Problem Sheet 2

Question 1:

a) The function `def swap(x: Int, y: Int) = { val t = x; x = y; y = t }` does the following. For a tuple input `(x,y)`, the function `swap(x,y)` changes the value of the global variables `x` and `y`.

b) The function

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

swaps the value of the i-th entry and the j-th entry.

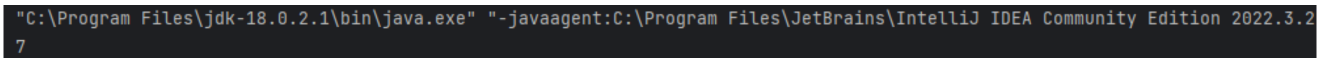
Question 2:

```
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)
}
```

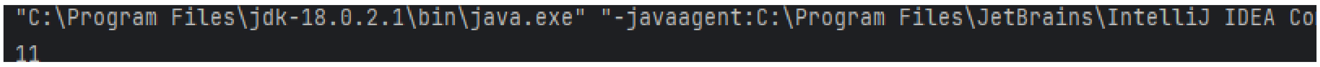
The output of this program is 7. If instead it was `y + nasty(x)`, the program would actually output 11.

Here is some output code:



```
"C:\Program Files\jdk-18.0.2.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.3.2
7
```

And with `y + nasty(x)`:



```
"C:\Program Files\jdk-18.0.2.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Co
11
```

Question 3:

I will be using the provided method `sameElements` in `Array` to test whether two lists contain the same contents:

```
object Array {
        // ...
        def sameElements(xs: Array[AnyRef], ys: Array[AnyRef]): Boolean =
          (xs eq ys) ||
          (xs.length == ys.length) && {
            var i = 0
            while (i < xs.length && xs(i) == ys(i)) i += 1
            i >= xs.length
          }

        // ...
```

Given that the type `Int` has already defined `==` as an acceptable operator, this should be okay.

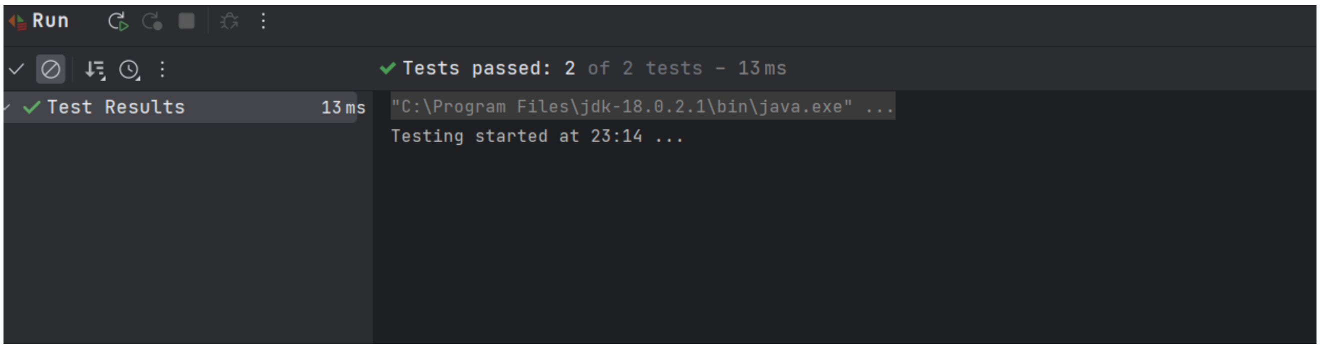
Here are the tests:

```
class Question3Test extends AnyFunSuite {

  test("Following list (2,6,4,11,5,5,7)") {
    val digits = Array(2, 6, 4, 11, 5, 5, 7)
    val sortedDigits = Array(2, 4, 5, 5, 6, 7, 11)
    assert(sortedDigits.sameElements(digits.sorted))
  }

  test("Following list (2,6,4,1,3,5,7)") {
    val digits = Array(2,6,4,1,3,5,7)
    val sortedDigits = Array(1,2,3,4,5,6,7)
    assert(digits.sorted.sameElements(sortedDigits))
  }
}
```

