(10) 1.

Sound	Complete.
- guarantees no false positives	- guarantees no false negatives
- cannot guarantee all vulnerabilities	- cannot guarantee false positives
- (meaning) doesnot find all vulnerabilities but if it says there is a bug, then there is a bug.	- (meaning) it might point out vulnerabilities that does not even exist but it does point out all possible vulnerabilities.
- eg: if analysis says that X is true then X is true.	- eg: if X is true, then analysis says X is true.

- True Positive: If analysis finds a bug which is an actual bug.
  real life example: Doctor concluding a visibly shown pregnant lady as pregnant.

- True Negative: If analysis finds there is no bug when there are actually no bugs.
  real life example: Doctor concluding a healthy man as not pregnant.

- False Positive: If analysis finds a bug when it is not an actual bug.
  real life example: Doctor concluding a healthy man as pregnant.

- False Negative: If analysis doesnot find any bug when there is a bug.
  real life example: Doctor concluding showing woman as not pregnant.

	Finding a bug (Positive)	Not finding a bug (Positive)
TP.	finding an actual bug	not finding an absent bug
TN.	finding no bugs when bugs not present.	not finding bugs when none present
FP.	falsely finding a bug which is not a bug	finding an incorreet bug.
FN.	not finding a bug which is present.	not finding bug when bug present.

2.

(18) (A) We're implementing an insertion sort algorithm in the following code. It begins by generating a random number, 'n,' which determines the size of an array. 'n' is selected randomly from the range 0 to 25. We use this 'n' as the size parameter when we call the 'GenerateArray' function. This function generates an array of size 'n' with random elements, each ranging from 0 to 10000. We use two generators, A and B, for this purpose.

The generated array is stored in the variable 'a.' Subsequently, we pass this 'a' to our insertion sort function, which sorts the array. The sorted array is then stored in 'b.'

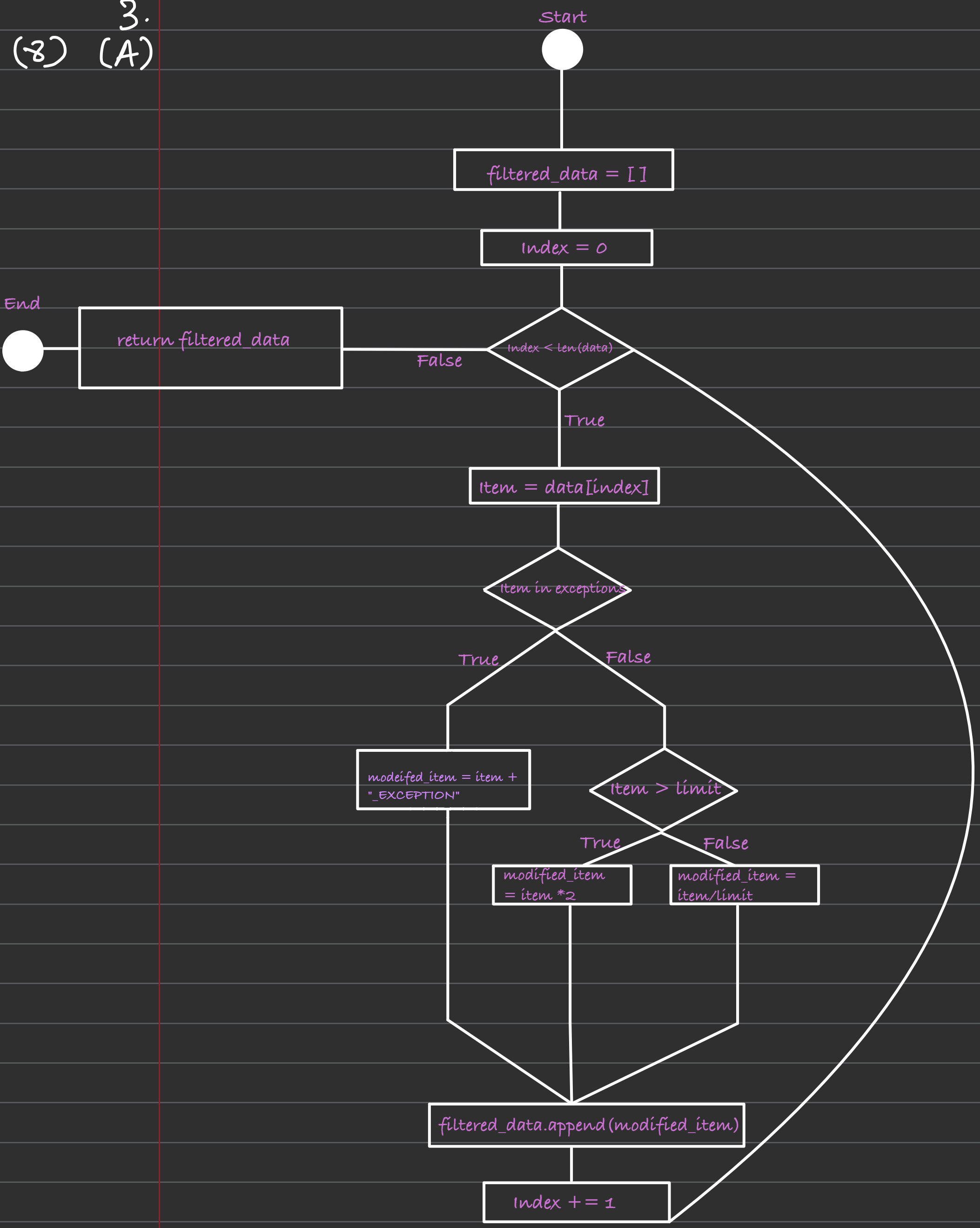
To ensure that the sorting operation has been successful, we iterate through the elements in the array and check if each element is smaller than the next one. If this order is not maintained, we raise an Assertion error.

