DIT636 / DAT560 - Practice Examination

There are a total of 14 questions on the practice exam (**there will be fewer on the real exam - we gave you some extra questions to study with**). On all essay questions, you will receive points based on the quality of the answer, not the quantity.

Question 1 (Warm Up) - 10 Points

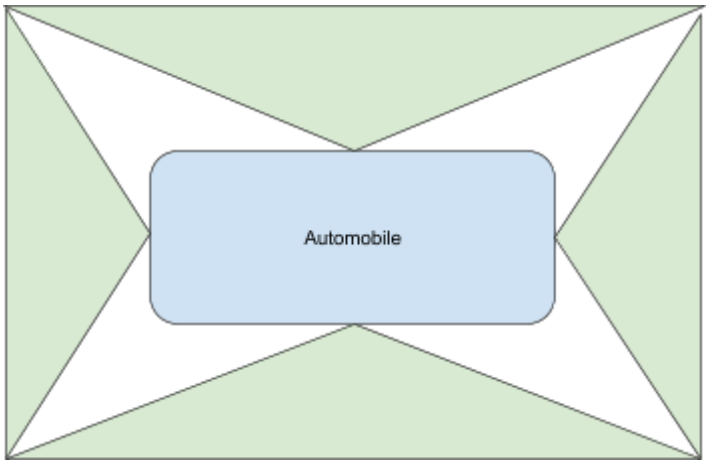
Multiple solutions may apply. Select all that are applicable. True/False worth 1 point each, multiple choice worth 2 points each.

1. For the expression `(a || !c) || (a && b)`, the test suite **(a, b, c) = {(T, F, T), (F, T, T), (T, T, T), (F, F, F)}** provides:
    a. Path Coverage
    b. Decision Coverage
    c. Basic Condition Coverage
    d. Compound Condition Coverage
2. During exploratory testing, the couch potato tour recommends running the system for a long period of time (e.g., overnight) to see if there are failures that emerge over time.
    a. True
    b. False
3. A liveness property is a property that should hold over a path of unknown length.
    a. True
    b. False
4. Mutation is considered useful because a sequence of small code changes can approximately model a larger fault in the program.
    a. True
    b. False
5. You have designed the software for an ATM to meet the following requirement: “If the amount of remaining money in the machine cannot be calculated, then display an error screen and prevent users from performing any additional interactions until the functionality is restored.” Which type of property is this?
    a. Correctness
    b. Reliability
    c. Robustness
6. An invalid mutant is one that is detected by many test cases.
    a. True
    b. False
7. You are designing a pedestrian detection system for a vehicle. Which **one** of the following quality attributes would be of most importance to you?
    a. Reliability
    b. Performance
    c. Scalability

    Briefly (1-2 sentences), explain why this is the most important.

Question 2 (Quality Scenarios) - 10 Points

Consider a camera-based object detection system in an automobile. This system uses a set of four cameras to capture the area around a vehicle (the green shaded areas in the image below):



This system offers the following functionality:

- Detects lanes on the road, warning the driver if the car crosses any lines that should not be crossed.
- Detects objects in the roadway (e.g., pedestrians or obstacles) that should not be collided with, and warns the driver if they are in the path of the vehicle.
- Warns the driver if there are conditions where the system cannot make accurate predictions. For example, if weather conditions cause a sharp reduction in image quality or if the camera has been covered.

Create one **reliability** and one **availability** scenario for this system, with a Description, System State, Environment State, External Stimulus, Required System Response, and Response Measure for each.

Sample Solution

Reliability:

Description: The system shall be able to reliably identify objects in the roadway, even when there is heavy snowfall.
System State: The system is operating under normal conditions (no recent failures).
Environment State: All cameras are functioning as expected, returning readings every 1ms. There is heavy snowfall (15cm accumulation in the past 12 hours, with more actively falling).
External Stimulus: An animal enters the road in front of the vehicle.
Required System Response: The object detection system detects the animal in one or more of the camera vision areas and issues a warning to the driver.
Response Measure: Even under the current weather conditions, we expect fewer than 2 out of every 10000 requests to fail (POFOD).

