CMPE3008 SOFTWARE ENGINEERING TESTING GROUP ASSIGNMENT

By The Predictive Pioneers

ELIZABETH RUSSELL, 17904982
MITCHELL PONTAGUE, 19126924
RYAN MACKINTOSH, 21171466

Semester 1 2024

Contents

Chosen Code Project Details

Source Location: sorting-algorithms by murraco

Date Accessed: 2 May 2024

Chosen Methods

We chose two sorting algorithms from the "sorting-algorithms by murraco" repository: Bubble Sort and Insertion Sort. We chose to focus on these algorithms because they both contain nested loops, and are complex enough to demonstrate the requirements of the task without being too simple or too complex.

Method 1: Bubble Sort

Source File Location: src/main/java/murraco/BubbleSort.java

Source Code:

```java
package murraco;

public class BubbleSort {

  // Time complexity: average O(n^2) and best O(n) - Space complexity: O(1)
  public static <T extends Comparable<T>> void bubbleSort(T[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
      for (int j = 1; j < arr.length; j++) {
        if (arr[j].compareTo(arr[j - 1]) < 0) {
          T temp = arr[j];
          arr[j] = arr[j - 1];
          arr[j - 1] = temp;
        }
      }
    }
  }
}
```

Test File Location: src/test/java/murraco/SortingAlgorithmsTest.java

Test Code:

```java
package murraco;

import static org.junit.Assert.assertEquals;

import java.util.Arrays;

import org.junit.Test;

import murraco.BubbleSort;
import murraco.Heapsort;
import murraco.InsertionSort;
import murraco.MergeSort;
import murraco.Quicksort;
import murraco.SelectionSort;

public class SortingAlgorithmsTest {

  @Test
  public void testBubbleSort() {
      final Integer[] data = {4, 3, 0, 11, 7, 5, 15, 12, 99, 1};
      BubbleSort.bubbleSort(data);
      assertEquals("[0, 1, 3, 4, 5, 7, 11, 12, 15, 99]", Arrays.toString(data));
  }

  @Test
  public void testInsertionSort() {
      final Integer[] data = {4, 3, 0, 11, 7, 5, 15, 12, 99, 1};
      InsertionSort.insertionSort(data);
      assertEquals("[0, 1, 3, 4, 5, 7, 11, 12, 15, 99]", Arrays.toString(data));
  }

  @Test
  public void testSelectionSort() {
      final Integer[] data = {4, 3, 0, 11, 7, 5, 15, 12, 99, 1};
      SelectionSort.selectionSort(data);
      assertEquals("[0, 1, 3, 4, 5, 7, 11, 12, 15, 99]", Arrays.toString(data));
  }

  @Test
  public void testMergeSort() {
      final Integer[] data = {4, 3, 0, 11, 7, 5, 15, 12, 99, 1};
      MergeSort.mergeSort(data);
      assertEquals("[0, 1, 3, 4, 5, 7, 11, 12, 15, 99]", Arrays.toString(data));
  }

  @Test
  public void testHeapsort() {
      final Integer[] data = {4, 3, 0, 11, 7, 5, 15, 12, 99, 1};
      Heapsort.heapSort(data);
      assertEquals("[0, 1, 3, 4, 5, 7, 11, 12, 15, 99]", Arrays.toString(data));
  }

  @Test
  public void testQuicksort() {
      final Integer[] data = {4, 3, 0, 11, 7, 5, 15, 12, 99, 1};
      Quicksort.quickSort(data);
      assertEquals("[0, 1, 3, 4, 5, 7, 11, 12, 15, 99]", Arrays.toString(data));
  }
}
```

Method 2: Insertion Sort

Source File Location: src/main/java/murraco/InsertionSort.java

```java
package murraco;

public class InsertionSort {

  // Time complexity: average O(n^2) and best O(n) - Space complexity: O(1)
  public static <T extends Comparable<T>> void insertionSort(T[] arr) {
    for (int i = 0; i < arr.length; i++) {
      T temp = arr[i];
      int j = i;
      while (j > 0 && arr[j - 1].compareTo(temp) > 0) {
        arr[j] = arr[j - 1];
        j--;
      }
      arr[j] = temp;
    }
  }

}
```