To run the program, compile the 'q2.java' file located in the 'q2' folder.

(8) (B)

3.

(8) (A)


Start
filtered_data = [ ]
Index = 0
End
return filtered_data
False
Index < len(data)
True
Item = data[index]
Item in exceptions
True
False
modeifed_item = item + "_EXCEPTION"
Item > limit
True
False
modified_item = item *2
modified_item = item/limit
filtered_data.append(modified_item)
Index += 1

(8) (B) For our code above, we can use Random Testing to check its robustness. However, we cannot fully rely on Random Testing in real life as it might not be "complete".

Random Testing: Generate random values for 'data', 'limit', 'exceptions'. Then, testing these randomly generated values making sure they are valid. (random elements array (data), number or integer (limit), integer array (exceptions) for comparision. Make sure the test is ran on wide range of input values of (data, limit, exceptions). We will use 'Filter Data' function to run/test the values of the parameters. After execution, compare it with exceptions, given the fact it meets all other code req. Take a note of all/any errors occured during the tests and keep testing. Keep a note of input values, expected results and actual results for comparison.

During the whole test we need to check limits to make sure data is above or below the limit and if not making sure exceptions are there.

5.

(4) (A) By manually checking the code, I found out that the line 7 contains error. `output_str += char * 2`

So, if we put an input of aump4 it does aauummpp44 which is not what we want. We wanted aauummpp

To fix it, we would either have to remove the multiplication by 2 or multiply it with the unit number 1 so it does not change the output and gives us expected output.

(5) 6. https://github.com/aumpandyaa/COSC3P95

Q4.

a)
Test case 1:
data = [2,4,6,8]
limit =4
exceptions = [4]
Statement coverage: (13/13) ~ 100%
Branch coverage: (4/4) ~ 100%

Test case 2:
data = [2,4,6,8]
limit =10
exceptions = [0]
Statement coverage: (11/13) = 84.615~ 85%
Branch coverage: (2/4) ~ 50%

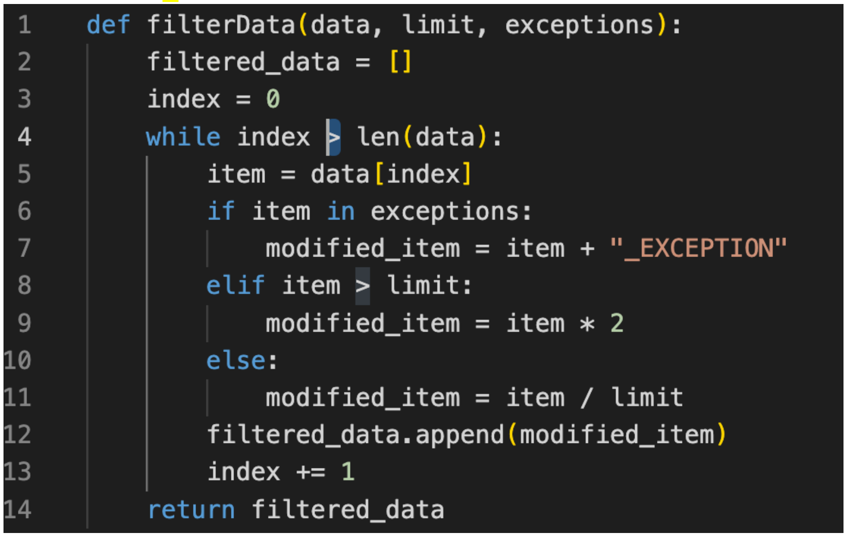
Test case 3:
data = [2,4,6,8]
limit =1
exceptions = [2,4,6,8]
Statement coverage: (9/13)= 69.12 ~ 69%
Branch coverage: (2/4) ~ 50%

