ST Project Report: Mutation Testing on Dynamic Programming based Algorithms

Siddharth Kothari (IMT2021019)
Sankalp Kothari (IMT2021028)

November 26, 2024

Contents

1 Introduction to Mutation Testing

1.1 Definition of Mutation Testing

Mutation testing is a software testing technique designed to assess the quality and effectiveness of test cases. It involves introducing small, deliberate modifications (called mutations) to a program's source code. These modifications simulate common programming errors which a human may introduce during the development phase, which can include changing of relational operators (a < operator may be substituted by <= operator etc.) negating a condition, usage of a wrong variable, or altering a constant. The goal is to determine if the existing test cases can detect and fail due to these mutations.
If a test case detects a mutation and fails as a result, the mutation is said to be killed. If a mutation goes undetected and does not cause any test failures, it is considered survived, which may indicate weaknesses in the test suite. The objective of Mutation Testing, thus, is to test whether the given testcases are able to indeed catch the various mistakes that a programmer may commit.

1.2 Need for Mutation Testing

Listed below are a few advantages of using Mutation Testing -

1. **Evaluate Test Suite Quality**: Mutation testing helps evaluate how effectively a test suite can detect real-world bugs by simulating potential programming errors. If tests cannot "kill" the mutants, it suggests gaps in the test coverage.
2. **Identify Weak Tests**: It pinpoints poorly designed test cases that might pass even when the program contains faults, thereby ensuring that the tests truly validate the behavior of the code.
3. **Enhance Test Design**: By exposing weaknesses in the test suite, mutation testing encourages developers to write more robust and comprehensive tests that catch subtle bugs.

While other forms of testing, including logic based testing or data flow based testing focus on coverage criteria, and the coverage of all parts of the code, mutation testing adopts a different approach, and instead focuses on catching subtle errors in the code. It offers the following advantages over other types of testing -

1. **Focus on Bug Detection**: As stated previously, mutation testing ensures that the tests are not just executing code but are also verifying its correctness by detecting subtle errors.
2. **Uncover Hidden Issues**: Unlike code coverage tools, which just highlight the number of lines of code executed, mutation testing uncovers undetected logical flaws which may go unnoticed.
3. **Challenges Superficial Tests**: Simple tests, like those checking basic functionality without addressing edge cases, are likely to fail mutation testing.

The primary drawback of mutation testing is the high cost associated with generating mutants and checking whether each mutant is killed or not, which may require that the entire test suite be run for each mutant.

1.3 Killing of Mutants

Mutants are defined as programs generated from the source code by the application of only one mutation operator at a time. We define the notions of weakly killing and strongly killing mutants. In the definitions below, the program on which testing is being performed is denoted by P, the set of mutants for the program is denoted by M, and the test case considered is denoted by t.

1. **Weakly Killing Mutants**: Given a mutant $m \in M$, that modifies a location l in a program P, and a test case t, t is said to weakly kill m if the state of execution of P on t is different from the state of execution of m immediately after l.
2. **Strongly Killing Mutants**: Given a mutant $m \in M$ for a ground string program P, and a test t, t is said to strongly kill m if the output of t on P is different from the output of t on m.

For our project, we primarily focus on strong killing of the mutants. This allows to assess the behavior and robustness of the code, and ensure that observable changes are introduced in the behavior of the program.

1.4 Levels of Mutation Testing

Mutation Testing can be applied to both unit and integration levels.

1. **Unit Mutation Testing** involves applying mutation testing at the unit level, where individual components or functions of a program are tested in isolation. Mutations are introduced into specific units of code (e.g., methods, classes, or functions), and the goal is to evaluate how well the unit tests can detect these modifications.
2. **Integration Mutation Testing** is applied at the integration level, where multiple units or modules are tested together to verify their interactions. Mutations are introduced into the interfaces, connections, or communication points between these units, focusing on detecting faults that occur when modules work together.

For our project, we primarily focus on Unit Testing.

1.5 Types of Mutants

There are primarily four types of mutants -

1. **Stillborn Mutants**: Mutants of a program result in invalid programs that cannot even be compiled. Such mutants should not be generated.
2. **Trivial Mutants**: A mutant that can be killed by almost any test case.
3. **Equivalent Mutants**: Mutants that are functionally equivalent to a given program. No test case can kill them.
4. **Dead Mutants**: Mutants that are valid and can be killed by a test case. These are the only useful mutants for testing.