Availability:

Description: If the snowfall is sufficiently bad to prevent the cameras from working, the system shall warn the driver of degraded capabilities.
System State: The system is operating under normal conditions (no recent failures).
Environment State: All cameras are functioning as expected, returning readings every 1ms. There is extremely heavy snowfall (25cm accumulation in the past 12 hours, with the quantity falling steadily increasing).
External Stimulus: The camera on the right-hand side of the vehicle has been covered in snow.
Required System Response: The system shall detect that the sensor is failing to send valid data (i.e., no image or an image with no usable data). After detecting that the sensor is no longer working as expected, it shall issue a warning on the driver's heads-up display that the system cannot reliably detect obstacles under the current conditions. The system shall continue to monitor the camera data. Once the camera resumes sending usable image data, then the warning shall be disabled.
Response Measure: The system shall detect the failure within 2ms in 97% of cases, and within 4ms in 99.99% of cases. The system shall then issue the warning within 1 additional ms in 95% of cases and 2ms in 99% of cases. Once the camera is working again, the system shall remove the warning within 1ms in 95% of cases and 2ms in 99% of cases.

As with assignment 1, we expect a lot of variance here.Important factors:

- **Make sure you understand the difference between reliability and availability. Availability scenarios relate to situations where there has been a failure and assess the ability of the system to detect, mitigate, and/or recover from the failure. Reliability scenarios measure how often a function is allowed to fail.**
- **Make sure response measures are appropriate.**
- **Time measures should offer a normal and worst-case time.**

5 points per scenario, 1 point per requested item in each scenario.

Question 3 (Testing Concepts) - 8 Points

- Define system (integration) testing and exploratory testing.
- Explain how systems are tested at each stage.
- Explain how the two stages differ.
- Explain types of faults that may be more likely to be exposed by each of the two stages.

Sample Solution

During system testing, a system as a whole is tested through a defined interface (e.g., an API, a CLI, or a GUI). Testing during this stage is largely automated, using code-based test cases. The tests are generally created based on the requirements, but could also target code coverage (although this is less common than at the unit testing level). Generally, the system will have already been through unit testing, so most faults exposed at this stage are interaction faults (e.g., two code units do not coordinate properly). This could be due to a mistake in understanding how to call one of the units, or a fault in the code performing the calls and coordinating the two units.

Exploratory testing is also a practice where systems as a whole are tested through a defined interface. However, in contrast to system testing, the testing is generally done manually, by a person using the system. Exploratory testing is generally conducted in time-bound sessions, and guided by tours - recommendations of actions to try. In exploratory testing, test cases are not designed beforehand. Rather, testers design and execute tests concurrently, based on the observations they have made up to that point in the session. This stage can expose interaction faults or general faults in the software, like system testing, but the practice is well-suited to discovering issues that affect the usability of the software or a user's perception of the software (e.g., graphical glitches, painfully slow performance, cases where a function is difficult to learn how to use, etc.)

Make sure these key points are captured, that you understand the two concepts, that you can contrast these concepts. -1 point per mistake or important omission.

Question 4 (Test Design) - 12 Points

Recall the object discussion system discussed in Question 2.

Internally, the following method is invoked each time that the sensor data is refreshed:

```
public String[] analyzeObstacles (List<Float> cameraData)
```

The input parameter `cameraData` represents a list of zero or more potential obstacles detected by that camera. Each Float represents the distance that a detected obstacle is from the camera.

For example, if:
`System.out.println(cameraData.get(0));`
prints “1.45” to the screen, this indicates that the camera has detected an object (`cameraData.get(0)`) that is 1.45 meters from that camera.