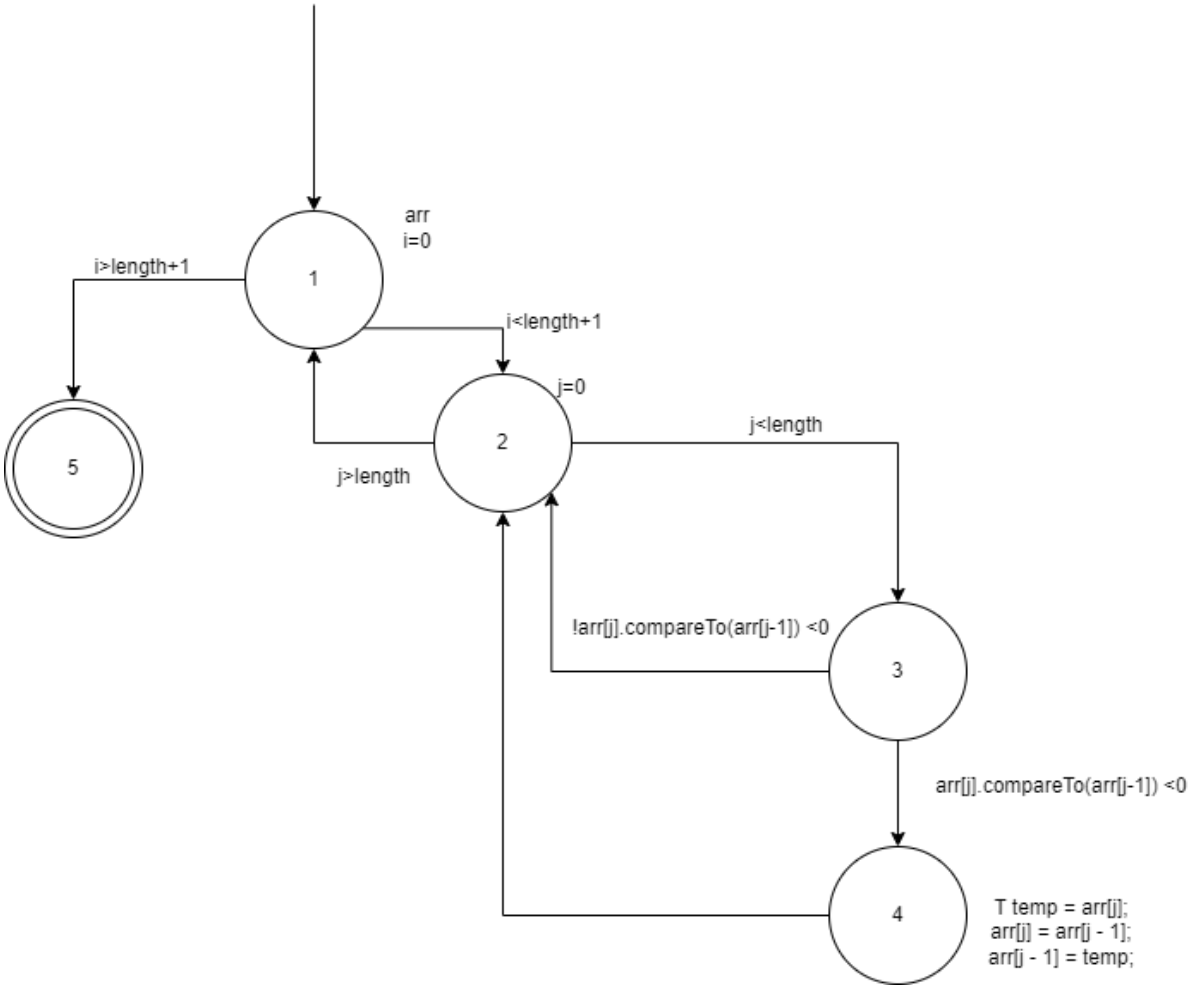
Test File Location: src/test/java/murraco/SortingAlgorithmsTest.java

```java
package murraco;

import static org.junit.Assert.assertEquals;

import java.util.Arrays;

import org.junit.Test;

import murraco.BubbleSort;
import murraco.Heapsort;
import murraco.InsertionSort;
import murraco.MergeSort;
import murraco.Quicksort;
import murraco.SelectionSort;

public class SortingAlgorithmsTest {

  @Test
  public void testBubbleSort() {
      final Integer[] data = {4, 3, 0, 11, 7, 5, 15, 12, 99, 1};
      BubbleSort.bubbleSort(data);
      assertEquals("[0, 1, 3, 4, 5, 7, 11, 12, 15, 99]", Arrays.toString(data));
  }

  @Test
  public void testInsertionSort() {
      final Integer[] data = {4, 3, 0, 11, 7, 5, 15, 12, 99, 1};
      InsertionSort.insertionSort(data);
      assertEquals("[0, 1, 3, 4, 5, 7, 11, 12, 15, 99]", Arrays.toString(data));
  }

  @Test
  public void testSelectionSort() {
      final Integer[] data = {4, 3, 0, 11, 7, 5, 15, 12, 99, 1};
      SelectionSort.selectionSort(data);
      assertEquals("[0, 1, 3, 4, 5, 7, 11, 12, 15, 99]", Arrays.toString(data));
  }

  @Test
  public void testMergeSort() {
      final Integer[] data = {4, 3, 0, 11, 7, 5, 15, 12, 99, 1};
      MergeSort.mergeSort(data);
      assertEquals("[0, 1, 3, 4, 5, 7, 11, 12, 15, 99]", Arrays.toString(data));
  }

  @Test
  public void testHeapsort() {
      final Integer[] data = {4, 3, 0, 11, 7, 5, 15, 12, 99, 1};
      Heapsort.heapSort(data);
      assertEquals("[0, 1, 3, 4, 5, 7, 11, 12, 15, 99]", Arrays.toString(data));
  }

  @Test
  public void testQuicksort() {
      final Integer[] data = {4, 3, 0, 11, 7, 5, 15, 12, 99, 1};
      Quicksort.quickSort(data);
      assertEquals("[0, 1, 3, 4, 5, 7, 11, 12, 15, 99]", Arrays.toString(data));
  }
}
```