In our work, we primarily focus on killing the mutants, and try to explain a few mutants which are not killed, primarily due to the fact that they are equivalent mutants.

1.6 Mutation Operators

Mutation operators are rules or transformations used to introduce small, deliberate changes (mutations) into a program's source code during mutation testing. Each operator targets a specific aspect of the program's structure, such as logic, arithmetic, or control flow.
Mutation operators for Unit Testing can be categorized based on the type of change they introduce into the following categories (non-exhaustive list) -

1. **Arithmetic Operators**: Modify arithmetic expressions by replacing operators. + can be replaced by a $-$, $*$ or / etc.
2. **Relational Operators**: Change relational operators to simulate errors in comparisons. A $>$ may be replaced by a $>=$, $<=$, or $<$ etc.
3. **Logical Operators**: Modify logical expressions, such as AND, OR, and NOT. && may be replaced by ||, conditions may be negated etc.
4. **Assignment Operators**: Alter assignment operations to simulate incorrect assignments. A $+=$ may be replaced by a $-=$, $*=$ or a $/*$ etc.
5. **Constant Replacement Operators**: Replace constants with different values. A 0 may be replaced by a 1 etc.

1.7 Mutation Score

Mutation Score is defined as the percentage of mutants killed out of the total number of mutants. A higher mutation score need not necessarily indicate high quality tests, but it is a good measure nonetheless.

2 Problem Statement Description, Motivation and Implementation

2.1 Problem Statement

We focus primarily on Dynamic Programming based algorithms for our project. The problems considered are a mixture of standard dynamic programming based algorithms, to combinatorics based problems which can use dynamic programming to speed up the computations, and have been described below -

1. **0/1 Knapsack** - Given N items where each item has some weight and profit associated with it and also given a bag with capacity W, (i.e., the bag can hold at most W weight in it). The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.
2. **Longest Increasing Subsequence** - Given an integer array, return the length of the longest strictly increasing subsequence. A subsequence can have non consecutive elements from the array, in the same order as they are present in the array.
3. **Longest Palindromic Subsequence** - Given a string s, find the length of the longest palindromic subsequence in s. A palindrome is a string, which when reversed, returns the same string.
4. **House Robber** - Given an integer array nums, find the maximum sum possible from the elements of the array, such that no two of the elements chosen are consecutive.
5. **Weighted Interval Scheduling** - Given N intervals, with a start time, end time, and a profit, find the subset of intervals with the maximum profit such that none of the intervals in the subset overlap.
6. **Word Break** - Given a dictionary of words and a string s, find whether s can be broken down into substrings which are present in the dictionary or not.
7. **Coin Change** - Given a set of denominations of coins and a target amount, find the minimum number of coins needed to make the target amount.
8. **Longest Common Subsequence** - Given two string s_1 and s_2, find the lenght of the longest subsequence common in both the strings.
9. **Catalan Number** - Given n, find the n^{th} Catalan Number, using both the recursive and the closed form solution.
10. **Factorial** - Given n, find $n!$
11. **Stirling Number** - Given r objects, find the number of ways to distribute the r objects into n non-empty subsets (denoted by $S(r, n)$).
12. **Levensthein Distance** - Given two strings word1 and word2, return the minimum number of operations required to convert word1 to word2. The operations available are to insert a character, delete a character or replace a character.
13. **Minimum Operations for Multiplication** - Given the dimension of a sequence of matrices in an array arr, where the dimension of the i^{th} matrix is (arr[i-1] * arr[i]), the task is to find the most efficient way to multiply these matrices together such that the total number of element multiplications is minimum.
14. **Maximum Product Subarray** - Given an integer array nums, find a subarray that has the largest product, and return the product.
15. **Fibonacci Number** - Given an integer n, find the n^{th} Fibonacci number.
16. **Binomial Coefficient** - Given integers n and k, find the value of $C(n, k)$.
17. **Derangement Count** - Given integer n, find the value of D_n. D_n is defined as the number of ways to arrange n objects into n boxes, where the i^{th} object should not go into the i^{th} box.