Test case 4:
data = []
limit =1
exceptions = [13]
Statement coverage: (4/13)= 30.769 ~ 31%
Branch coverage: (1/4) ~ 25%

b)

Mutation 1:

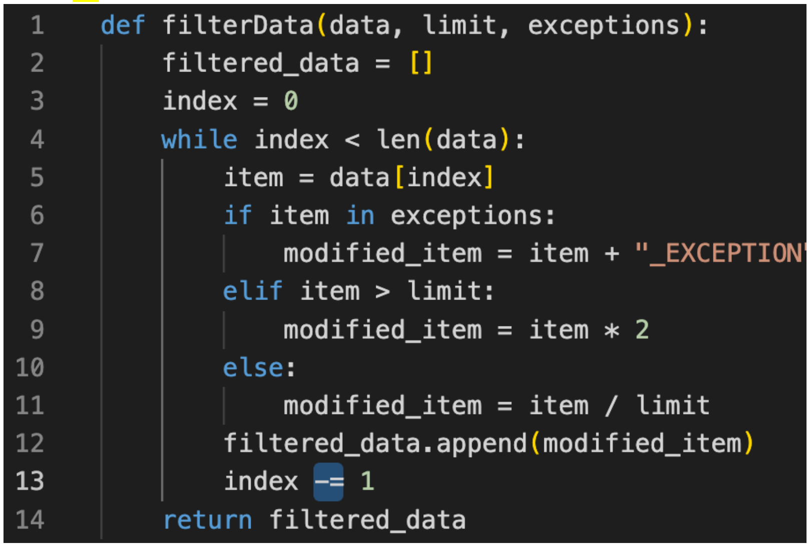
While index > len(data)



```python
def filterData(data, limit, exceptions):
    filtered_data = []
    index = 0
    while index > len(data):
        item = data[index]
        if item in exceptions:
            modified_item = item + "_EXCEPTION"
        elif item > limit:
            modified_item = item * 2
        else:
            modified_item = item / limit
        filtered_data.append(modified_item)
        index += 1
    return filtered_data
```

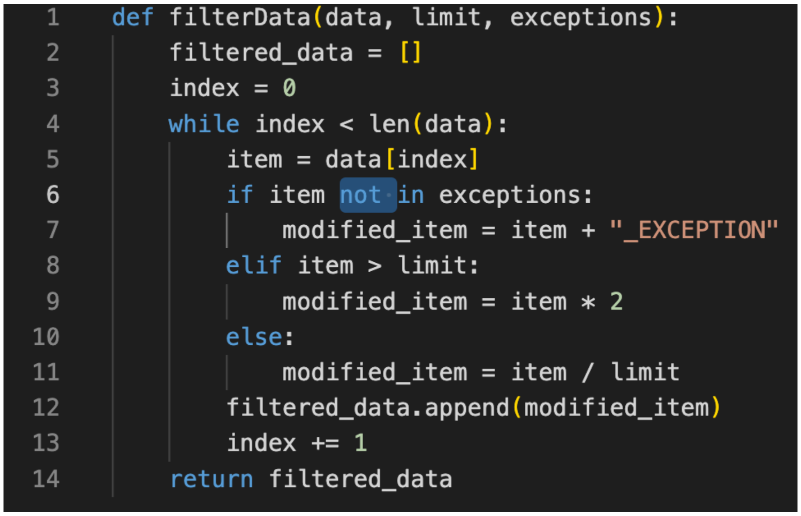
Mutation 2:

