting of:

- A unique identifying flight code (string, three characters followed by four numbers).
- The originating airport code (three character string).
- The scheduled departure time from the originating airport (in universal time).
- The destination airport code (three character string).
- The scheduled arrival time at the destination airport (in universal time).

There is also a **flight database**, where each record contains:

- Three-letter airport code (three character string).
- Airport country (two character string).
- Minimum connection time (integer, minimum number of minutes that must be allowed for flight connections to be valid).

`ValidityCode` is an integer with value 0 for OK, 1 for invalid airport code, 2 for a connection that is too short, 3 for flights that do not connect (Flight A does not land in the same location that Flight B departs from), or 4 for any other errors (malformed input or any other unexpected errors).

Design system test cases using the category-partition method for the `validConnection` function.

1. Identify choices (aspects that you control and that can vary the outcome) for the two input flights and the database.
2. For each choice, identify a set of representative values.

3. Apply ERROR, SINGLE, and IF constraints.
    a. ERROR = This representative value will trigger an error no matter that it is paired with.
    b. SINGLE = This representative value should give an OK response, but we want to make sure we try it once.
    c. IF = This representative value can only be used if a certain value is set for another choice.

Note - you do not need to form actual test specifications or concrete test cases as part of your answer.

Sample Solution

Recall the lectures on system testing.

The approximate process of writing system tests is the following:

1. **For each high-level independently testable feature surfaced by an interface, you need to identify the parameters. These can be explicit (passed into the function) or implicit (configuration options or other environmental factors - such as databases - that influence the outcome of the function).**
2. **Each parameter can be manipulated in many ways through testing. For each parameter, you must identify choices - aspects of that input that you can vary, and that will have some impact on the outcome of testing this function. For example, if an input is a data structure, the choices might include fields of that data structure that impact the outcome of the function. If that data structure is serialized from a file, then choices may include the status of the file (whether that file exists, is corrupted, and so on).**
3. **You cannot exhaustively test a function, there are too many possible values that can be fed in. So, instead, you partition the input domain for each choice into representative values (types of input). If you try at least one concrete input from each of these value types, you should trigger different outcomes and be more likely to notice faults. We discussed some methods of performing this partitioning in class.**
4. **Once representative values are chosen, you can form test specifications - abstract recipes for tests - by choosing one value for each choice. You can transform these recipes into actual test cases by coming up with concrete input values that fit into the category of value for each choice.**

In this exercise, you have been asked to perform Steps 1-3 above - identify the parameters, split the parameters into choices, then partition the input values for each choice into representative values. You can constrain the number of test specifications by adding further constraints (ERROR, SINGLE, IF) that note when certain representative values should be used in combination with other values. Note that you do not have to use all constraints (i.e., you do not need to use SINGLE unless it makes sense to do so).

This function has two explicit inputs - the two flights - and an implicit input - an airport database. A flight is a complex data structure containing several fields, each of those

fields represents a controllable input category. Your choices should revolve around those fields.

Remember that the function's parameters may influence each other (testing this function requires considering both Flight A and B's field values as well as what is in the database), so the representative values must reflect how different choices or variables can interact. IF-constraints are also a good way to indicate when representative values for two choices should be paired.

Parameter: FlightA

Choice: Flight code:
- ***Malformed (string that does not follow stated formatting convention) [error]***
- ***not in database [error] (Note: I assumed here that there was a flight database too. State any assumptions you make in your answer.)***
- ***valid***

Choice: Originating airport code:
- ***malformed (not a three-letter string) [error]***
- ***not in database [error]***
- ***valid city***

Choice: Scheduled departure time:
- ***malformed (not following formatting convention) [error]***
- ***out of legal range (not a valid time) [error]***
- ***legal***

Choice: Destination airport (transfer airport - where connection takes place):
- ***malformed (not a three-letter string) [error]***
- ***not in database [error]***
- ***valid city***

Choice: Scheduled arrival time (tA):
- ***malformed (not following formatting convention) [error]***
- ***out of legal range (not a valid time) [error]***
- ***legal***

Parameter: FlightB

Choice: Flight code:
- ***Malformed (string that does not follow stated formatting convention) [error]***
- ***not in database [error]***
- ***valid***

Choice: Originating airport code:
- ***Malformed (not a three-letter string) [error]***
- ***not in database [error]***
- ***differs from transfer airport [error]***