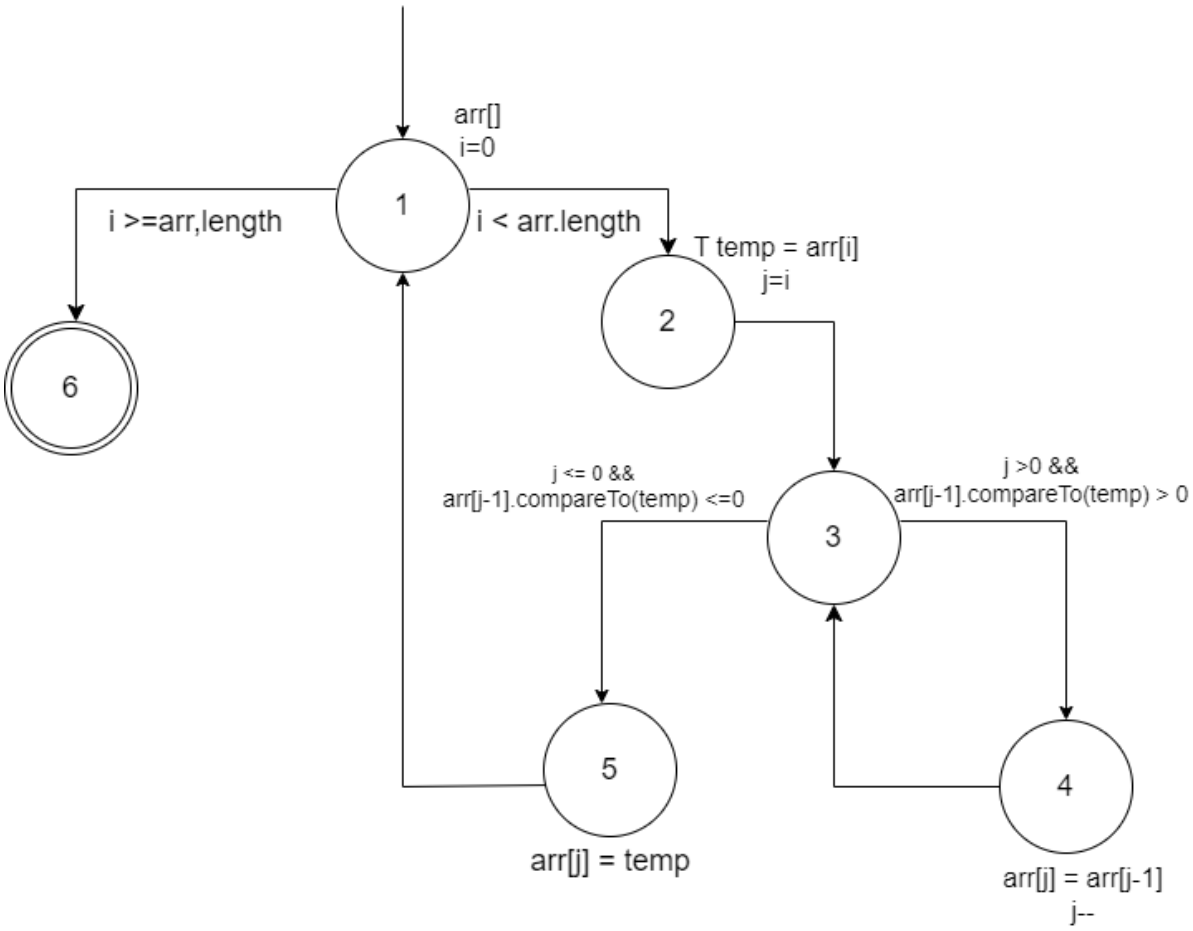
Coverage Analysis

Method Graphs

Graph for Method 1: BubbleSort



Graph for Method 2: InsertionSort



Prime Path Coverage

Method 1: BubbleSort Test Coverage Analysis

BubbleSort Test Requirements (TR)

BubbleSort Level 0 (L0)

- [1]
- [2]
- [5]
- [3]
- [4]

BubbleSort Level 1 (L1)

- [1,2]
- [1,5]
- [2,1]
- [2,3]
- [3,4]
- [3,2]
- [4,2]

BubbleSort Level 2 (L2)

- [1,2,1]
- [1,2,3]
- [1,5]
- [2,1,2]
- [2,1,5]
- [2,3,4]
- [2,3,2]
- [3,4,2]
- [3,2,1]
- [3,2,3]
- [4,2,1]
- [4,2,3]

BubbleSort Simple Paths

- [1,2]
- [1,5]
- [2,1]
- [2,3]
- [3,4]
- [3,2]
- [4,2]
- [1,2,1]
- [1,2,3]
- [2,1,2]
- [2,1,5]
- [2,3,4]
- [2,3,2]
- [3,4,2]
- [3,2,1]
- [3,2,3]
- [4,2,1]
- [4,2,3]
- [1,2,3,4]
- [2,3,4,2]
- [3,4,2,1]
- [3,4,2,3]
- [3,2,1,5]
- [4,2,1,5]
- [4,2,3,4]
- [3,4,2,1,5]

BubbleSort Prime Paths

- [3,4,2,1,5]
- [2,3,4,2]
- [1,2,3,4]
- [3,4,2,3]
- [4,2,3,4]
- [3,2,1,5]

- [2,1,2]
- [1,2,1]
- [2,3,2]
- [3,2,3]

BubbleSort Test Paths for Prime Path Coverage

Test Paths	Test Requirements
[1,2,1,2,1,2,3,4,2,1,5]	[3,4,2,1,5], [1,2,1], [2,1,2]
[1,2,3,4,2,3,4,2,3,2,1,5]	[3,2,1,5], [2,3,4,2], [1,2,3,4], [3,4,2,3], [4,2,3,4], [2,3,2]
[1,2,3,2,3,2,1,5]	[3,2,3]

Method 2: InsertionSort

InsertionSort Test Requirements (TR)

InsertionSort Level 0 (L0)

- [1]
- [2]
- [3]
- [4]
- [5]
- [6]

InsertionSort Level 1 (L1)