Index-=1



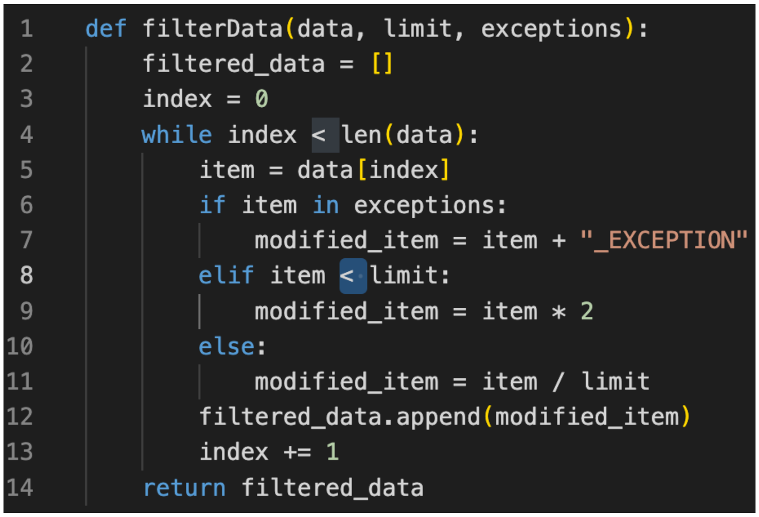
```python
def filterData(data, limit, exceptions):
    filtered_data = []
    index = 0
    while index < len(data):
        item = data[index]
        if item in exceptions:
            modified_item = item + "_EXCEPTION
        elif item > limit:
            modified_item = item * 2
        else:
            modified_item = item / limit
        filtered_data.append(modified_item)
        index -= 1
    return filtered_data
```

Mutation 3:
if item not in exceptions:



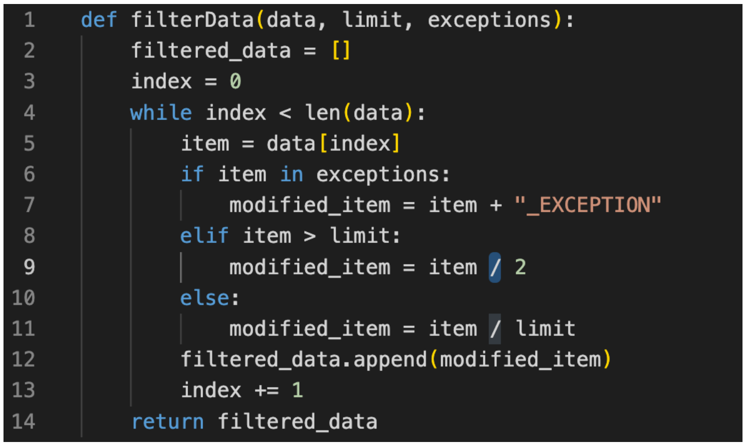
```
def filterData(data, limit, exceptions):
    filtered_data = []
    index = 0
    while index < len(data):
        item = data[index]
        if item not in exceptions:
            modified_item = item + "_EXCEPTION"
        elif item > limit:
            modified_item = item * 2
        else:
            modified_item = item / limit
        filtered_data.append(modified_item)
        index += 1
    return filtered_data
```

Mutation 4:
elif item < limit:



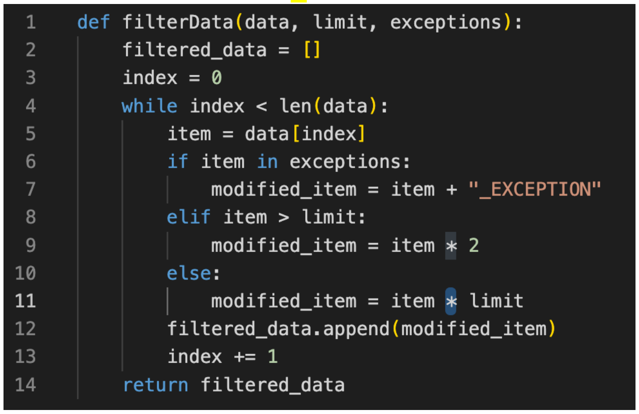
```
def filterData(data, limit, exceptions):
    filtered_data = []
    index = 0
    while index < len(data):
        item = data[index]
        if item in exceptions:
            modified_item = item + "_EXCEPTION"
        elif item < limit:
            modified_item = item * 2
        else:
            modified_item = item / limit
        filtered_data.append(modified_item)
        index += 1
    return filtered_data
```

