Enunciado 1

Considere o método busca abaixo [Ammann e Offutt 2016, cap 6]:

public static int busca(List lista, Object item)

// Saídas: se lista ou item forem nulos, lança NullPointerException

// senão, se item está na lista, retorna o índice da sua posição; senão retorna -1

// Exemplo: busca([3, 1, 3], 3) pode retornar 0 ou 2

// busca([7, 5, 0], 2) retorna -1

Suponha que foi proposta a seguinte partição baseada na localização do item na lista:

Classe 1: item é o primeiro da lista

Classe 2: item é o último da lista

Classe 3: item está em alguma posição que não é a primeira e nem a última.

1. **O particionamento viola a propriedade de completeza. Mostre com um exemplo.**

R: Um exemplo é a busca([1,2,3], 4), no qual o item não está na lista, não se encaixando em nenhuma das classes de equivalência.

2. **Este particionamento viola a propriedade de disjunção. Mostre um exemplo.**

R: Um exemplo é a busca([1], 1) que se encaixa tanto na classe 1 quanto na classe 2.

3. **Proponha novas classes para o particionamento dado que não viole as propriedades de disjunção e completeza.**

R: O conjunto das seguintes classes satisfaz as propriedades de disjunção e completeza:

- Classe 1: lista é nula
- Classe 2: item é nulo
- Classe 3: item está na lista
- Classe 4: item não está na lista

Enunciado 2

Considere a classe PrimeNumbers a seguir.

```
// Introduction to Software Testing
// Authors: Paul Ammann & Jeff Offutt
// Chapter 3;

import java.util.*;

/*
 * CLASS NAME: PrimesNumbers
 *               Class to compute N prime numbers
 * ORIGINAL AUTHOR: Michael Wilson
 *
 * NOTE: The class has a fault that results in false negatives
```

```
 *
 */
public class PrimeNumbers implements Iterable<Integer>
{
   private List<Integer> primes = new ArrayList<Integer>();

   /*
    * creates a list of n prime numbers
    * @param  n - the number of primes to compute
    * silently treats negative arguments as zero
    */
   public void computePrimes (int n)
   {
      int count = 1; // count of primes
      int number = 2; // number tested for primeness
      boolean isPrime; // is this number a prime

      while (count <= n)
      {
         isPrime = true;
         for (int divisor = 2; divisor <= number / 2; divisor++)
         {
            if (number % divisor == 0)
            {
               isPrime = false;
               break; // for loop
            }
         }
         if (isPrime && (number % 10 != 9)) { // THIS IS THE FAULT!!!
            primes.add (number);
            count++;
         }
         number++;
      }
   }

   /*
   * Returns an Iterator that will iterate through the primes
  */
   @Override public Iterator<Integer> iterator()
   {
      return primes.iterator();
   }

  /*
   * Returns a String representation
  */
   @Override public String toString()
   {
      return primes.toString();
   }

   public static void main (String[] argv)
```

```
    {
        PrimeNumbers primes = new PrimeNumbers();
        primes.computePrimes(8);
        System.out.println("Primes: " + primes);

        Iterator<Integer> itr = primes.iterator();
        System.out.println("First prime: " + itr.next());
    }
}
```

```
  Para este código, você vai criar casos de teste conforme indicado nas
questões a seguir.
Se não puder criar um caso de teste, justifique.
Somente na última questão é que os testes serão implementados usando o JUnit.
```

4. **Considere a classe PrimeNumbers dado no enunciado.**
    a. **Particione o domínio do parâmetro de entrada em classes de equivalência.**

R: Uma partição possível é:

- Classe 1: n < -2147483648
- Classe 2: -2147483648 ≤ n ≤ 0
- Classe 3: n = 1
- Classe 4: 1 < n < 8
- Classe 5: 8 ≤ n ≤ 2147483647
- Classe 5: n > 2147483647