- [1,2]
- [2,3]
- [3,4]
- [4,3]
- [3,5]
- [5,1]
- [1,6]

InsertionSort Level 2 (L2)

- [1,2,3]
- [2,3,4]
- [2,3,5]
- [3,4,3]
- [4,3,4]
- [4,3,5]
- [3,5,1]
- [5,1,2]
- [5,1,6]
- [1,6]

InsertionSort Simple Paths

- [1,2]
- [2,3]
- [3,4]

- [4,3]
- [3,5]
- [5,1]
- [1,6]
- [1,2,3]
- [2,3,4]
- [2,3,5]
- [3,4,3]
- [4,3,4]
- [4,3,5]
- [3,5,1]
- [5,1,2]
- [5,1,6]
- [1,2,3,4]
- [1,2,3,5]
- [2,3,5,1]
- [4,3,5,1]
- [3,5,1,2]
- [3,5,1,6]
- [5,1,2,3]
- [1,2,3,5,1]
- [2,3,5,1,2]
- [2,3,5,1,6]
- [4,3,5,1,2]
- [4,3,5,1,6]
- [3,5,1,2,3]
- [5,1,2,3,4]
- [5,1,2,3,5]

InsertionSort Prime Paths

- [2,3,5,1,6]
- [4,3,5,1,2]
- [2,3,5,1,2]
- [1,2,3,5,1]

- [4,3,5,1,6]
- [5,1,2,3,5]
- [5,1,2,3,4]
- [3,5,1,2,3]
- [3,4,3]
- [4,3,4]

InsertionSort Test Paths for Prime Path Coverage

Test Paths	Test Requirements
[1,2,3,4,3,4,3,4,1,6]	[4,3,5,1,6] , [4,3,4]
[1,2,3,5,1,2,3,5,1,2,3,4,3,5,1,2,3,5,1,6]	[2,3,5,1,6],[4,3,5,1,2],[2,3,5,1,2],[1,2,3,5,1],[5,1,2,3,5],[5,1,2,3,4],[,3,5,1,2,3],[3,4,3]

Syntax-Based Testing Analysis

Mutants for Method 1: BubbleSort

```
public static <T extends Comparable<T>> void bubbleSort(T[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
Δ1  for (int i = 0; i < arr.length - 1; i+=2) {
      for (int j = 1; j < arr.length; j++) {
Δ2    for (int j = 1; j < arr.length; j+=2) {
        if (arr[j].compareTo(arr[j - 1]) < 0) {
 Δ3   if (arr[j].compareTo(arr[j - 1]) > 0) {
          T temp = arr[j];
          arr[j] = arr[j - 1];
          arr[j - 1] = temp;
 Δ4   T temp = arr[j - 1]; // Swap positions
          arr[j - 1] = arr[j];
          arr[j] = temp;
        }
      }
    }
  }
```

Mutant 1: Arithmetic Operator Replacement

Reachability:

The change to the increment in the outer for loop will be reached in all scenarios.

Infection:

The infection will always be present as the code will always reach the for loop and the value of i will be infected.

Propagation:

Its possible for the infection to be present without propagation being satisfied, these cases are when the passed array is empty, null, already sorted. Other cases as well may pass when the array is mostly sorted.

As an **example:**

[3,2,1,2,1,2,1,2,1,2,2,5,4] will weakly kill.

[10,9,8,7,6,5,4,3,2,1] will strongly kill the mutant.

Mutant 2: Arithmetic Operator Replacement

Reachability:

The mutant will be reached in the cases that the array is of two or greater elements which gets the code inside the outer for loop and into the inner loop.

Infection:

The infection will always be present as long as reachability is satisfied.

Propagation:

Similair to mutant one an empty array, array that is null and arrays of 1 will lead to no propagation. Other instances such as a already sorted array and arrays of the same value along with arrays that have duplicate values present can pass in some instances.

As an **example:**

[3,2,1] will weakly kill the mutant.

[1,2,1] will strongly kill the mutant.

Mutant 3: Relational Operator Replacement

Reachability:

The mutant will always be reached as long as the array is not null or empty.

Infection:

The infection is present when the value of arr[j] is greater then the value of arr[j-1] where the values are the same the infection is not present.

Propagation:

The array will now end up being in descending order instead of ascending order in any case where the array is not empty, null or of the same value.

As an **example:**

[1,1,1] will weakly kill the mutant.

[7,1,7,20,-5] will strongly kill the mutant.

Mutant 4: Statement Replacement

Reachability:

The conditions for the mutant to be reached are as follows; the array must consist of at least two elements that are not currently in order for the code to meet the mutant.

Infection:

No infection is present due to the fact that the mutant is a equivalent mutant.

Propagation:

This mutant is an equivalent mutant thus no propagation will occur.

Mutants for Method 2: InsertionSort

```
public class InsertionSort {

  public static <T extends Comparable<T>> void insertionSort(T[] arr) {
    for (int i = 0; i < arr.length; i++) {
  Δ4   //  for (int i = 0; i < arr.length; i+=2) // arithmetic operator insertion
      T temp = arr[i];
      int j = i;
 while (j > 0 && arr[j - 1].compareTo(temp) > 0) {
   Δ1   // while (j > 0 && arr[j - 1].compareTo(temp) > 0) // delete the condition
{
   Δ2   if (j == 1) { throw new RuntimeException("bomb trigger"); // Specific condition to
trigger the bomb
}
   Δ3   // while (j >= 0 && arr[j - 1].compareTo(temp) > 0) // arithmetic operator
insertion
{
        arr[j] = arr[j - 1];
        j--;
      }
      arr[j] = temp;
    }
  }

}
```

Mutant 1: Statement Deletion Operator

Reachability:

If the array has more than one element the code will enter the for loop and reach the mutant.