Mutation 5:
modified_item = item / 2; (line 9)



```
1   def filterData(data, limit, exceptions):
2       filtered_data = []
3       index = 0
4       while index < len(data):
5           item = data[index]
6           if item in exceptions:
7               modified_item = item + "_EXCEPTION"
8           elif item > limit:
9               modified_item = item / 2
10          else:
11              modified_item = item / limit
12          filtered_data.append(modified_item)
13          index += 1
14      return filtered_data
```

Mutation 6:
modified_item = item * limit; (line 11)

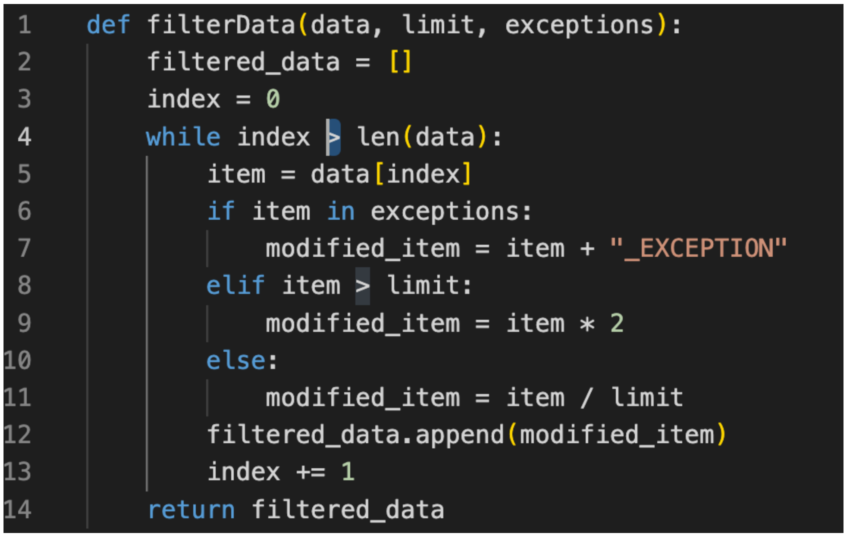


```
1   def filterData(data, limit, exceptions):
2       filtered_data = []
3       index = 0
4       while index < len(data):
5           item = data[index]
6           if item in exceptions:
7               modified_item = item + "_EXCEPTION"
8           elif item > limit:
9               modified_item = item * 2
10          else:
11              modified_item = item * limit
12          filtered_data.append(modified_item)
13          index += 1
14      return filtered_data
```

c)

Mutation 1: (Rank=1)

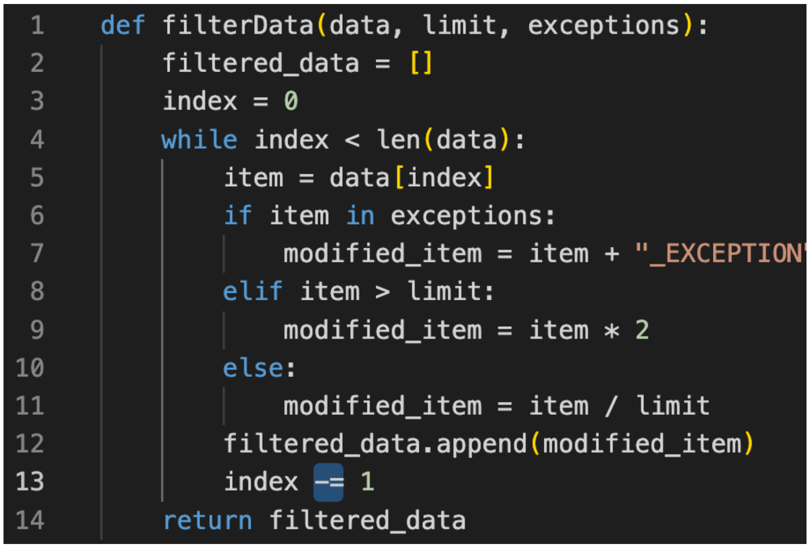
For this version we had changed "While index > len(data)". When we run mutated and the original program with the test suite, we can say that we are able to say that the mutation is detected and we are able to kill the mutation in early stage.



```python
def filterData(data, limit, exceptions):
    filtered_data = []
    index = 0
    while index > len(data):
        item = data[index]
        if item in exceptions:
            modified_item = item + "_EXCEPTION"
        elif item > limit:
            modified_item = item * 2
        else:
            modified_item = item / limit
        filtered_data.append(modified_item)
        index += 1
    return filtered_data
```