- ***same as transfer airport***

Choice: Scheduled departure time:
- ***malformed (not following formatting convention) [error]***
- ***out of legal range (not a valid time) [error]***
- ***before arriving flight time (tA) [error]***
- ***between tA and tA + minimum connection time (CT) [error]***
- ***equal to tA + CT [single]***
- ***greater than tA + CT***

Choice: Destination airport code:
- ***malformed (not a three-letter string) [error]***
- ***not in database [error]***
- ***valid city***

Choice: Scheduled arrival time:
- ***malformed (not following formatting convention) [error]***
- ***out of legal range (not a valid time) [error]***
- ***legal***

<u>Parameter: Database record</u>

This parameter refers to the database record corresponding to the transfer airport.

Choice: Airport code:
- ***Malformed (not a three-letter string) [error]***
- ***blank [error]***
- ***valid***

Choice: Airport country:
- ***Malformed (non-string) [error]***
- ***blank [error]***
- ***Invalid (not a country) [error]***
- ***valid***

Choice: Minimum connection time:
- ***malformed (not following formatting convention) [error]***
- ***blank [error]***
- ***> 0 [error]***
- ***0 [single]***
- ***valid***

Question 5 (System Testing - Combinatorial Interaction Testing)

You are designing system-level tests for a web browser with multiple configuration options. You have extracted the following choices, with the following representative values for each:

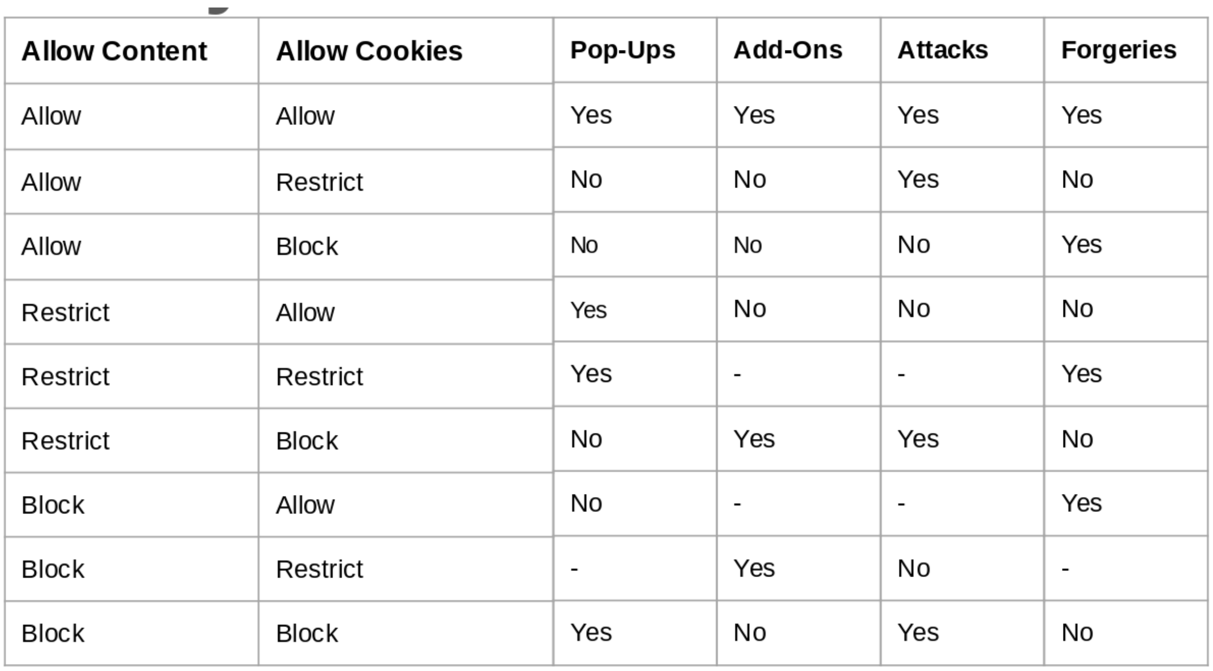
Allow Content to Load	Notify About Pop-Ups	Allow Cookies	Warn About Add-Ons	Warn About Attack Sites	Warn About Forgeries
Allow	Yes	Allow	Yes	Yes	Yes
Restrict	No	Restrict	No	No	No
Block		Block			

The full set of possible test specifications contains 144 options.

Create a covering array of specifications that covers all **pairwise value combinations** in fewer test specifications.