Infection:

Deletion of the while loop leads to ArrayIndexOutOfBoundsException, because without checking the condition j > 0, decrementing j will lead to trying to access: arr[-1], which is an invalid index.

Propagation:

The program crashed due to an unhandled condition: <<java.lang.ArrayIndexOutOfBoundsException: Index -1 out of bounds for length 10>>.

The mutant 1 results in ArrayIndexOutOfBoundsException because the loop may try to access arr[j-1]. When deleting the while loop, there is no boundary check from the deleted condition causing the j to go past index 0, leading to trying to access an index of array that doesn't exist. The test harness in the source code doesn't explicitly handle this exception. If mutants are active, the code will crash due to unhandled exceptions. The test case from the source doesn't assert or expect any exceptions.

Mutant 2: Bomb Statement Replacement

Reachability:

If condition j ==1 should be met for the bomb() simulation to be executed, if the second element is less than the first element and the array is not sorted.

Infection:

If condition j == 1 is met during sorting it will trigger (‘RuntimeException’).

Propagation:

The program crashed due to an unhandled condition: java.lang.RuntimeException: bomb trigger.

The test array is: << final Integer[] data = { 4, 3, 0, 11, 7, 5, 15, 12, 99, 1 } >> the condition j==1 will be reached, which leads to execution of Bomb(). The test case will fail as it doesn't have any exceptions.

Mutant 3: Relational Operator Replacement

Reachability:

Always reached if array is not empty or null.

Infection:

When j = 0 this leads to the condition arr[j-1] to access index -1 instead of passing over the while loop at the conditional operator is not satisfied.

Propagation:

This leads to the program crashing due to an unhandled exception: <<java.lang.ArrayIndexOutOfBoundsException: Index -1 out of bounds for length 10>> as an example.

Mutant 3 results in ArrayIndexOutOfBoundsException because of the loop trying to access arr[j-1] which will be the index -1. As there is no implemented exception handling for the method the program doesn't gracefully handle these.

Mutant 4: Arithmetic Operator Replacement

Reachability:

In all scenarios will the mutant be reached as the for loop is the first line in the method.

Infection:

The infection will always be present as the code will always reach the for loop and the value of i will be infected.

Propagation:

The mutant leads to a wrong output, only every second element is being sorted.

- Test data: [0,4,3,7,11,5,15,12,99,1]
- Expected result: [0,1,3,4,5,7,11,12,15,99]
- Actual result: [0,4,3,7,11,5,15,12,99,1]

Test Method Analysis

Analysis for Test Method 1: BubbleSort

Observations:

The test method that was chosen is testBubbleSort()' this test method initialises an array of unsorted Integers that is then passed to method to be sorted and then compare to see that the results match the defined sorted order. The test data is [4,3,0,11,7,5,15,12,99,1] with an expected result of [0,1,3,4,5,7,11,12,15,99]. The test data under the mutation testing strongly kills mutants 1,2 and 4 and the third mutant remains undetected due to the fact it is an equivalent mutant.

Sufficiency of Testing:

Overall, the test suite effectively evaluates the sorting methods validity by using a strong set of test data, leading to discovery of the majority of mutants and potential defects.

Analysis for Test Method 2: InsertionSort

Observations:

For the Insertion Sort method we tested the effects of four differnt mutants to ensure that the selected test harness is utilising strong test cases to make sure that no defects are present in the code. From this we can see that the test harness doesn't accurately handle exceptions which can be detrimental in more complex code bases.Mutants 1 and three both result in an a array index out of bounds error with mutant 2 causing a run time exception, these exceptions are not handled by any try catch method and thus lead to the program ending prematurely. Mutant 4 leads to an result where very second element is sorted cuasing the actual not to match the expected result.

Sufficiency of Testing:

As the source 'Insertion Sort' code is not as complex, it clearly depicts how important it is to properly handle exceptions, in case if the precondition is not met (null values or inconsistent data types), it's better for the code to fail fast with a clear exception message. In that case, there is no need to look through the code, in order to identify what caused the crash.

To improve test harness, boundary checks must be included and exception handling to prevent runtime errors due to out of bounds access.

Exception handling should include:

- i) Null Elements in Array: Handling null values correctly to either sort them to the beginning or end based on defined behavior, or throw an informative exception.
- ii) Single Element Array and Empty Array, e.g. if(arr== null) OR (arr.length<=1).
- iii) Handling Non-Comparable Data: to prevent runtime exceptions when elements are not comparable.

Testing Tool Investigation

Chosen Tool: Cypress

Introduction

Cypress is a popular **JavaScript-based end-to-end testing framework** developed specifically for the needs of modern **web application testing**. Cypress **integrates with other tools and frameworks** (e.g. Mocha), but also works well as a standalone framework. It's uniquely built to simplify every aspect of testing—from setup to debugging—by running in the same execution cycle as the application under test. This approach not only **enhances test reliability and speed** but also **integrates seamlessly with development workflows** (Cypress.io, 2024).

Purpose and Functionality

Cypress is designed to handle both simple and complex testing scenarios with full **native access** to the Document Object Model (DOM) and all related objects. This **direct access eliminates the need for third-party drivers or proxies**, allowing tests to **interact with the application as a user would** (Cypress.io, 2024).