Mutation 2: (Rank=2)

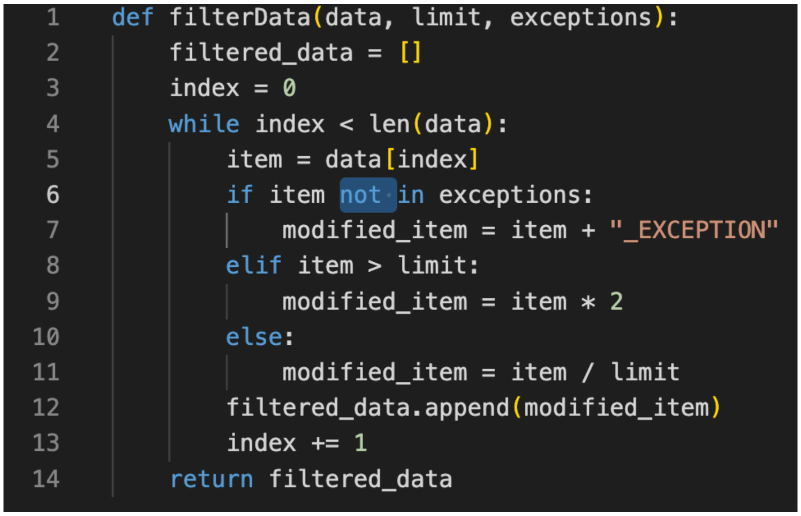
For this version we had changed "Index-=1". When we run mutated and the original program with the test suite, we can say that we are able to detect and kill the mutation after the 1 pass of the program as it will throw some exception when it tries to access data[-1]



```python
def filterData(data, limit, exceptions):
    filtered_data = []
    index = 0
    while index < len(data):
        item = data[index]
        if item in exceptions:
            modified_item = item + "_EXCEPTION
        elif item > limit:
            modified_item = item * 2
        else:
            modified_item = item / limit
        filtered_data.append(modified_item)
        index -= 1
    return filtered_data
```

Mutation 3: (Rank=3)

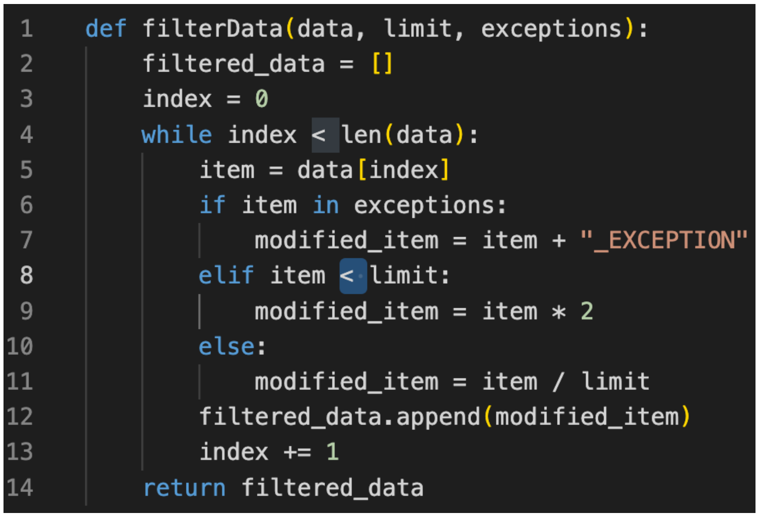
For this version of mutation we have changed “if item not in exceptions:”. When we run the test suite on original and mutated version we can easily see that the elements that are not in exceptions will be appended with _EXCEPTION which will be flagged out early and we can say that by mutation analysis we are able to kill the mutation



```
def filterData(data, limit, exceptions):
    filtered_data = []
    index = 0
    while index < len(data):
        item = data[index]
        if item not in exceptions:
            modified_item = item + "_EXCEPTION"
        elif item > limit:
            modified_item = item * 2
        else:
            modified_item = item / limit
        filtered_data.append(modified_item)
        index += 1
    return filtered_data
```

Mutation 4: (Rank=4)