This method should analyze the camera data and return a String array, where each item is a warning for the driver. The warnings should include:

- Any detected obstacle that is within 2 meters of the camera.
- Any situation where an obstruction may be blocking the camera (indicated by a detected object that is listed as 0.00 meters from the camera).

Perform functional test design for this method.

1. Identify choices (controllable aspects that can be varied when testing)
2. For each choice, identify representative input values.
3. For each value, apply constraints (IF, ERROR, SINGLE) if they make sense.

You do not need to create test specifications or concrete test cases. For invalid input, **do not** just write “invalid” - be specific. If you wish to make any additional assumptions about the functionality of this method, state them in your answer.

Grading Advice

Like with Assignment 2, there can be some variance here in how you split up the choices. We are looking for a range of outcomes being covered, and will make sure error handling is covered.

A sample solution:

Choice: Number of items in cameraData

- **0**
- **1**
- **10 (i.e., “typical”)**
- **10000 (i.e., “large set to test performance”) [single]**

Choice: Number of obstacles that could cause a collision (< 2 meters)

- **0**
- **1 [if number of items > 0]**
- **2+ [if number of items > 1]**

Choice: Is there a potential obstruction? (0.00 meters)

- **Yes [if number of items > 0]**
- **No**

Choice: Are there items in cameraData that could cause an error?

- **No**
- **At least one null value [error] [if number of items > 0]**
- **At least one negative value [error] [if number of items > 0]**

In general, -1-2 points per mistake, -1-2 points per missing piece of functionality or missing constraint.

Question 5 (Test Design)

Consider a personnel management program that offers an API where, among other functions, a user can apply for vacation time:

```
public boolean applyForVacation (String userID, String startingDate,
String endingDate)
```

A user ID is a string in the format “firstname.lastname”, e.g., “gregory.gay”.
The two dates are strings in the format “YYYY-DD-MM”.

The function returns TRUE if the user was able to successfully apply for the vacation time. It returns FALSE if not. An exception can also be thrown if there is an error.

This function connects to a user database. Each user has the following relevant items stored in their database entry:

- User ID
- Quantity of remaining vacation days for the user
- An array containing already-scheduled vacation dates (as starting and ending date pairs)
- An array containing dates where vacation cannot be applied for (e.g., important meetings).

Perform functional test design for this function.

1. Identify choices (controllable aspects that can be varied when testing)
2. For each choice, identify representative input values.
3. For each value, apply constraints (IF, ERROR, SINGLE) if they make sense.

You do not need to create test specifications or concrete test cases. For invalid input, **do not** just write “invalid” - be specific. If you wish to make any additional assumptions about the functionality of this method, state them in your answer.

Sample Solution

Note that your solution may not match this exactly, but should contain but should be detailed and account for normal and error scenarios.

- **Choice: Value of userID**
  - **Existing user**
  - **Non-existing user [error]**
  - **Null [error]**
  - **Malformed user ID (not in format “firstname.lastname”) [error]**
- **Choice: Value of starting date**
  - **Valid date**
  - **Date before the current date [error]**
  - **Current date [single]**
  - **Null [error]**
  - **Malformed date (not in format “YYYY-MM-DD”) [error]**
- **Choice: Value of ending date**
  - **Valid date**
  - **Date before the current date [error]**
  - **Current date [single]**
  - **Date before the starting date [error]**
  - **Date same as the starting date [single]**
  - **Null [error]**
  - **Malformed date (not in format “YYYY-MM-DD”) [error]**
- **Choice: Remaining vacation time for the userID**
  (Note: We are assuming the database schema prevents storing malformed/invalid values)
  - **0 days remaining**
  - **1 day remaining, 1 day applied for [single]**
  - **Number of days remaining < number applied for**
  - **Number of days remaining = number applied for [single]**
  - **Number of days remaining > number applied for**
  - **User does not exist [if “value of userID” is “non-existing user”]**
- **Choice: Conflicts with vacation time**
  (Note: We are assuming the database schema prevents storing malformed/invalid date ranges)
  - **No conflicts with already-scheduled vacation or banned dates**
  - **Banned date(s) fall within the starting and ending dates applied for**
  - **Starting date falls within already-scheduled vacation time**
  - **Ending date falls within already-scheduled vacation time**
  - **Already-scheduled vacation time falls within starting and ending dates applied for**
  - **The starting and ending dates fall within already-scheduled vacation time**
  - **User does not exist [if “value of userID” is “non-existing user”]**