Usage

Installing Cypress requires just a few commands via the Node Package Manager (npm), integrating **directly into any web project**. It's particularly **suited for JavaScript developers**, providing an intuitive **API** and **syntax** (Cypress.io, 2024).

Basic Example Test: Login Form

```
describe('Login Form', () => {
  it('successfully logs in the user', () => {
    // Visit the login page
    cy.visit('https://example.com/login');

    // Type in the username and password fields
    cy.get('input[name=username]').type('user@example.com');
    cy.get('input[name=password]').type('password123');

    // Click the login button
    cy.get('button[type=submit]').click();

    // Check the URL to be redirected to the dashboard
    cy.url().should('include', '/dashboard');

    // Optional: Verify the presence of an element that is only visible when logged in
    cy.get('.welcome-message').should('contain', 'Welcome, user!');
  });
});
```

This example demonstrates how you might write a Cypress test to check the functionality of a login form on a web application. The test will navigate to the login page, enter credentials, submit the form, and verify that the login was successful based on the expected URL change.

Cypress operates entirely within the browser and does not require any external drivers or servers to run. This test showcases how Cypress alone can handle:

- Navigation
- DOM querying
- Interaction (clicks, typing)
- Assertions to verify states and contents

By utilising Cypress's rich API for handling HTTP requests, manipulating cookies, and interacting with local storage, you can extend such tests to include more complex scenarios like handling authentication tokens or testing different user roles.

This approach ensures that Cypress can fully manage and execute sophisticated end-to-end tests without the need for additional testing frameworks or tools.

Detailed Example Test: Multi-Step Registration Form

```
describe('Multi-Step Registration Form', () => {
  it('successfully registers a new user', () => {
    // Visit the registration page
    cy.visit('https://example.com/register');

    // Step 1: Enter personal details
    cy.get('input[name=firstName]').type('John');
    cy.get('input[name=lastName]').type('Doe');
    cy.get('input[name=email]').type('john.doe@example.com');
    cy.get('button.next').click();

    // Step 2: Enter address details
    cy.get('input[name=address]').type('1234 Elm Street');
    cy.get('input[name=city]').type('Metropolis');
    cy.get('select[name=state]').select('New York');
    cy.get('input[name=zip]').type('12345');
    cy.get('button.next').click();

    // Step 3: Set username and password
    cy.get('input[name=username]').type('john_doe');
    cy.get('input[name=password]').type('secure12345');
    cy.get('button[type=submit]').click();

    // Verify that the registration was successful based on the expected confirmation message
    cy.contains('Please check your email to activate your account').should('be.visible');

    // Optional: Check for an API call that sends a confirmation email
    cy.intercept('POST', '/api/send-confirmation-email').as('confirmationEmail');
    cy.wait('@confirmationEmail').its('response.statusCode').should('eq', 200);
  });
});
```

This example demonstrates how you might write a Cypress test to check the functionality of a multi-step registration form on a web application. The test will navigate to the registration page, enter user details, handle form validations, navigate through multiple steps, and finally verify registration by checking for a confirmation email.

Cypress facilitates comprehensive testing of complex web application functionalities like multi-step forms, including:

- **Navigation and Interaction**: Managing complex user flows through different stages of a form.
- **Advanced Assertions**: Ensuring that each step is completed successfully and the final confirmation is received.
- **Network Requests**: Intercepting and asserting on network requests to validate backend processes like sending confirmation emails.

This approach highlights Cypress's ability to handle sophisticated end-to-end tests, providing real-time feedback and robust debugging capabilities. It showcases how effectively Cypress integrates into modern web development workflows, making it a powerful tool for ensuring application reliability and user satisfaction.

Other Considerations

While Cypress's rapid adoption in the developer community and its extensive plugin ecosystem offer substantial advantages, there are several considerations that teams should evaluate before choosing it as their primary testing tool:

1. **Single Language Support**: Cypress is built exclusively for JavaScript. This tight integration allows for robust features and an intuitive testing experience for JavaScript developers. However, this can also be a limitation for teams working in multi-language environments or those that prefer using languages like Python, Ruby, or Java for their testing frameworks. Teams should consider their project's language alignment and developer expertise when choosing a testing tool (Cypress.io, 2024).
2. **Specific Browser Compatibility**: Cypress supports testing in Chrome, Firefox, Edge, and Brave directly. While this covers a significant portion of web users, the lack of support for browsers like Safari or Internet Explorer might be a constraint for projects requiring broad browser compatibility, especially those targeting diverse user demographics or corporate environments still using older browsers (Cypress.io, 2024).

3. **Community and Documentation**:
    - **Robust Community and Resources**: Cypress benefits from a strong and active community. Its extensive documentation, active forums, and numerous plugins enhance its functionality and ease of use. This strong community support is invaluable for troubleshooting, sharing best practices, and continuous learning, making Cypress a top choice for teams that value community engagement.
    - **Plugin Ecosystem**: The availability of a wide range of plugins can extend Cypress's capabilities beyond basic testing scenarios. These plugins include integrations for accessibility testing, visual regression testing, and more, allowing teams to adapt the tool to their specific needs (Cypress.io Documentation, 2024).
4. **Performance and Real-Time Testing**:
    - **Fast Execution Within the Browser**: Because Cypress tests run in the same loop as the application, test execution is generally faster and less prone to synchronisation issues compared to tools that operate outside the browser. This can lead to more efficient test cycles, especially during development phases.
    - **Real-Time Feedback**: The Cypress Test Runner provides visual feedback and step-by-step test execution, which is a significant advantage during test development and debugging. Developers can see exactly what happens at each step of their test and quickly identify where things go wrong (Cypress.io, 2024).
5. **Architectural Considerations**:
    - **Native Access to Application Code**: Cypress operates within the application context, allowing for deeper integration with the application being tested. This native access facilitates more complex interactions and state management within tests, providing a more thorough testing experience.
    - **Test Isolation and Control**: While Cypress provides excellent control over test execution and environment setup, its model of clearing the state between tests ensures test independence and reliability. However, managing state across tests can sometimes require additional setup or use of workarounds, particularly for complex state persistence scenarios.
6. **Ease of Integration**: Cypress is straightforward to integrate into existing JavaScript projects and plays well with various build systems and CI/CD pipelines. However, teams need to consider the integration overhead if they are migrating from another testing framework or integrating with non-JavaScript parts of their tech stack (Cypress.io, 2024).