We implemented these functions in both Java and Python, and compare the mutation testing results. We compare the operators provided, along with the mutation scores for the 2 languages. We also identify examples of equivalent mutants from the source code.

2.2 Motivation for the Problem

All of the problems listed above are standard Dynamic Programming and Combinatorics Problems, which are fundamental in the understanding of the design and analysis of algorithms, algorithm implementation techniques, and discrete mathematics.
Another reason to choose the above problems is due to the large presence of arithmetic, conditional and relational operators in the codes for these programs. The codes largely utilize arithmetic operators, and heavily rely on conditional statements, along with relational operators to form logical conditions. This allows us to test out a wide variety of mutation operators, and hence we chose this problem statement.

2.3 Implementation of Algorithms and Test Cases

The codes have been implemented in Java and Python, test cases for Java have been written using JUnit, with PIT being utilised for mutation testing, while the test cases in Python are written using MutPy and unittest library. We focus primarily on unit testing only. We wrote a total of **99** test cases per language.

Approximate Lines of Code $\sim$ **2400** lines (1000 code + 1400 tests).

1. Java -
    - ClassicalDP Class - 500 lines
    - Interval Class - 10 lines
    - Test Cases in JUnit - 800 lines
2. Python -
    - ClassicalDP Class - 460 lines
    - Interval Class - 10 lines
    - Test Cases - 600 lines

3 Mutation Tests and Results

We implemented the DP algorithms in 2 programming languages, namely - Java and Python and performed Unit testing and performed Mutation testing on said Unit testcases.
For Java, unit tests were written using JUnit and the mutation testing was performed using PIT, whereas for Python, unit testing was performed using PyTest and mutation testing was done using MutPy.

3.1 Test Case Design

For brevity, we only discuss the mutation test results. We wrote different test cases that would take all different control paths atleast once, so that almost all of the code can get covered in our test cases.

Initially, we wrote inadequate test cases, which resulted in not all of the mutants being killed. We discuss the same using the report from Java. As we can see in Figure 1. As we can see, initially the code coverage was approximately 85%, and there were multiple mutants which had not been killed. We then identified those mutants, along with test cases which could kill them. An example for the same is discussed below.

Figure 2 shows the code for the method calculating Stirling number for a given n and r. Since there are so many if blocks and we would need all of them to be covered for better Code Coverage, we manually designed multiple test cases as shown in figure 3 that would take all of the different if blocks and PIT would be able to perform mutation testing on all said test cases, thus leading to better coverage.
Another example is shown in Figure 4. As we can see, by the introduction of new test cases, we are able to kill more mutants.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	90%	219/242	85%	292/342	91%	292/321

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.example.services	1	90%	219/242	85%	292/342	91%	292/321

Report generated by PIT 1.9.11

Enhanced functionality available at arcmutate.com

Figure 1: Initial coverage

```java
// Function to calculate the Stirling number of the second kind S(n, r)
public int stirling_number(int r, int n) {
    if (n < 0 || r < 0) {
        return -1;
    }
    if (r < n) {
        return 0;
    }
    if (r == 0) {
        return 1;
    }
    if (n == 0) {
        return 0;
    }
    if (n == r) {
        return 1;
    }

    // Create a 2D list to store the Stirling numbers
    int[][] dp = new int[n + 1][r + 1];

    // Base cases
    for (int i = 1; i <= n; i++) {
        dp[i][i] = 1;
    }
    for (int i = 1; i <= r; i++) {
        dp[1][i] = 1;
    }

    // Fill in the rest of the dp table
    for (int j = 2; j <= r; j++){
        for (int i = 2; i <= n; i++){
            dp[i][j] = dp[i - 1][j - 1] + (i) * dp[i][j - 1];
        }
    }

    // Return the Stirling number
    return dp[n][r];
}
```