Question 6 (Exploratory Testing) - 8 Points

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 **other than the supermodel tour**.
2. Consider a web-based discussion forum. This forum offers the following functionality:
   - Users can register for an account, using their e-mail address. They can choose a display name (must be unique) and a password for their account.
   - Once registered, users can log into their account.
   - Some users, called “administrators”, have additional privileges.
   - Administrators can create, edit the name and description of, and delete “boards”.
   - A “board” is an area where users can post topics related to the subject of that board.
   - Users can also reply to topics.
   - Users can edit their own posts in topics.
   - Administrators can also edit or delete posts in topics.

   Describe three distinct sequences of interactions with one or more functions of this system you would explore during exploratory testing of this system, based on the tour you described above. Explain the interactions you would take when executing that sequence, and why those actions fulfill the goals of that tour. These sequences should be different from each other (i.e., limited overlap).

Sample Solution

Part 1: Fed-Ex Tour

FedEx is a company that does package-delivery. They pick up packages, move them around their various distribution centers, and send them to their final destination. For this tour, instead of packages moving around the planet through the FedEx system, think of data moving through the software. The data starts its life as input and gets stored internally in variables and data structures where it is manipulated, modified, and used in computation. Finally, much of this data is finally "delivered" as output to some user or destination - some other part of the system or an external system.

During this tour, a tester must concentrate on this data. Try to identify inputs that are stored and "follow" them around the software. For example, when a mailing address is entered into a shopping site, how is that address used? What features consume this address? If it is used as a billing address, make sure you exercise that feature. If it is used as a shipping address, make sure you use that feature. If it can be updated, update it. Does it ever get printed or purged or processed? Try to find every feature that touches the data so that, just as FedEx handles their packages, you are involved in every stage of the data's life cycle.

Part 2:

Sequence 1:

- **Register for a new account.**
- **Log in.**
- **Log out.**
- **Attempt to register a new account with the same e-mail address or display name.**
- **Verify that the new account request was rejected.**
- **Log into the account created in step 1, verify that you could log in successfully.**

This sequence tests account creation. It ensures that you can create an account, then log into it. It also ensures that a duplicate account cannot be created, and that the attempt to register a new duplicate account does not interfere with the original account.

Sequence 2:

- **Log in as an administrator.**
- **Post a topic to a board.**
- **Verify that the topic was created.**
- **Edit the post.**
- **Check that the edits are reflected after saving them.**
- **Add a reply to the topic.**
- **Verify that the reply was added.**
- **Delete the reply.**
- **Verify that the reply was deleted**

This sequence checks that posts can be created, edited, and deleted.

Sequence 3:

- **Log in as an administrator.**

- **Create a new board.**
- **Verify that it was created.**
- **Edit the name and description of the board.**
- **Verify that changes are saved successfully.**
- **Delete the board.**
- **Verify that the board was deleted.**

This sequence checks that boards can be created, edited, and deleted.

These sequences relate to the FedEx tour because they all relate to the creation, manipulation, and deletion of data items. In particular, because multiple forms of data manipulation are performed in the same interaction sequence, we further ensure the system is robust to combining multiple forms of data manipulation in rapid sequence.

In Part 1, you should accurately describe one of the tours. In Part 2, the interaction sequences should clearly relate to the tour. There should be an explicit statement of how the interactions relate to the tour and fulfill its goals.

Four points for part 1, four for part 2. -1 point per mistake.

