1 Killed Mutant

I have selected the mutant in **ListUtils.java** line 101 within the *intersection* function. The specific code is as follows:

```
88      public static <E> List<E> intersection(final List<? extends E> list1, final List<? extends E> list2) {
89          final List<E> result = new ArrayList<>();
90
91          List<? extends E> smaller = list1;
92          List<? extends E> larger = list2;
93  2       if (list1.size() > list2.size()) {
94              smaller = list2;
95              larger = list1;
96          }
97
98          final HashSet<E> hashSet = new HashSet<>(smaller);
99
100         for (final E e : larger) {
101 1           if (hashSet.contains(e)) {
102                 result.add(e);
103                 hashSet.remove(e);
104             }
105         }
106 1       return result;
107     }
```

One of the test cases is from **ListUtilsTest.java line 103**, from `org.apache.commons.collections4` (so do the rest of the mutants in my report) in which the test case tests whether a list is equal to an intersection with itself.

The mutant will first check if an element from the larger list exists in the hashSet. If the hashSet contains the element, the hashSet will not add the element to the result. In short, the mutant actually returns a list of elements in the larger list that do not exist in the smaller list. In the test case, the list provided is `['a','b','c','d','e']`, and the hashSet. In this case, the mutant will return an empty list according to the explaination above. The test case assertion will be false as `['a','b','c','d','e']` is not equal to an empty list, so the mutant is killed.

```
99      /**
100      * Tests intersecting a non-empty list with itself.
101      */
102     @Test
103     public void testIntersectListWithSelf() { assertEquals(fullList, ListUtils.intersection(fullList, fullList)); }
106
```

The screenshot is the test case that kills the mutant.

2 Improving Mutant Score

I have selected the mutant in **ListUtils.java line 711** within the *get* function. The specific code is as follows:

```
696     private static class Partition<T> extends AbstractList<List<T>> {
697         private final List<T> list;
698         private final int size;
699
700         private Partition(final List<T> list, final int size) {
701             this.list = list;
702             this.size = size;
703         }
704
705         @Override
706         public List<T> get(final int index) {
707             final int listSize = size();
708 2           if (index < 0) {
709                 throw new IndexOutOfBoundsException("Index " + index + " must not be negative");
710             }
711 2           if (index >= listSize) {
712                 throw new IndexOutOfBoundsException("Index " + index + " must be less than size " +
713                                                     listSize);
714             }
715 1           final int start = index * size;
716 1           final int end = Math.min(start + size, list.size());
717 1           return list.subList(start, end);
718         }
```

Note that this function is within the partition class that overides the `get()` function from the list class. The mutant that has survived is the changed conditional boundary case, where it has changed to `if(index > listSize)` This mutant should be killed as when `index == listSize`, it should still be out of the bound of the list. The test case is located at line 432 in *ListUtilsTest.java*. With a list `[0,1,2,3,4,5,6]`, a partition of size 3 is given as `[[0,1,2],[3,4,5],[6]]`. The size of this partition list is 3, so the maximum index that can be called to this partition list is 2. The test case provided calls `assertEquals(1,partition.get(2).size())`. If I can change 2 to 3 as in: `assertEquals(1,partition.get(3).size())`, the mutant should be killed. As now the original function will throw as error whereas the mutant does not throw an arrow. With a different behaviour, the mutant is killed.

```
public void testPartition() {
    final List<Integer> strings = new ArrayList<>();
    for (int i = 0; i <= 6; i++) {
        strings.add(i);
    }

    final List<List<Integer>> partition = ListUtils.partition(strings, size: 3);

    assertNotNull(partition);
    assertEquals( expected: 3, partition.size());
    assertEquals( expected: 1, partition.get(2).size());

    try {
        partition.get(3).size();
        fail("Index out of bounds");
    } catch (final IndexOutOfBoundsException e) {}
```

My Test Case

The bottom try catch block is my test case that kill will the proposed mutant.

3 Improving Coverage and Mutation Score

I have selected the mutant in **ListUtils.java line 255** within the *isEqualList()* function. The specific code is as follows:

```
237     public static boolean isEqualList(final Collection<?> list1, final Collection<?> list2) {
238 1       if (list1 == list2) {
239 1           return true;
240         }
241 3       if (list1 == null || list2 == null || list1.size() != list2.size()) {
242 1           return false;
243         }
244
245         final Iterator<?> it1 = list1.iterator();
246         final Iterator<?> it2 = list2.iterator();
247         Object obj1 = null;
248         Object obj2 = null;
249
250 2       while (it1.hasNext() && it2.hasNext()) {
251             obj1 = it1.next();
252             obj2 = it2.next();
253
254 3           if (!(obj1 == null ? obj2 == null : obj1.equals(obj2))) {
255 1               return false;
256             }
257         }
258
259 3       return !(it1.hasNext() || it2.hasNext());
260     }
```

The mutation is to change `return false;` to `return true;`. For this particular function, there are test cases in which the lists have different lengths, or one list is `null`. However, there is no test case in which a list contains a `null`. Hence, I can try comparing two lists of the same length, with one list containing `null` and another list **not** containing `null` to kill this mutation.

In my test case, I assign list A to be `[1,2,3,null,5]` and list B to be `[1,2,3,4,5]`, and we expect the result to be false. The two lists will pass `if(list1 == null || list2 == null || list1.size() != list2.size())`, and it wil reach to the Iterator part:

The first 3 objects in the iterators of list1 and list2 will be fine, but when it reaches the fourth object, which is `null` in list1 and `4` in list2, the mutant will return true while the original code will return false. Hence, we are allowed to kill the mutant with the test case designed above.