Figure 2: Stirling number problem code

```java
@Test
public void stirling_number_case1() {
    // Case 1: n = 0, r = 0 (base case)
    int n = 0, r = 0;
    int result = dp.stirling_number(r, n);
    assertEquals(expected:1, result);
}

@Test
public void stirling_number_case2() {
    // Case 2: n = 5, r = 5 (n == r)
    int n = 5, r = 5;
    int result = dp.stirling_number(r, n);
    assertEquals(expected:1, result);
}

@Test
public void stirling_number_case3() {
    // Case 3: r = 3, n = 2 (valid input)
    int r = 3, n = 2;
    int result = dp.stirling_number(r, n);
    assertEquals(expected:3, result); // Stirling number S(3, 2) = 3
}

@Test
public void stirling_number_case4() {
    // Case 4: r = 2, n = 3 (n > r)
    int r = 2, n = 3;
    int result = dp.stirling_number(r, n);
    assertEquals(expected:0, result);
}

@Test
public void stirling_number_case5() {
    // Case 5: n = -1, r = 3 (invalid input)
    int n = -1, r = 3;
    int result = dp.stirling_number(r, n);
    assertEquals(-1, result);
}

@Test
public void stirling_number_case6() {
    // Case 5: n = 3, r = 9 (valid input)
    int n = 3, r = 9;
    int result = dp.stirling_number(r, n);
    assertEquals(expected:3025, result);
}
```

Figure 3: Testcases for Stirling number problem

```
395      // Function to find the minimum number of operations to multiply matrices
396      public int matrixMultiplication(int[] arr) {
397          int N = arr.length;
398 2        if (N < 2) {
399 1            return -1;
400          }
401 4        if (Arrays.stream(arr).anyMatch(dim -> dim <= 0)){
402 1            return -1;
403          }
404
405          int[][] dp = new int[N][N];
406
407          // Loop through chain lengths from 2 to N
408 2        for (int length = 2; length < N; length++) {
409 3            for (int i = 0; i < N - length; i++) {
410 1                int j = i + length;
411                  dp[i][j] = Integer.MAX_VALUE;
412 3                for (int k = i + 1; k < j; k++) {
413 4                    dp[i][j] = Math.min(dp[i][j], dp[i][k] + dp[k][j] + arr[i] * arr[k] * arr[j]);
414                  }
415              }
416          }
417
418 2        return dp[0][N - 1];
419      }
```

(a)

```
395      // Function to find the minimum number of operations to multiply matrices
396      public int matrixMultiplication(int[] arr) {
397          int N = arr.length;
398 2        if (N < 2) {
399 1            return -1;
400          }
401 4        if (Arrays.stream(arr).anyMatch(dim -> dim <= 0)){
402 1            return -1;
403          }
404
405          int[][] dp = new int[N][N];
406
407          // Loop through chain lengths from 2 to N
408 2        for (int length = 2; length < N; length++) {
409 3            for (int i = 0; i < N - length; i++) {
410 1                int j = i + length;
411                  dp[i][j] = Integer.MAX_VALUE;
412 3                for (int k = i + 1; k < j; k++) {
413 4                    dp[i][j] = Math.min(dp[i][j], dp[i][k] + dp[k][j] + arr[i] * arr[k] * arr[j]);
414                  }
415              }
416          }
417
418 2        return dp[0][N - 1];
419      }
```

(b)

Figure 4: Some examples of the generated mutants

3.2 Java (PIT)

The major mutation operators used are listed below -

1. CONDITIONALS_BOUNDARY
2. EMPTY_RETURNS
3. FALSE_RETURNS
4. INCREMENTS
5. INVERT_NEGS
6. MATH
7. NEGATE_CONDITIONALS
8. NULL_RETURNS
9. PRIMITIVE_RETURNS
10. TRUE_RETURNS
11. VOID_METHOD_CALLS

The code structure for PIT testing and some of the mutants used along with the final generated report are shown here in figures 5,6,7 -