These considerations should be weighed carefully against the specific requirements and constraints of a project to determine if Cypress is the most suitable testing tool for its needs. It excels in environments that can leverage its strengths—JavaScript-centric development, need for rapid testing feedback, and robust community support—but might present challenges in more diverse tech ecosystems or where extensive cross-browser testing is crucial.

Recommendation

Cypress is highly recommended for teams engaged in developing interactive, dynamic web applications, particularly those using modern JavaScript frameworks such as React, Angular, or Vue. It is especially beneficial for projects that prioritise rapid development cycles and high-quality user interfaces.

Presentation Slides: Cypress Presentation

See Appendix 1 for images of each slide or click the link above to view online.

Comparative Analysis with Rivals

Feature/Tool	Cypress	Selenium	Puppeteer	Playwright
Language Support	✓ (JavaScript)	✓ (Multi-lang)	✓ (JavaScript)	✓ (Multi-lang)
Architecture	✓ (Internal)	(External)	(External)	✓ (Internal)
Ease of Setup	✓		✓	✓
Execution Speed	✓		✓	✓
Browser Support		✓		✓
Use Case	✓			✓
Community and Ecosystem	✓	✓		✓
Testing Capabilities		✓		✓

[1]

Pros and Cons Comparison

Cypress

- **Pros:** Direct access to DOM, real-time test execution, simplified debugging.
- **Cons:** Limited language and browser support, not suited for multi-domain management.

Selenium

- **Pros:** Supports multiple languages, extensive browser support, mature ecosystem.
- **Cons:** Complex setup, slower execution.

Puppeteer

- **Pros:** Fast execution in Chrome, ideal for direct browser control tasks.
- **Cons:** Limited to Chrome, not designed for cross-browser testing.

Playwright

- **Pros:** Supports multiple languages and browsers, fast and reliable execution.
- **Cons:** Less mature than Selenium, smaller community.

[1] NOTE: This analysis was discussed in the presentation and provided here for informational purposes only.