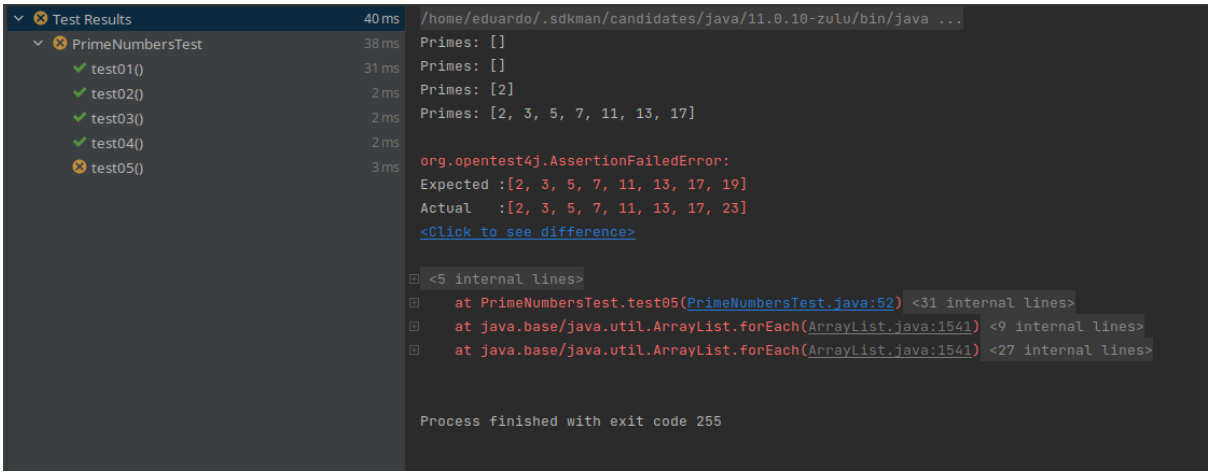
Note que inteiros em java podem ir de -2147483648 a 2147483647 e os casos de limite superior e inferior seriam inviáveis para teste.
Note também que, a partir do valor de entrada 8, o primeiro número que não satisfaz a condição `number % 10 != 9`, que é o número 19, passa a ser omitido da lista.

    b. **Crie casos de teste usando análise de valores-limite.**

R: Casos de teste gerados a partir da partição acima que utiliza análise de valores-limite:

- n = -2147483649; erro de compilação, por isso não pode ser automatizado
- n = -2147483648; lista/iterador vazio
- n = 0; lista/iterador vazio
- n = 1; [2]
- n = 7; [2, 3, 5, 7, 11, 13, 17]
- n = 8; [2, 3, 5, 7, 11, 13, 17, 19]
- n = 2147483647; [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, …], porém é computacionalmente inviável
- n = 2147483648; erro de compilação, por isso não pode ser automatizado

    c. **Aplique os casos de teste ao programa. Entregue um arquivo PDF contendo:**
        i. **as classes de equivalência.**
        ii. **os casos de teste gerados em 2.**
        iii. **a captura de tela mostrando os resultados de cada teste.**




```java
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class PrimeNumbersTest {

    @Test
    public void test01() {
        PrimeNumbers primes = new PrimeNumbers();
        int n = -2147483648;
        String expectedResult = "[]";
        primes.computePrimes(n);
        Assertions.assertEquals(expectedResult, primes.toString());
        System.out.println("Primes: " + primes);
    }

    @Test
    public void test02() {
        PrimeNumbers primes = new PrimeNumbers();
        int n = 0;
        String expectedResult = "[]";
        primes.computePrimes(n);
        Assertions.assertEquals(expectedResult, primes.toString());
        System.out.println("Primes: " + primes);
    }

    @Test
    public void test03() {
        PrimeNumbers primes = new PrimeNumbers();
        int n = 1;
        String expectedResult = "[2]";
        primes.computePrimes(n);
        Assertions.assertEquals(expectedResult, primes.toString());
        System.out.println("Primes: " + primes);
    }
```

```java
    @Test
    public void test04() {
        PrimeNumbers primes = new PrimeNumbers();
        int n = 7;
        String expectedResult = "[2, 3, 5, 7, 11, 13, 17]";
        primes.computePrimes(n);
        Assertions.assertEquals(expectedResult, primes.toString());
        System.out.println("Primes: " + primes);
    }

    @Test
    public void test05() {
        PrimeNumbers primes = new PrimeNumbers();
        int n = 8;
        String expectedResult = "[2, 3, 5, 7, 11, 13, 17, 19]";
        primes.computePrimes(n);
        Assertions.assertEquals(expectedResult, primes.toString());
        System.out.println("Primes: " + primes);
    }
}
```