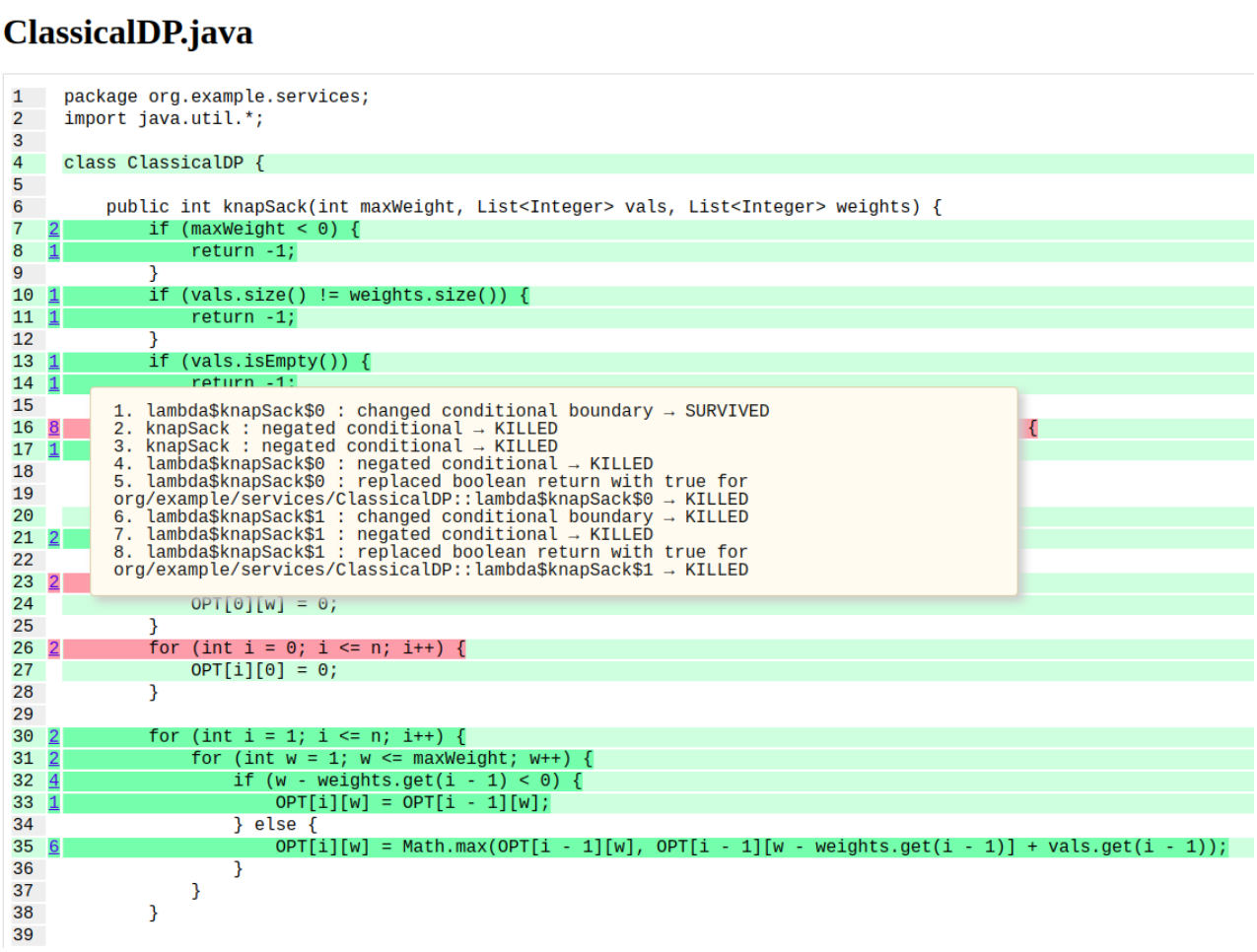

Figure 5: Code Structure in HTML report for PIT testing in Java

Mutations

11	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
12	1. replaced int return with 0 for org/example/services/ClassicalDP::knapSack → KILLED
14	1. negated conditional → KILLED
15	1. replaced int return with 0 for org/example/services/ClassicalDP::knapSack → KILLED
17	1. negated conditional → KILLED
18	1. replaced int return with 0 for org/example/services/ClassicalDP::knapSack → KILLED
20	1. negated conditional → KILLED 2. negated conditional → KILLED 3. changed conditional boundary → SURVIVED 4. negated conditional → KILLED 5. replaced boolean return with true for org/example/services/ClassicalDP::lambda$knapSack$0 → KILLED 6. changed conditional boundary → KILLED 7. negated conditional → KILLED 8. replaced boolean return with true for org/example/services/ClassicalDP::lambda$knapSack$1 → KILLED
21	1. replaced int return with 0 for org/example/services/ClassicalDP::knapSack → KILLED
25	1. Replaced integer addition with subtraction → KILLED 2. Replaced integer addition with subtraction → KILLED

Figure 6: Mutation operators for PIT testing in Java

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	97%	235/242	91%	312/342	93%	312/337