(hint: start with two variables with the most values and add additional variables one at a time)

Sample Solution:



Allow Content	Allow Cookies	Pop-Ups	Add-Ons	Attacks	Forgeries
Allow	Allow	Yes	Yes	Yes	Yes
Allow	Restrict	No	No	Yes	No
Allow	Block	No	No	No	Yes
Restrict	Allow	Yes	No	No	No
Restrict	Restrict	Yes	-	-	Yes
Restrict	Block	No	Yes	Yes	No
Block	Allow	No	-	-	Yes
Block	Restrict	-	Yes	No	-
Block	Block	Yes	No	Yes	No

Your specific answer might vary, but you should be able to cover this in 9 tests (3*3). In most cases (at least, those that would be given on a test), you should be able to cover

the pairwise combinations in N*M tests where N and M are the number of representative values for the two variables with the most values. Always start with those two variables, then add additional ones in order of their number of representative values).

Question 6 (Exploratory Testing)

Exploratory testing typically is guided by “tours”. Each tour describes a different way of thinking about the system-under-test, and prescribes how the tester should act when they explore the functionality of the system.

1. Describe one of the tours that we discussed in class.
2. Consider a banking website, where a user can do things like check their account balance, transfer funds between accounts, open new accounts, and edit their personal information. Describe three actions you might take during exploratory testing of this system, based on the tour you described above.

Sample Solution

1. **The supermodel tour is focused on testing the GUI of the application. It is not concerned with functional correctness (e.g., that the correct data is displayed on the screen). Rather, it is concerned with the visual appearance of the GUI and whether it is correct. It focused on whether graphical elements display in the correct locations and without “glitches” (e.g., rendering errors, size or rotation issues). It also examines timing aspects of the GUI, such as how long it takes for a mouse cursor to move, text to update on the screen, for new screens to be drawn, etc. This tour can also look for typos in displayed text, or for usability issues (e.g., suggestions on how to make the GUI easier for new users to learn how to work with).**
2. **For the banking website, you might examine:**
    a. **Click on a drop down menu and ensure that the menu displays quickly, that all required items are present and displayed correctly, and that the menu does not cause any issues when it appears over other on-screen items.**
    b. **When an account is selected, ensure that account information is displayed on the screen, that it is displayed in the correct locations, and that this information is easy for a user to see if they are searching for it on the screen (e.g., that good font, color, and size choices are made).**
    c. **When the user goes to edit personal information, ensure that the existing information is displayed on the screen and that edited segments are refreshed and displayed to the user correctly.**

Question 7 (Unit Testing)

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.

Sample Solution

```
// A is larger than B, and this should be reflected in the answer. I included multiple assertions to illustrate
ways you could show this - your answer does not need to have multiple assertions.
@Test
  public void aLarger() {
    double a = 16.0;
    double b = 10.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertTrue("a should be larger", actual > b);
    assertEquals("the result should be 16", expected, actual);
  }

// B is larger than A this time. This is basically the same test, but flipped.
@Test
  public void bLarger() {
    double a = 10.0;
    double b = 16.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertTrue("b should be larger", actual > a);
    assertEquals("the result should be 16", expected, actual);
  }

// This test has A and B as equal values. This ensures that we get the ex
@Test
  public void bothEqual() {
    double a = 16.0;
    double b = 16.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertEquals("the result should be 16", expected, actual);
  }

// Test where both are negative to show that negative values are handled correctly.
@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(“Should be -1”, expected, actual);
  }
```

Question 8 (Structural Testing)

Consider the following situation: After *carefully and thoroughly* developing a collection of tests based on the requirements and your own intuition, and running your test suite, you determine that you have achieved only 60% statement coverage. You are surprised (and saddened), since you had done a very thorough job developing the requirements-based tests and you expected the result to be closer to 100%.

1. Briefly describe two (2) things that might have happened to account for the fact that 40% of the code was not exercised during the requirements-based tests.
2. Should you, in general, be able to expect 100% statement coverage through thorough requirements-based testing alone (why or why not)?
3. Some structural criteria, such as MC/DC, prescribe obligations that are impossible to satisfy. What are two reasons why a test obligation may be impossible to satisfy?