Here are the test results:



```
✓ Tests passed: 2 of 2 tests – 13 ms
✓ Test Results    13 ms
"C:\Program Files\jdk-18.0.2.1\bin\java.exe" ...
Testing started at 23:14 ...
```

Question 4:

a) Take the very basic example of the following function

```
def count(): Unit = {
  var x = 0.0;  val n = 0.9
  while (x < n) {
    x += 0.1
    println("Current Value of x: " + x)
  }
}
```

Will happen to print all the numbers from 0.1 all the way to 0.999999... since x is actually rounded down to 0.7999... during the supposedly last iteration of the loop, where it should've been actually halted then. Rounding errors can cause inequality checks to pass when they were intended to fail.

b) Time may be inaccurate by the margin of how large timestep is. So, if `time` will be off by `time +- 1*timeStep`.

c) For this solution, we well need to define a constant `SMALL = 0.00000001`, an arbitrarily small non-zero floating point constant.

Now, to ensure that postcondition is correct, here is what we do:

$$\begin{aligned} \text{time} &< \text{timeEnd} \\ &\vdots \\ 0 &< \text{timeEnd} - \text{time} \end{aligned}$$

but since `timeEnd - time` will be greater than zero, even when they should be equal, we can replace that zero with our `SMALL` constant:

Our invariant therefore becomes: $I \mathrel{\hat{=}} \text{SMALL} < \text{timeEnd} - \text{time}$.

Now in this case, `time` will only be inaccurate by the absolute uncertainty of our `SMALL` constant. If we either make it too large where its redundant, then it won't help. But if we make it too small where it doesn't register any rounding errors, then the error will return to being `+- 1*timeStep`.

Question 5

```
class Question5Test extends AnyFunSuite {

  val tester = new Question5()

  /**
   *Since the pattern is not a substring of   * our string, we should return
false from   * the function. If we don't, this test   * will fail.*/  test("Part
a - if found was initialised to be true") {
    val string_a = "i do exist inside string_a, because i am myself!"
    val patt = "i don't exist inside string_a"

    assert(!tester.search(patt.toCharArray, string_a.toCharArray))
  }
  /**
   * This test makes sure that even if the pattern is found   * at the end of the
string, it is still accounted   * for and returned true   */  test("Part b -
ending the initial loop too early") {
    val string_a = "hey, I don't exist here, i exist only at the end"
    val patt = "i exist only at the end"

    assert(tester.search(patt.toCharArray, string_a.toCharArray))
  }

  /**
   * For this test, we shall test this by giving the patt.size > string_a.size
* they should return false since the empty an empty string   * doesn't contain
any string length*/  test("Part c - ending the initial loop too late") {
    val string_a = ""
    val patt = "a"

    assert(!tester.search(patt.toCharArray, string_a.toCharArray))
    // notice how this search actually gives an ArrayIndexOutOfBounds exception
when run
  }

  /**
   * For this test, I will be keeping everything the same except for   * the
first character of the pattern   */  test("Part d - initialising k at 1 instead
of 0") {
    val string_a = "i do contain bigger"
    val patt = "digger"

    assert(!tester.search(patt.toCharArray, string_a.toCharArray))
  }

  /**
   * For this test, we will essentially be running out of   * bounds since no
such element for the pattern exists*/  test("Part e - changing the second test
from k<K to k<=K") {
    val string_a = "hello"
    val patt = "hel"
    assert(tester.search(patt.toCharArray, string_a.toCharArray))
  }

  /***
   * For this test, changing k == K to k >= K doesn't really   * alter the
correctness of the algorithm, but does   * change the logic of the program   */
test("Part f - changing the found = (K == k) to found = (k >= K)") {
    // i couldn't find a test that would fail with this change
  }
```

```
}
```

with `Question5` class being:

```
class Question5 {

  def search(patt: Array[Char], line: Array[Char]): Boolean = {
    val K = patt.size;
    val N = line.size
    // Invariant: I: found = (line[i..i+K) = patt[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) = patt[0..K)
      // Invariant: line[j..j+k) = patt[0..k)
      var k = 0

      while (k < K && line(j + k) == patt(k)) k = k + 1
      found = (k == K)
      j = j + 1

    }
    // I && (j=N-K+1 || found)
    // found = ( line[i..i+K) = patt[0..K) for some i in [0..N-K+1) )    found

  }

}
```

