Tuan

Question 1:

a) `def swap(x: Int, y: Int) = { val t = x; x = y; y = t }`

This function would throw an error because the parameters that are passed are immutable. Therefore, the values of the parameters cannot be changed.

b)
```
def swapEntries(a: Arrayp[Int], i: Int, j: Int) = {
    val t = a(i); a(i) = a(j); a(j) = t
  }
```

The effect of this function is to swap the value of the i+1-th element of the array with the j+1-th element of the array. The function would throw an error because the elements of the array a are mutable, but the array a is immutable.

Question 2:

```
object SideEffects{
  var x = 3; var y = 5

  def nasty(x: Int) : Int = { y = 1; 2 * x }

  def main(args: Array[String]) = println(nasty(x) + y)
}
```

The function `nasty` sets the variable `y` to 1 and returns `2 * x`. So it would return 7 because `y` is already set to 1. In the other case when we write `y + nasty(x)`, the result would be 11.

Question 3:

```
object Test{
  // Tests to check whether the sort function is correct
  def main(args : Array[String]) = {
    val array1 = Array("book", "dog", "books", "paper", "machine",
"laptop")
    val array2 = Array("lamp", "pen", "telephone", "cards", "cards",
"people")
    val array3 = Array("", "a", "t", "d", "r", "b")

    val sortedArray1 = Array("book", "books", "dog", "laptop",
"machine", "paper")
    val sortedArray2 = Array("cards", "cards", "lamp", "pen", "people",
"telephone")
    val sortedArray3 = Array("", "a", "b", "d", "r", "t")

    assert(array1.sorted.deep == sortedArray1.deep)
    assert(array2.sorted.deep == sortedArray2.deep)
```

```
    assert(array3.sorted.deep == sortedArray3.deep)

    val sortedRevArray1 = Array("paper", "machine", "laptop", "dog",
"books", "book")
    val sortedRevArray2 = Array("telephone", "people", "pen", "lamp",
"cards", "cards")
    val sortedRevArray3 = Array("t", "r", "d", "b", "a", "")

    assert(array1.sortWith(_ > _).deep == sortedRevArray1.deep)
    assert(array2.sortWith(_ > _).deep == sortedRevArray2.deep)
    assert(array3.sortWith(_ > _).deep == sortedRevArray3.deep)
  }
}
```

Question 4:

a) If `numStep` is strictly larger than 1/ε, then `timeStep` would be 0. So in that case the loop would go forever until the stack overflows. If `numStep` is smaller than 1/ε, then the `timeStep` can be represented as a floating number that is bigger than 0. Therefore, in this case the variable `time` would increment by `timeStep`. So the body of the loop would be executed `numStep` number of times.

b) After the loop `time` might be inaccurate by 1, if `numStep` is larger than 1/ε. Otherwise, it would be inaccurate by at most ε.

c)
```
while(time < timeEnd)
  {
    // Inv: 0 <= time <= timeEnd and time = k*timeStep for some k in N
    if(timeStep < e) timeStep = e // val e = machine epsilon
    time += timeStep
  }
  // Inv => time == timeEnd
```

With this loop, when `timeStep` is smaller than ε, `time` will be inaccurate by at most ε.

Question 5:

```
object Search{
  /** Does pat appear as a substring of line? */
  def search(pat: Array[Char], line: Array[Char]) : Boolean = {
    val K = pat.size; val N = line.size
    // Invariant: I: found = (line[i..i+K) = pat[0..K) for
    //                       some i in [0..j)) and 0 <= j <= N-K
    var j = 0; var found = false
    while(j <= N-K && !found){
      // set found if line[j..j+K) = pat[0..K)
```

```
      // Invariant: line[j..j+k) = pat[0..k)
      var k = 0
      while(k<K && line(j+k)==pat(k)) k = k+1
      found = (k==K)
      j = j+1
    }
    // I && (j=N-K+1 || found)
    // found = ( line[i..i+K) = pat[0..K) for some i in [0..N-K+1) )
    found
  }

  def main(args : Array[String]) = {
    val array1 = "word".toArray
    val array2 = "Does this sentence contain word?".toArray
    val array3 = "Why did the chicken cross the road?".toArray
    val array4 = "lamp".toArray
    val array5 = "p".toArray
    val array6 = "This contains wor".toArray
    val array7 = "This contains bord".toArray
    val array8 = "Does this sentence contain word".toArray

    assert(search(array1, array2) == true)

    // This would return exception if found = true
    assert(search(array1, array3) == false)

    // This would return exception if <= is replaced with <
    assert(search(array5, array4) == true)

    // This would return exception if N-K is replaced with N-K+1
    assert(search(array1, array6) == false)

    // This would return exception if k is 1 instead of 0
    assert(search(array1, array7) == false)

    // This would return out of bound if k <= K instead of k < K
    assert(search(array1, array8) == true)

    // Not sure if I can test whether == is replaced with >= on line 12
    // If k = K then the loop would stop, so k cannot be larger than K
  }
}
```