Sample Solution

1. ***There are several reasons. The most obvious one being doing a poor job finding the black-box test cases. Since we assume we did a good job, this is not the case.***
    1. ***We are missing requirements. The requirements document is incomplete and somewhere along the development of the software these missing requirements have been informally filled in by the development team, but the requirements were never added to the requirements document. Developing black-box tests from an incomplete specification to test a more complete implementation will naturally lead to poor coverage.***
    2. ***We have large amounts of dead or inactivated code. The software may have gone through several major changes and code needed for an earlier version is now not used. This code will not be covered. Also, debugging code deactivated through some global variable will not be covered. Furthermore, any malicious code may not get covered. There are many reasons why unneeded or undesirable code might make it into the software—this code is likely to not be covered with your black-box tests.***
    3. ***There may be valid optimizations in the code. The programmers might have done some very smart things in terms of optimizing the code, but this leads to a potentially large code base that is only used in various special cases. For example, the programmer might have used some lookup tables for various trigonometric functions (implemented as a switch statement) instead of the built in trigonometric functions. With requirements-based testing you are unlikely to cover much of those switch statements.***
2. ***In general there will be optimizations, debug code, exception handling, etc. in the program that the black-box testing is quite unlikely to reveal. Thus it is highly unlikely that we will get close to 100% through black-box testing alone.***

3. ***The criterion may call for an impossible combination of conditions within a decision statement. You may have also performed defensive programming, resulting in error-handling code that cannot actually be triggered. In addition, there may be unreachable or unused code that cannot be called directly or reached through normal execution paths.***

Question 9 (Structural Testing)

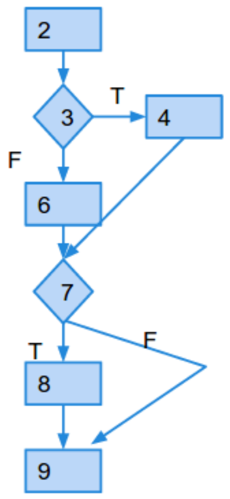
For the following function,

a. Draw the control flow graph for the program.
b. Develop test input that will provide statement coverage.
c. Develop test input that will provide branch coverage.
d. Develop test input that will provide path coverage.
e. Modify the program to introduce a fault so that you can demonstrate that even achieving path coverage will not guarantee that we will reveal all faults. Please explain how this fault is missed by your test cases.

```
1.   int findMax(int a, int b, int c) {
2.        int temp;
3.        if (a>b)
4.           temp=a;
5.        else
6.           temp=b;
7.        if (c>temp)
8.           temp = c;
9.        return temp;
10.   }
```

(Input -> Output pairs will be fine - no need for assertions or writing full JUnit cases)

Sample Solution



```
1. int findMax(int a, int b, int c) {
2.   int temp;
3.   if (a>b)
4.       temp=a;
5.   else
6.       temp=b;
7.  if (c>temp)
8.       temp = c;
9.  return temp;
10. }
```

a)
b) (3, 2, 4); (2, 3, 4)
c) (3, 2, 4); (3, 4, 1)
d) (4, 2, 5); (4, 2, 1); (2, 3, 4); (2, 3, 1)

e) ***If we have (a>b+1) in the first condition as opposed to (a>b), the tests in part D will not reveal this flaw. Only a boundary value test will.***

Question 10 (Structural Testing - Data Flow)

Identify all DU Pairs in the following code, and provide test input that achieves all DU Pairs Coverage over the code (you do not need to provide assertions or full JUnit test cases). Explain which DU Pairs are covered by each test input.

```
1. public int inflections(int[] a, int n) {
2.    int v = 0; // number of inflections
3.    int d = 0; // current run direction (+/-)
4.    while (n > 1) {
5.          n = n - 1;
6.          if ((d * (a[n]-a[n-1])) < 0) // direction change
7.                v = v + 1; // => inflection point
8.          if (a[n] != a[n-1])
9.                d = a[n] - a[n-1]; // record direction
10.   }
11.   return v;
12. }
```

Sample Solution

1. DU Pairs

Variable	**DU Pairs**
a	(1, 6), (1, 8), (1,9)
n	(1, 4), (1, 5), (5, 6), (5, 8), (5, 9), (5, 4), (5, 5)
v	(2, 7), (2, 11), (7, 7), (7, 11)
d	(3, 6), (9, 6)