Question 7 (Unit Testing) - 9 Points

Consider the obstacle detection method that you developed test specifications for in Question 4:

```
public String[] analyzeObstacles (List<Float> cameraData)
```

Based on your test specifications, write three JUnit-format test cases:

1. Create one test case that checks a normal usage of the method, with at least one detected obstacle.
2. Create one test case that checks a normal usage of the method, with a potential obstruction.
3. Create one test case reflecting an error-handling scenario (an exception is thrown).

Sample Solution

Check for proper JUnit formatting and naming conventions, check that you match the expected input and output format of the method, check that you use assertions properly when an exception is expected to be thrown.

Java syntax does not need to be perfect (this is a paper exam), but should be close.

3 points per test, -1 point per mistake

Sample solution:

```
@Test
public void testDetectedObstacle() {
    List<Float> input = new ArraytList<Float>();
    input.add(0.50);
    String[] expected = new String[1];
    expected[0] = “Obstacle detected at 0.50 meters”;
    String[] output = analyzeObjects(input);
    assertArrayEquals(output, expected);
    assertTrue(output.length == 1);
}

@Test
public void testDetectedObstruction() {
    List<Float> input = new ArraytList<Float>();
    input.add(0.00);
    String[] expected = new String[1];
    expected[0] = “There is a potential obstruction blocking the camera”;
    String[] output = analyzeObjects(input);
    assertArrayEquals(output, expected);
    assertTrue(output.length == 1);
}

@Test
public void testNullContents() {
    List<Float> input = new ArraytList<Float>();
    input.add(null);

    assertThrows(NullPointerException.class, () -> {
        analyzeObjects(input);
    });
}
```

Question 8 (Structural Testing)

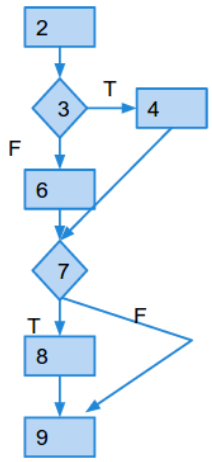
For the following function,

a. Draw the control flow graph for the program.
b. Develop test input that will provide statement coverage. For each test, note which lines are covered.
c. Develop test input that will provide branch coverage. For each test, note which branches are covered. You may reuse input from the previous problem.
d. Develop test input that will provide path coverage. For each test, note which paths are covered. You may reuse input from the previous problem.
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.  }
```

(Just including test input is sufficient - you do not need to write 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) covers lines 2, 3, 4, 7, 8, 9
(2, 3, 4) covers lines 2, 3, 6, 7, 8, 9

c) (3, 2, 4) covers branches 3T, 7T
(3, 4, 1) covers branches 3F, 7F

d) (4, 2, 5) covers path 3T, 7T
(4, 2, 1) covers path 3T, 7F
(2, 3, 4) covers path 3F, 7T
(2, 3, 1) covers path 3F, 7F

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 9 (Structural Testing) - 16 Points

This function takes an array and string. It will then remove the letters in the string from the array, one at a time, and return the array. If a letter is in the array multiple times, it will only be removed the number of times that it appears in the string (e.g., if “b” is in the array twice, and the string is “big”, then “b” will only be removed one time).