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.example.services	1	97%	235/242	91%	312/342	93%	312/337

Report generated by PIT 1.9.11

Enhanced functionality available at arcmutate.com

Figure 7: Coverage report generated for PIT testing in Java

3.3 Python (MutPy)

The mutation operators used are listed below -

- AOR - Arithmetic Operator Replacement
- AOD - Arithmetic Operator Deletion
- ASR - Assignment Operator Replacement
- BCR - Break Continue Replacement
- COI - Conditional Operator Insertion
- LCR - Logical Connector Replacement
- ROR - Relational Operator Replacement
- SIR - Slice Index Remove

Below are the necessary screenshots from the generated report (fig. 8 & 9)

Figure 8: Some examples of the generated mutants

3.4 Reasons for non killing of a few mutants

There are primarily 2 reasons due to which the mutants may not be killed -

1. Inadequacy of test cases.
2. Presence of equivalent mutants.

We formally identify a few equivalent mutants in our code, and discuss them in the next section.

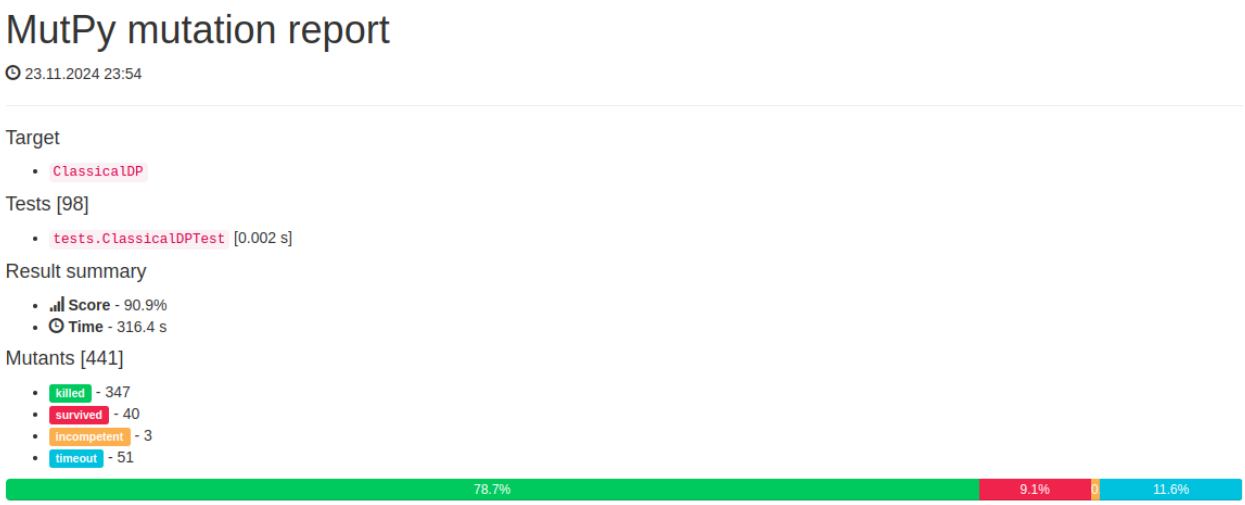

Figure 9: Coverage report generated for MutPy testing in Python

4 Equivalent Mutants

As stated previously, there are a few mutants which cannot be killed simply due to the fact that they are equivalent mutants. We provide a few examples in this section.
As we see in Figure 10, if we replace the $<$ with a $<=$ in line 79, we get an equivalent mutant. The reason is simply that the inner for loop will not run a single iteration due to the condition $i < n - length$, and hence running an extra iteration of the outer for loop will not have any effect on the final answer. This is an example of a mutant which cannot be killed.
Another example of a mutant which cannot be killed is shown in Figure 11. In this example, when in the mutant $j <= i$ is applied instead of $j < i$, we find that this is an equivalent mutant. The reason is that $dp.get(j)$ is never true in the start. We start from $j = 0$, and continue up. If we find a j which satisfies the condition given below, we immediately break and do not reach the case when $j = i$. But if we do not find such a $j < i$, even in that case, $dp.get(j)$ would never be true when $j = i$, hence that loop iteration will also not have any effect on the output.
These are examples which show that we can never have 100% mutation score, as some generated mutants may be equivalent mutants, which cannot be killed.