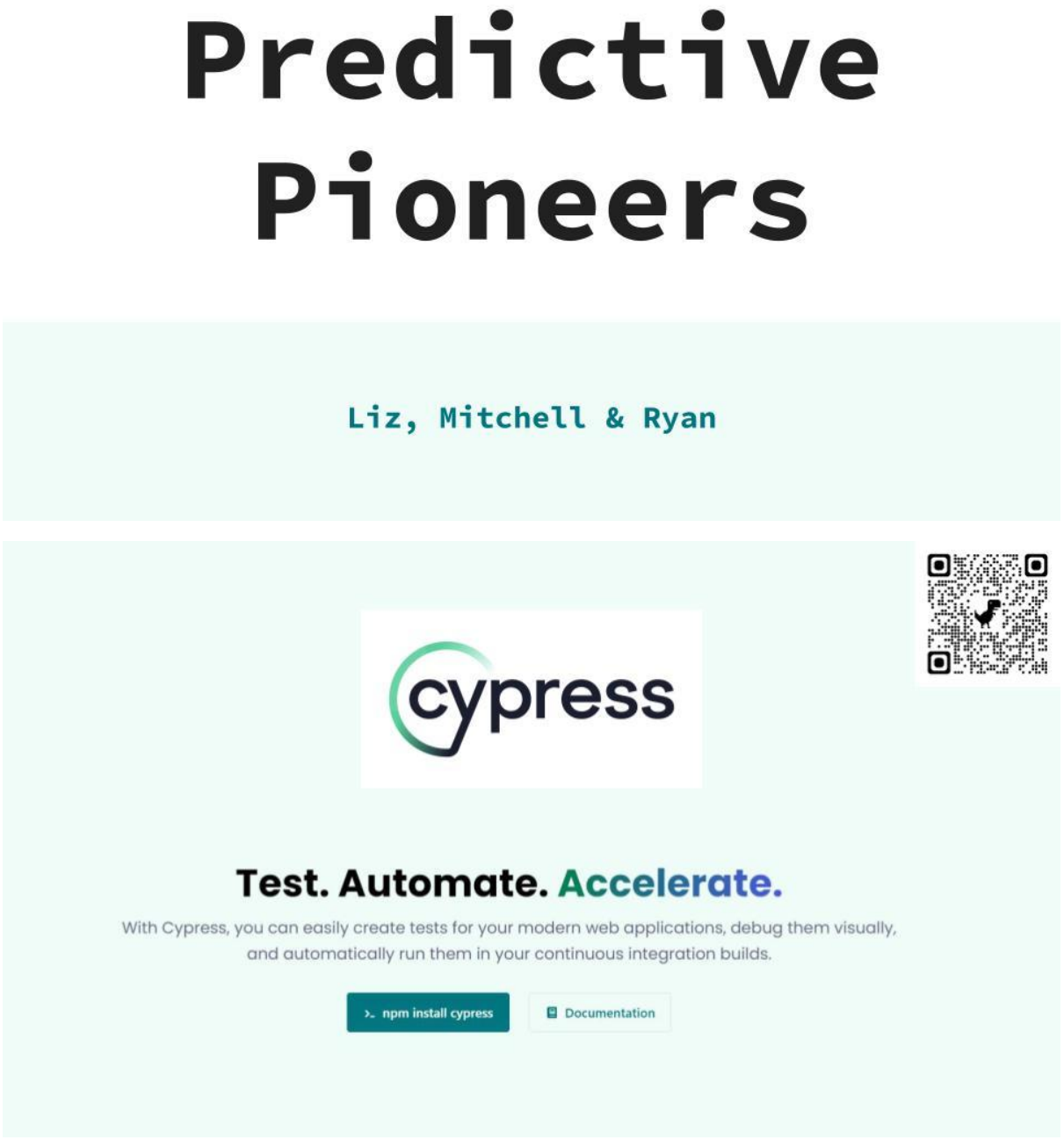
References

- Cypress.io. (2024). Retrieved from https://www.cypress.io/
- Cypress.io. (2024). Cypress Documentation. Retrieved from https://docs.cypress.io
- Cypress.io. (2024). How Cypress Works. Retrieved from https://www.cypress.io/how-it-works/
- Cypress.io. (2024). JavaScript End to End Testing Framework. Retrieved from https://go.cypress.io/get-started

APA 7th Edition.

Appendices

Appendix 1: Presentation Slides



Who is Cypress?

Founded in 2014 Venture Backed

5M+ Weekly Downloads

45k+ GitHub Stars

1M+ Dependent Repositories

23.5k+ Companies Atlassian, Siemens, Dell

Why Cypress?

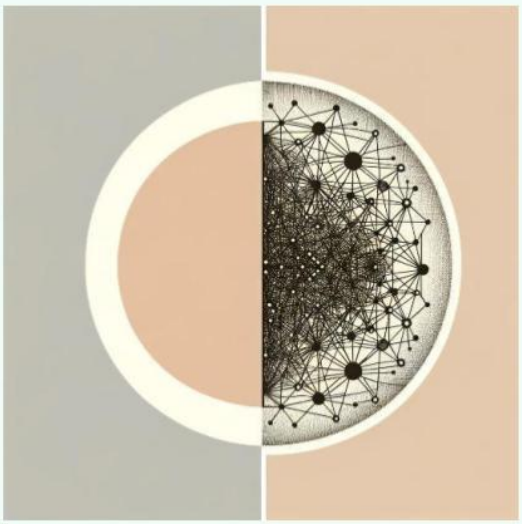
FEATURE/TOOL	CYPRESS	SELENIUM	PUPPETEER	PLAYWRIGHT
LANGUAGE SUPPORT			✓	✓
ARCHITECTURE				✓
				✓
E				✓
BROWSER SUPPORT		✓		✓
USE CASE	✓			✓
COMMUNITY & ECOSYSTEM	✓	✓		✓
TESTING CAPABILITIES		✓		✓

10/10 Documentation

Purpose

Designed for both simple and complex scenarios.

Direct access to the DOM without third-party drivers.



Usage

Install with just a few commands via npm.

Ideal for JavaScript developers.

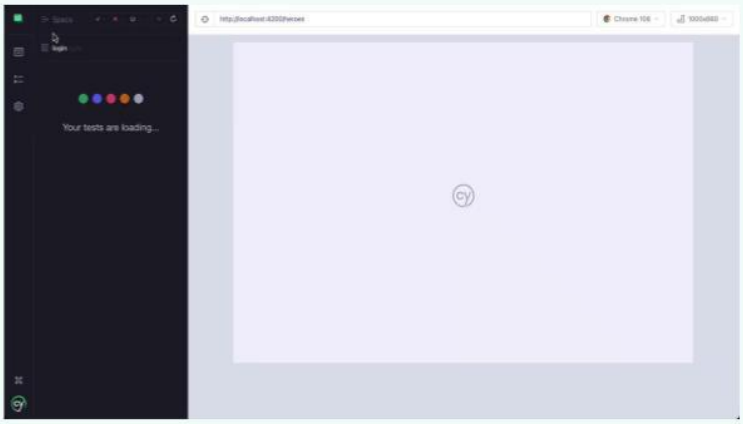
Example

Navigate, interact, and verify-all in the browser

```
describe('Login Form', () => {
  it('successfully logs in the user', () => {
    // Visit the login page
    cy.visit('https://example.com/login');

    // Type in the username and password fields
    cy.get('input[name=username]').type('user@example.com');
    cy.get('input[name=password]').type('password123');

    // Click the login button
    cy.get('button[type=submit]').click();

    // Check the URL to be redirected to the dashboard
    cy.url().should('include', '/dashboard');

    // Optional: Verify the presence of an element that is
only visible when logged in
    cy.get('.welcome-message').should('contain', 'Welcome,
user!');
  });
});
```

Example

Navigate, interact, and verify-all in the browser



Considerations

Primarily supports JavaScript.

Optimised for Chrome, Firefox, Edge, and Brave.

Yay or Nay

Enhances testing process quality and reliability.

Highly recommended for dynamic web applications.

Thanks for listening!