2. Test Input

Input	**Additional DU Pairs Covered**
[1,2,3], 3	a: (1, 6), (1, 8), (1, 9) n: (1, 4), (1, 5), (5, 6), (5, 8), (5, 9), (5, 4), (5, 5) v: (2, 11) d: (3, 6), (9, 6)
[2,1,3], 3	v: (2, 7), (7, 11) (requires at least one inflection point)

[2, 1, 2, 1, 2], 5	v: (7, 7) (requires at least two inflection points)

Question 11 (Mutation Testing)

Consider the following function:

```
public void bSearch(int[] A, int value, int start, int 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 operator, even if it fits multiple categories.

1. Equivalent Mutant
2. Invalid Mutant
3. Valid, but not Useful Mutant
4. Useful Mutant

Sample Solution

1. **Equivalent Mutant - SES (end block shift) - Result will be the same.**

```
public void bSearch(int[] A, int value, int start, int 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;
}
```

2. **Invalid Mutant - SDL (statement deletion) - Will not compile.**

```
public void bSearch(int[] A, int value, int start, int 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;
        }
}
```

3. **Valid, but not Useful - ROR (relational operator replacement) - will almost always fail if method is called correctly (as long as end!=start).**

```
public void bSearch(int[] A, int value, int start, int 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;
        }

}
```

4. **Useful Mutant - CRP (constant-for-constant replacement) - Will not always fail. Requires input that triggers that specific else if, and may still return the right result as long as it doesn't skip the correct entry.**

```
public void bSearch(int[] A, int value, int start, int 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;
        }

}
```

Question 12 (Automated Test Generation)

Metaheuristic search techniques can be divided into local and global search techniques.

1. Define what a “local” search and a “global” search is.
2. Contrast the two approaches. What are the strengths and weaknesses of each?
3. Choose one search algorithm and briefly explain how it works. State whether it is a global or local search, and explain why it belongs to that category.

Sample Solution

1. **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).**
2. **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.**
3. **An example of a local search is Simulated Annealing. Initially, a solution is generated at random. Then, during each round of the search, a random neighboring solution is picked (created by making a small change to the current solution). If that neighbor is better, it becomes the new solution. If it is worse or identically good, at a certain probability, that solution will become the new solution anyways. This probability is based partially on how many rounds the search has progressed through. At earlier rounds, the search is more likely to accept a worse solution to avoid getting stuck in a local maxima. Over time, it will be more likely to reject worse solutions. This is a local search because it manipulates one solution at a time and focuses on the local neighborhood when making changes.**

Question 13 (Finite State Verification)

Suppose that finite state verification of an abstract model of some software exposes a counter-example to a property that is expected to hold true for the system. This means that the model can be shown to not satisfy the property.

Briefly describe what follow-up actions would you take, and why you would take them?

Sample Solution

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 model code, bad assumptions, incorrect interpretation of requirements).**
- **There is an issue with the property. The property may not say what you intended it to say. It can be difficult to formulate a property in temporal logic.**
- **There is an issue with your requirements. The requirement may be incorrect, unclear, or incomplete.**

The action you take depends on which of the above is true. You should look at each angle, and find the source of the problem. If the model is incorrect, you should locate and correct the fault. If the property is incorrect, it should be reformulated. If the requirement is incorrect, it should be reformulated - then the property must also be rewritten to match. Fixing the requirement may also require updating the model as well or updating related requirements.

Question 14 (Finite State Verification)

Temporal Operators: A quick reference list. p is a Boolean predicate or atomic variable.

- G p: p holds globally at every state on the path from now until the end
- F p: p holds at some future state on the path (but not all future states)
- X p: p holds at the next 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 reaching out from a state, used in CTL as a modifier for the above properties (for example, AG p)
- E: for one or more paths reaching out from a state (but not all), used in CTL as a modifier for the above properties (for example, EF p)

An LTL example:

- G (SEND -> F (RECEIVED))
- It is always true (G), that if property SEND is true, then at some point in the future (F), property RECEIVED will become true.

A CTL example:

- EG (WIND -> AF (RAIN))
- There is a potential future where it is a certainty (EG) that - if there is wind (property WIND is true) - it will always be followed eventually (AF) by rain (RAIN is true).

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”.

   A trap property is when you write a normal property that is expected to hold, then you negate it (saying that the property will NOT be true). The verification framework will then produce a counter-example indicating that the property actually can be met - including a concrete set of input steps that will lead to the property being true.

Sample Solution

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 15 (Finite State Verification)

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.

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.

Sample Solution

CTL:

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

LTL:

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)))**