Question 6:

```
object RepeatingArrays{
  /** Test two strings for equality */
  def search(a : Array[Char], b : Array[Char]) : Boolean = {
    if(a.size != b.size) return false
    // Invariant : a[0..k) == b[0..k) && 0 <= k <= n
    var k = 0; val n = a.size
    while(k < n){
      if(a(k) != b(k)) return false else k += 1
    }
    // k = n, so a[0..n) = b[0..n)
    return true
  }

  /** Calculates n such that the array recurs with period n
    * Pre: length of arrays is non-negative */
  def recurs(a : Array[Char]) : Int = {
    // Invariant : a[0..N-i) == a[i..N) && 1 <= i <= N
    // Variant : N - i
    // N - size of a
    var i = 1
    while(i <= a.size){
      var array1 = new Array[Char](i); var array2 = new Array[Char](i)

      var j = 0
      // Invariant : array1[0..j] = a[0..j] && array2[0..j] = a[i..N-1]
&& 0 <= j < i
      // Variant : i - j
      while(j < a.size - i - 1 && j < i){
        array1(j) = a(j)
        array2(j) = a(j+i)
        j += 1
      }

      // array1 = array2, so i is the smallest number that the array
recurs with
      if(search(array1, array2)) return i
      i += 1
    }
    return i
  }
```

```
  def main(args : Array[String]) = {
    if(args.size != 1) println("Please enter a non-empty string")
    else{
      val a = args(0).toArray
      println(recurs(a))
    }

    assert(recurs("olfrudolfrudolfrudolf".toArray) == 6)
    assert(recurs("rudolfrudolfrudolf".toArray) == 6)
    assert(recurs("phonebook".toArray) == 8)
    assert(recurs("phonephobook".toArray) == 11)
  }
}
```

Question 7:

```
object Exists{
  /** Returns whether there is an element that satisfies some rule
    * Pre: N >= 0 */
  def exist(p : Int => Boolean, N : Int) : Boolean = {
    // Invariant : is p(i) true for any number in [0..i] ^ 0 <= i < N
    // Variant : N - i
    var i = 0
    while(i < N){
      // If the condition is true for i, return true and stop the loop
      if(p(i)) return true
      i += 1
    }
    // i = N, so all p(i) are false => does not exist a number that
satisfies the rule
    return false
  }

  def main(args : Array[String]) = {
    if(args.size != 1){
      println("Wrong number of arguments")
    }
    else{
      val input = args(0).toInt
      val threedigits : (Int => Boolean) = i => {i >= 100 && i < 1000}
      println(exist(threedigits, input))

      assert(exist(threedigits, 100) == false)
```

```
      assert(exist(threedigits, 999) == true)
      assert(exist(threedigits, 101) == true)
      assert(exist(threedigits, 1000) == true)
    }
  }
}
```

Question 8:

a)

```
object Fractions{
  /** Expresses a fraction as a sum of distinc reciprocals
    * Pre: 0 < p < q */
  def fractions(p : Int, q : Int) : Int = {
    // Invariant : q <= m * p && 2 <= m <= q
    // Variant : q - m
    var m = 2
    while(q > (m * p) && m <= (q/p)){
      m += 1
    }
    // q <= m * p, so m is the smallest integer that satisfies
    // 1/m <= p/q
    m
  }

  def main(args : Array[String]) = {
    if(args.size != 2) println("Wrong number of arguments")
    else{
      val a = args(0).toInt
      val b = args(1).toInt
      if(a > 0 && b > 0) println(fractions(a, b))
      else println("Please enter non-negative numbers")
    }

    assert(fractions(5, 6) == 2)
    assert(fractions(2, 35) == 18)
    assert(fractions(1, 10) == 10)
  }
}
```