```
1.  public String[] removeLetters(String[] letters, String word) {
2.      int lettersRemoved = 0;
3.      while(word.length() != 0){
4.          for(int i = 0; i < letters.length; i++){
5.              if(letters[i].equals(word.substring(0, 1))){
6.                  letters[i] = "";
7.                  lettersRemoved++;
8.                  break;
9.              }
10.          }
11.          word = word.substring(1);
12.      }
13.      int idx = 0;
14.      String[] output = new String[letters.length - lettersRemoved];
15.      for(int i = 0; i < letters.length; i++){
16.           if(!letters[i].equals("")){
17.               output[idx] = letters[i];
18.               idx++;
19.           }
20.      }
21.      return output;
22.  }
```

For example:

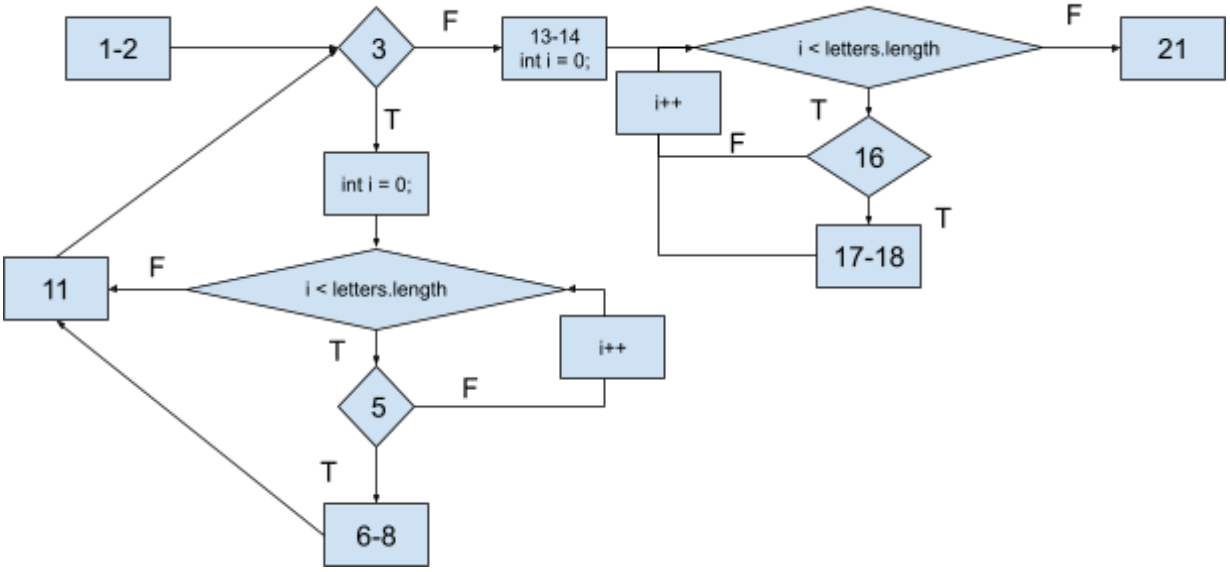
- removeLetters(["s", "t", "r", "i", "n", "g", "w"], "string") → ["w"]
- removeLetters(["b", "b", "l", "l", "g", "n", "o", "a", "w"], "balloon") → ["b", "g", "w"]
- removeLetters(["d", "b", "t", "e", "a", "i"], "edabit") → [ ]

1. Draw the control-flow graph for this function. You may refer to line numbers instead of writing the full code.
2. Identify test input that will provide statement and branch coverage. You do not need to create full unit tests, just supply input for the function.

   For each input, list the line numbers of the statements covered as well as the specific branches covered (use the line number and T/F, i.e., “3-T” for the true branch of line 3). Do not use the test input from the examples above. Come up with your own input.

Sample Solution

The control flow graph for the method is:



removeLetters(["d", "i", "t"], "dir") → ["t"]
Covers:

- **1-2, 6-8, 11, 13-14, 17-18, 21**
- **3T, 3F**
- **4T, 4F**
- **5T, 5F**
- **15T, 15F**
- **16T, 16F**

Check that all lines and branches are covered.
Make sure you do not reuse the sample inputs.

5 points for the CFG, -1 point per mistake
10 points for Part 2. -1 point per uncovered or unstated line or branch. -8 points if you do not state what lines/branches are covered (more if the tests clearly do not achieve coverage).

Question 10 (Data Flow Testing) - 12 Points

Using the same code from Question 9:

1. Identify the def-use pairs for all variables.
2. Identify test input that achieves all def-use pairs coverage.

Note: You may treat arrays as a single variable for purposes of defining DU pairs. This means that a definition to **`letters[i]`** or to array **`letters`** are both definitions of the same variable, and references to **`letters[i]`** or **`letters.length`** are both uses of the same variable.

Sample Solution

Variable	**D-U Pairs**
letters	(1, 4), (1, 5), (6, 4), (6, 5), (6, 6), (1, 14), (6, 14), (1, 15), (6, 15), (1, 16), (6, 16), (1, 17), (6, 17)
word	(1, 3), (1, 5), (1, 11), (11, 5), (11, 11), (11,3)
lettersRemoved	(2, 7), (7, 7), (2, 14), (7, 14)
i	(4, 4), (4, 5), (4, 6), (15, 15), (15, 16), (15, 17)
idx	(13, 17), (13, 18), (18, 18), (18, 17)
output	(14, 21), (17,21)

Your input needs to cover these paths:

- **Will need input that either skips the first loop or where no letters are removed from the array.**
- **Will need input that skips the second loop or where the output is empty (no letters are left in the array).**
- **Loops may need to cycle two or more times to ensure new definitions are used.**

This input covers these situations:

removeLetters(["d", "i", "t", “y”], "dir") → ["t", “y”]

Variable	**New D-U Pairs Covered**
letters	(1, 4), (1, 5), (6, 4), (6, 5), (6, 6), (6, 14), (6, 15), (6, 16), (6, 17)
word	(1, 3), (1, 5), (1, 11), (11, 3), (11, 5), (11, 11)
lettersRemoved	(2, 7), (7, 7), (7, 14)
i	(4, 5), (4, 6), (4, 4), (15, 16), (15, 15), (15, 17)
idx	(13, 17), (13, 18), (18, 18), (18, 17)
output	(17, 21)

removeLetters(["d"], "x") → ["d"]

Variable	New D-U Pairs Covered
letters	(1, 14), (1, 15), (1, 16), (1, 17)
lettersRemoved	(2, 14)

removeLetters(["d"], "d") → [ ]

Variable	New D-U Pairs Covered
output	(14, 21)

6 points for Part 1 (D-U Pairs), 6 points for part 2.
-1 point per variable with a missing DU pair or per missing variable
-1 point for DU pair not covered by a test case
-4 points for not stating which DU pairs are covered (more if the tests clearly can't cover all DU pairs)

Question 11 (Data Flow Testing)

The following function returns true if you can partition an array into one element and the rest, such that this element is equal to the product of all other elements excluding itself.
For example:

- **`canPartition([2, 8, 4, 1])`** returns true (8 = 2 * 4 * 1)
- **`canPartition([-1, -10, 1, -2, 20])`** returns false.
- **`canPartition([-1, -20, 5, -1, -2, 2])`** returns true (-20 = -1 * 5 * -1 * -2 * 2)