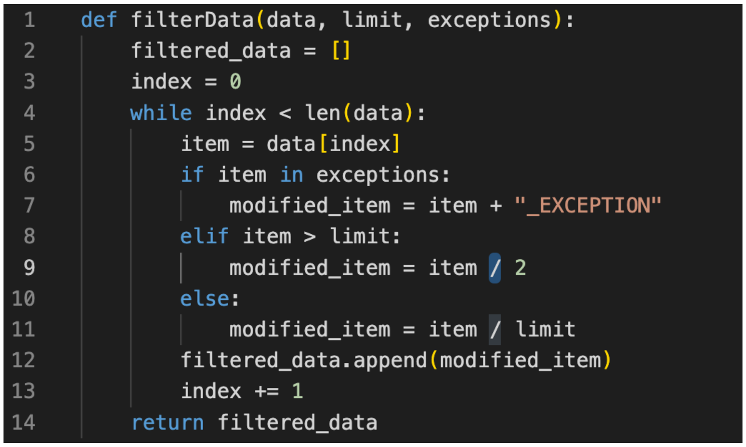
For this mutated version we have changed “elif item < limit:”. When we run the original and mutated code with the test suite by help of mutation analysis we are able to kill the mutation as any of the element inside data is less than the limit it will be multiplied by 2 instead of being divided by limit. Thus we are able to kill the mutation.



```
def filterData(data, limit, exceptions):
    filtered_data = []
    index = 0
    while index < len(data):
        item = data[index]
        if item in exceptions:
            modified_item = item + "_EXCEPTION"
        elif item < limit:
            modified_item = item * 2
        else:
            modified_item = item / limit
        filtered_data.append(modified_item)
        index += 1
    return filtered_data
```

Mutation 5: (Rank =5)

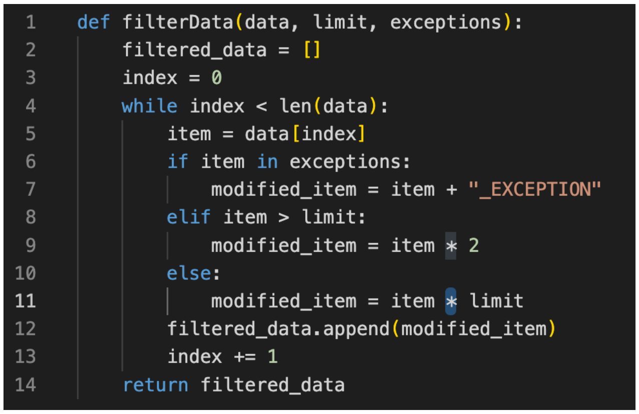
For this we have mutated “modified_item = item / 2; (line 9)”. When we run test suite against original and mutated code by help of mutation analysis we are able to kill the mutation as if data item is greater than limit it will be divided by 2 instead of being multiplied by 2.



```
 1  def filterData(data, limit, exceptions):
 2      filtered_data = []
 3      index = 0
 4      while index < len(data):
 5          item = data[index]
 6          if item in exceptions:
 7              modified_item = item + "_EXCEPTION"
 8          elif item > limit:
 9              modified_item = item / 2
10          else:
11              modified_item = item / limit
12          filtered_data.append(modified_item)
13          index += 1
14      return filtered_data
```

Mutation 6: (Rank=6)

For this version we have mutated “modified_item = item * limit;” (line 11). When we run the test suite against mutated and original code. By mutation analysis we are able to kill the mutation as the elements that are supoosed to be divided by the limit are instead being multiplied by the limit.



```
 1  def filterData(data, limit, exceptions):
 2      filtered_data = []
 3      index = 0
 4      while index < len(data):
 5          item = data[index]
 6          if item in exceptions:
 7              modified_item = item + "_EXCEPTION"
 8          elif item > limit:
 9              modified_item = item * 2
10          else:
11              modified_item = item * limit
12          filtered_data.append(modified_item)
13          index += 1
14      return filtered_data
```

d)

In order to path test the above code, I will be writing a test suite that covers all the Paths of the code thus making sure that all paths are covered at least once.

For example the following test suite tests all paths of the code:

data = [22,44,66,88]

limit =4

exceptions = [88]

Statement coverage: (13/13) ~ 100%

Branch coverage: (4/4) ~ 100%

In order to Branch test above code, I will be writing a test suite that makes sure that all the paths are tested at least once which means that all the branches are visited at least once. The above test case satisfies branch testing.

Since Statement static analysis involves checking the code for syntactic errors and the best way to test it is by compiling it using any of the compilers by this way we can make sure that our code is free of any syntactic errors.