Question 6:

```
def findSmallestPeriod(s: Array[Char]): Int = {
  val N = s.length
  // Start with the smallest possible period, which is 1
  for (n <- 1 to N) {
    var isPeriod = true
    // Check if s recurs with period n
    for (i <- 0 until N - n if isPeriod) {
      if (s(i) != s(i + n)) {
        isPeriod = false
      }
    }
    // If s recurs with period n, return n
    if (isPeriod) return n
  }
  // If no smaller period is found, s recurs with period N
  N
}
```

Question 7:

```
def exists(p: Int => Boolean, N:Int) : Boolean = {
        var i = 0
        // Invariant: p(j) is false for all j in [0..i)
        while (i < N) {
                if (p(i)) true
                i += 1
        }
        false
}
```

Preconditions: `i = 0` and true over the range `[0..i)`.

Postcondition: `i = N` and true over the range `[0..N)`

Question 8:

Since $1/m \leq p/q \to m \geq q/p$. Essentially, the smallest m which satisfies this equation works as our m. Here is the Scala expression for it:

```
val p : Int = ... // some integer p
val q : Int = ... // some integer q

// Compute m
val m: Int = (q.toDouble / p).ceil.toInt
```

b) Here is the iterated version of this:

```
def expressAsSumOfDistinctReciprocals(p: Int, q: Int): Array[Int] = {
  var remainingP = p
  var remainingQ = q
  var d = Array[Int]()

  while (remainingP > 0) {
    val m = (remainingQ.toDouble / remainingP).ceil.toInt
    d = d :+ m // Append m to the array d
    // Update remainingP and remainingQ to reflect the new fraction p'/q' = p/q -
1/m
    remainingP = remainingP * m - remainingQ
    remainingQ = remainingQ * m
    // Reduce the fraction p'/q' to its simplest form
    val gcd = BigInt(remainingP).gcd(BigInt(remainingQ)).toInt
```

```
      remainingP /= gcd
      remainingQ /= gcd
    }

    d
  }
```

c) To prove that the loop always terminates, what we need to show here is actually that the value of p reduces to zero. Given that m is selected to be $\lceil q/p \rceil$, we ensure that the next values of `q` and `p` are bound to be much smaller since $(p \cdot m - q)/(q \cdot m) < q/p$. As this is done recursively, due to the nature of the GCD function, we know that at some point, q is bound to become zero.

d) Each subtraction $p/q - 1/m_i$ yields a new fraction smaller than the original. This is because we are effectively removing a portion of the numerator's contribution to the next fraction, thereby reducing its value. Now, for the next iteration, m_{i+1} to be valid, it must be chosen that $1/m_{i+1} \leq p'/q'$ is smaller than the original p/q due to the subtraction. This must mean that $m_i \geq p/q > p'/q'$, and since $m_{i+1} < p/q$, this means that $m_i > m_{i+1}$ by transitivity of the inequality. This therefore proves that the sequence is strictly increasing.

Question 9:

```
def floor_log3(x : Int) : Int = {
        require(x>=1)

        var product = 1
        y = - 1

        while (product <= x) {
                product = product * 3
                y += 1
        }
        return y
 }
```

preconditions: `product = 1`, `y = -1` since this ensures that the first multiplication brings it to the base level of 0, since $\log_3 1 = 0$.

invariant: $3^y \leq x < 3^{y+1}$

postconditions: $y = \lfloor \log_3 x \rfloor$.

Question 10:

```
def eval (a: Array[Double], x: Double) : Double = {
        var evaluated = 0
        for (i <- 0 until a.size) {
                evaluated += a(i)*(x)^(i)
        }
        evaluated
}
```