```
1. public static boolean canPartition(int[] arr) {
2.       Arrays.sort(arr);
3.       int product = 1;
4.       if ((Math.abs(arr[0]) >= arr[arr.length-1]) || arr[0] == 0) {
5.           for (int i = 1; i < arr.length; i++){
6.               product *= arr[i];
7.           }
8.           return arr[0] == product;
9.       } else{
10.         for (int i = 0; i < arr.length-1; i++){
11.             product *= arr[i];
12.         }
13.         return arr[arr.length-1] == product;
14.      }
15.  }
```

1. Identify the def-use pairs for all variables.
2. Identify test input that achieves all def-use pairs coverage.

Note: You may treat arrays as a single variable for purposes of defining DU pairs. This means that a definition to **`arr[0]`** or to array **`arr`** are both definitions of the same variable, and references to **`arr[0]`** or **`arr.length`** are both uses of the same variable.

Sample Solution

1. DU Pairs

arr	(1, 2), (2, 4), (2, 5), (2, 6), (2, 8), (2, 10), (2, 11), (2, 13)
product	(3, 6), (6, 6), (3, 8), (6, 8), (3, 11), (11, 11), (11, 13)
i	(5, 5), (5, 6), (10, 10), (10, 11)

2. Test Input

Input	**Additional DU Pairs Covered**
`[2, 8, 4, 1]`	arr: (1, 2), (2, 4), (2, 10), (2, 11), (2, 13) product: (3, 11), (11, 11), (11, 13) i: (10, 10), (10, 11)
`[-1, -10, 0, 10]`	arr: (2, 5), (2, 6), (2, 8) product: (3, 6), (6, 6), (6, 8) i: (5, 5), (5, 6)
`[0]`	product: (3, 8)

Question 12 (Mutation Testing) - 15 Points

This function takes in an integer, and inserts duplicate digits on both sides of all digits which appear in a group of one.