b)

```
object Fractions{
  /** Expresses a fraction as a sum of distinc reciprocals
```

```
  * Pre: 0 < p < q */
def fractions(p : Int, q : Int) : Int = {
  // Invariant : q <= m * p ^ 2 <= m <= q
  // Variant : q - m
  var m = 2
  while(q > (m * p) && m <= (q/p)){
    m += 1
  }
  // q <= m * p, so m is the smallest integer that satisfies
  // 1/m <= p/q
  m
}

/** Returns an array with all the distinct denominators
  * Pre: 0 < p < q */
def sumFractions(p : Int, q : Int) : Array[Int] = {
  // Invariant : p1/q1 = p/q - 1/a(i)
  //        a(i) = fractions(p, q) ^ 0 <= p1/q1 < p/q ^ 0 <= i
  // Variant : p1/q1
  var a = new Array[Int](10) // Assuming we won't need more than 10
  var i = 1
  a(0) = fractions(p, q)
  var p1 = p * (a(0) / gcd(a(0), q)) - q / (gcd(a(0), q))
  var q1 = a(0) * q / gcd(a(0), q)
  while(p1 != 0){
    a(i) = fractions(p1, q1)
    // Introduce this new variable to calculate q1 with the old value
    // val temporaryP1 = p1
    p1 = p1 * (a(i) / gcd(a(i), q1)) - q1 / (gcd(a(i), q1))
    q1 = a(i) * q1 / gcd(a(i), q1)
    i += 1
  }
  // p1 = 0, so all the possible fractions are calculated
  a
}

/** Calculate the greates common devisor of m and n
  * Pre: m >= 0, n >= 0 */
def gcd(m : Int, n : Int) : Int = {
  var a = m; var b = n; var r = a; var q = 0
  //Invariant I : a = qb + r && 0 <= r < b
  while(r != 0){
```

```
      //I
      q = a / b
      r = a - q * b
      if(r != 0){
        a = b
        b = r
      }
    }
    // I and r = 0, so a = qb
    b
  }

  def main(args : Array[String]) = {
    var array = new Array[Int](10)
    if(args.size != 2) println("Wrong number of arguments")
    else{
      val a = args(0).toInt
      val b = args(1).toInt
      if(a > 0 && b > 0){
        array = sumFractions(a, b)
      }
      else println("Please enter non-negative numbers")
    }

    var i = 0
    while(i < 10){
      if(array(i) != 0) print("1/" + array(i) + " ")
      i += 1
    }

    //assert(fractions(5, 6) == 2)
    //assert(fractions(2, 35) == 18)
    //assert(fractions(1, 10) == 10)
  }
}
```

Question 9:

```
object Log3{
  /** Calculates floor log3 of a number using linear search
    * Pre: n >= 1 */
  def log(n : Int) : Int = {
    require(n >= 1)
```

```
    // Invariant : m <= n && i = log(3)(m) ^ 0 <= i ^ 1 <= m <= n
    // Variant : n - m
    var i = 0; var m = 1
    while(m * 3 <= n){
      i += 1
      m *= 3
    }
    // (m + 1) * 3 > n and i = log(3)(m), so i is floor of log(3)(n)
    i
  }

  def main(args : Array[String]) = {
    if(args.size != 1) println("Wrong number of arguments")
    else{
      val input = args(0).toInt
      if(input < 1) println("Please enter a number larger or equal to
1")
      else println(log(input))
    }

    assert(log(27) == 3)
    assert(log(26) == 2)
    assert(log(1) == 0)
    assert(log(100) == 4)
  }
}
```

Question 10:

```
object Polynomial{
  /** Calculates the value of a polynomial */
  def polynomial(a : Array[Double], x : Double) : Double = {
    // Invariant : total = x * (total + a(i)) ^ 0 < i < a.size
    //             total = total + a(0)
    // Variant : i
    var i = a.size - 1; var total : Double = 0.0
    while(i > 0){
      total = (total + a(i)) * x
      i -= 1
    }
    // i = 0, so total = sum[a(j) * x ^ j] for 0 < j < a.size
    // we have to add a(0)
    total + a(0)
```

```
  }

  def main(args : Array[String]) = {
    val s = args.size
    val array = new Array[Double](s-1)

    // Invariant : a[0..i] = args[0..i] for 0 <= i < args.size - 1
    // Variant : args.size - 1 - i
    var i = 0
    while(i < s - 1){
      array(i) = args(i).toDouble
      i += 1
    }

    val x = args(s-1).toDouble
    println(polynomial(array, x))

    val array1 = Array(3.7, 5.1, 6.2, 7)
    assert(263.8 - polynomial(array1, 3) <= 0.000001)
  }
}
```