5 Comparison between PIT and MutPy

Table 1 provides a comparison of our observations when we apply Mutation operators from PIT and MutPy to our code. Both tools achieve comparable mutation scores, with PIT at 91% and MutPy at 90.9%. However, MutPy generates and kills more mutants (441 and 401, respectively) than PIT (342 and 312). Despite this, PIT is significantly faster, with a runtime of 26.102 seconds, compared to MutPy's 329.13 seconds. This highlights PIT's efficiency, while MutPy demonstrates a higher number of mutants handled, albeit at the cost of runtime performance.

Feature	**PIT**	**MutPy**
Mutation Score	91%	90.9%
Number of Mutants Generated	342	441
Number of Mutants Killed	312	401
Runtime	26.102 s	329.13 s

Table 1: Comparison of Python and MutPy for Mutation Testing

```
68      public int longestPalindrome(String arg) {
69          int n = arg.length();
70  1       if (n == 0) {
71              return 0;
72          }
73
74          int[][] OPT = new int[n][n];
75  2       for (int i = 0; i < n; i++) {
76              OPT[i][i] = 1;
77          }
78
79  2       for (int length = 1; length < n; length++) {
80  3           for (int i = 0; i < n - length; i++) {
81  1               int j = i + length;
82  1               if (arg.charAt(i) == arg.charAt(j)) {
83  3                   OPT[i][j] = OPT[i + 1][j - 1] + 2;
84                   } else {
85  2                   OPT[i][j] = Math.max(OPT[i + 1][j], OPT[i][j - 1]);
86                   }
87               }
88          }
89
90  2       return OPT[0][n - 1];
91      }
```

Figure 10: Equivalent Mutant in Longest Palindromic Subsequence Problem

```
195     public boolean wordBreak(String s, List<String> wordDict) {
196         Set<String> wordSet = new HashSet<>(wordDict);
197         int n = s.length();
198 1       if (n == 0) {
199 1           return true;
200         }
201
202 1       List<Boolean> dp = new ArrayList<>(Collections.nCopies(n + 1, false));
203         dp.set(0, true);
204
205 2       for (int i = 1; i <= n; i++) {
206 2           for (int j = 0; j < i; j++) {
207 2               if (dp.get(j) && wordSet.contains(s.substring(j, i))) {
208                     dp.set(i, true);
209                     break;
210                 }
211             }
212         }
213
214 2       return dp.get(n);
215     }
216
```

Figure 11: Equivalent Mutant in Word Break Problem

Table 2 provides a general comparison between PIT and MutPy. PIT excels in execution speed, efficient test detection, and generates detailed HTML reports, making it ideal for large, enterprise Java projects. It benefits from a large community, active maintenance, and extensive documentation. In contrast, MutPy, while suited for Python projects in academic or small-scale applications, is slower due to less optimization and runs entire test suites for each mutation. MutPy provides basic text-based reporting, has limited mutation operators, and lacks the same level of community support and documentation as PIT. Overall, PIT is better for robust, large-scale use, while MutPy is more tailored for Python-specific needs.

Feature	**PIT**	**MutPy**
Language Support	Java	Python
Execution Speed	Highly optimized for speed with efficient test execution strategies.	Slower compared to PIT due to less optimization and Python's interpreted nature.
Test Detection Mechanism	Runs only the test cases affected by the introduced mutations.	Typically runs the entire test suite for each mutation, leading to longer execution times.
Reporting	Generates detailed, customizable HTML reports.	Provides reports in text and basic formats but lacks advanced customization.
Community Support and Documentation	Large community and active maintenance. Extensive Documentation.	Smaller community and less frequent updates. Limited Documentation.
Examples of Use Cases	Best suited for enterprise Java projects with complex test suites.	Ideal for Python projects in academic or small-to-medium scale applications.

Table 2: Comparison of PIT and MutPy Mutation Testing Tools

6 Individual Contributions

We worked together on designing the testcases, and picking up the problems. Siddharth worked on the implementation of the code in Python, and unit tests in Java, while Sankalp worked on the implementation in Java, and unit tests in Python.

7 Relevant Links

The Github Link to the project can be found at - https://github.com/siddharth-kothari9403/Mutation_Testing.