```
1. public static long lonelyNumbers(int n) {
2.     String s = " " + n + " ";
3.     long a = 0;
4.     StringBuilder sb = new StringBuilder();
5.     for(int i = 1; i < s.length() - 1; i++){
6.         if(s.charAt(i) == s.charAt(i - 1) || s.charAt(i) ==
               s.charAt(i + 1)){
7.             sb.append(s.charAt(i));
8.         } else
9.             sb.append("" + s.charAt(i) + s.charAt(i) + s.charAt(i));
10.     }
11.     a = Long.parseLong(sb.toString().trim());
12.     return a;
13.   }
```

For example:

- `lonelyNumbers(4666)` → `444666`
- `lonelyNumbers(544)` → `55544`
- `lonelyNumbers(123)` → `111222333`
- `lonelyNumbers(33)` → `33`

Answer the following three questions for **each** of the following mutation operators:

- Relational operator replacement (ror)
- Arithmetic operator replacement (aor) (including short-cut operators)
- Constant for constant replacement (crp)

1. Identify all lines that can be mutated using that operator.
2. Choose **one** line that can be mutated by that operator and create **one** non-equivalent mutant for that line.
3. For that mutant, identify test input that would detect the mutant. Show how the output (return value of the method) differs from that of the original program.

Sample Solution

ROR: 5, 6
Line 6: `if(s.charAt(i) != s.charAt(i - 1) ...`

`lonelyNumbers(4666)`
Original: returns 444666
Mutant: returns 466666
The mutation does not add the duplicate “4”s that should be present. It also adds additional “6”s at the end.

AOR: 2, 5, 6, 9
Line 6: `if(s.charAt(i) == s.charAt(i + 1) ...`

`lonelyNumbers(4666)`
Original: returns 444666
Mutant: returns 44466666
The mutation adds additional “6”s at the end.

CRP: 2, 3, 5, 6, 9
Line 6: `if(s.charAt(i) == s.charAt(i - 0) ...`

`lonelyNumbers(4666)`
Original: returns 444666
Mutant: returns 4666
The mutation does not add the duplicate “4”s that should be present.

Make sure that you use the mutation operator correctly. Take a quick look and make sure that it is non-equivalent. Ok if it is invalid (does not compile).
Make sure you state the return value of the original and the mutated version.
Make sure the explanation is clear about why the mutant is detected.

5 points per mutation operator (1 point for part 1, 2 for part 2, 2 for part 3).
Points off for mistakes.

Question 13 (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 (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 (EF p)

An LTL example:

- G ((MESSAGE_STATUS = SENT) -> F (MESSAGE_STATUS = RECEIVED))
- It is always true (G), that if the message is sent, then at some point after it is sent (F), the message will be received.
    - More simply: A sent message will always be received eventually.

A CTL example:

- EG ((WEATHER = WIND) -> AF (WEATHER = RAIN))
- There is a potential future where it is a certainty (EG) that - if there is wind - it will always be followed eventually (AF) by rain.
    - More simply: At a certain probability, wind will inevitably lead to eventual rain. (However, that probability is not 100%)

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, but we do not know how long it will take.

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 (again, we do not know how long it will take).

   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 14 (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)))**