Министерство науки и высшего образования Российской Федерации
Национальный научно-исследовательский университет ИТМО
Факультет программной инженерии и компьютерной техники

Лабораторная работа №1
по дисциплине
“Тестирование программного обеспечения”.

Работу выполнил:
Афанасьев Кирилл Александрович.
студент группы P3306.
Преподаватель:
Харитонова Анастасия Евгеньевна.

Санкт-Петербург, 2025

Содержание

Задание

1. Для указанной функции провести модульное тестирование разложения функции в степенной ряд. Выбрать достаточное тестовое покрытие.
Вариант: *arccos(x)*

2. Провести модульное тестирование указанного алгоритма. Для этого выбрать характерные точки внутри алгоритма, и для предложенных самостоятельно наборов исходных данных записать последовательность попадания в характерные точки. Сравнить последовательность попадания с эталонной.
Вариант: *Программный модуль для работы с Фибоначчиевой кучей (Logical Representation, http://www.cs.usfca.edu/~galles/visualization/FibonacciHeap.html)*

3. Сформировать доменную модель для заданного текста. Разработать тестовое покрытие для данной доменной модели
Вариант: *"Золотое Сердце" плыл через космическую ночь, теперь уже на обычном фотоновом двигателе. Четыре человека, составлявшие его экипаж, чувствовали себя неуютно, зная, что они вместе не по собственной воле и не по простому совпадению, а по странному физическому принципу -- как будто отношения между людьми подчиняются тем же законам, что отношения между атомами и молекулами.*

Исходный код

Github: https://github.com/Zerumi-ITMO-Related/test1_070225_666666

Листинг задания 1:
Arccos.kt:

```kotlin
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.assertThrows
import org.junit.jupiter.params.ParameterizedTest
import org.junit.jupiter.params.provider.CsvFileSource
import kotlin.math.PI
import kotlin.math.sqrt
import kotlin.math.pow
import kotlin.test.assertEquals

// arccos(x) = pi/2 - arcsin(x)
fun arccos(x: Double): Double =
    (PI / 2 - (x + x.pow(3) / 6 + 3 * x.pow(5) / 40 + 15 * x.pow(7) / 336 +
            105 * x.pow(9) / 3456 + 945 * x.pow(11) / 42240 +
            10395 * x.pow(13) / 599040 + 135135 * x.pow(15) / 9676800 +
            2027025 * x.pow(17) / 175472640 + 34459425 * x.pow(19) /
3530096640))
        .also { require(x in -1.0..1.0) }


class ArccosTest {
    @ParameterizedTest
    @CsvFileSource(resources = ["/data.csv"], numLinesToSkip = 1)
    fun `testArccos should pass tests from coverage table`(
        testName: String,
        input: Double,
        expected: Double,
        tolerance: Double
    ) {
        assertEquals(
            expected,
            arccos(input),
            tolerance,
            "$testName failed, actual: ${arccos(input)}"
        )
    }

    @Test
    fun `testArccos should return 3,141 when x is -1`() {
        assertEquals(PI, arccos(-1.0), 0.25)
    }

    @Test
    fun `testArccos should return 2,094 when x is -0,5`() {
        assertEquals(2 * PI / 3, arccos(-0.5), 1e-2)
    }
```

```
    @Test
    fun `testArccos should return 1,5708 when x is 0`() {
        assertEquals(PI / 2, arccos(0.0), 1e-2)
    }

    @Test
    fun `testArccos should return 1,047 when x is 0,5`() {
        assertEquals(PI / 3, arccos(0.5), 1e-2)
    }

    @Test
    fun `testArccos should return 0,785 when x is 0,707`() {
        assertEquals(PI / 4, arccos(sqrt(2.0) / 2), 1e-2)
    }

    @Test
    fun `testArccos should return 0 when x is 1`() {
        assertEquals(0.0, arccos(1.0), 0.25)
    }

    @Test
    fun `testArccos should throw exception when x is greater than 1`() {
        assertThrows<IllegalArgumentException> { arccos(1.1) }
    }

    @Test
    fun `testArccos should throw exception when x is less than -1`() {
        assertThrows<IllegalArgumentException> { arccos(-1.1) }
    }
}
```

Листинг FibonacciHeapTest.kt:

```
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.assertTimeoutPreemptively
import w01fe.fibonacci_heap.FibonacciHeap
import w01fe.fibonacci_heap.insert
import java.time.Duration
import kotlin.test.assertEquals
import kotlin.test.assertNotNull

class FibonacciHeapTest {
    @Test
    fun `testHeap sequentially inserting 5 minimums`() {
        assertTimeoutPreemptively(Duration.ofSeconds(5)) {
            val heap = FibonacciHeap()
            for (i in 10..15) heap.insert(i)
            val min = heap.min()
            assertNotNull(min)
            assertEquals(10, min.key as Int)
        }
    }

    @Test
    fun `testHeap sequentially inserting 1000000 minimums`() {
        assertTimeoutPreemptively(Duration.ofSeconds(5)) {
```

```
            val heap = FibonacciHeap()
            val nums = 1..1000000
            for (i in nums) heap.insert(i)
            for (i in nums) {
                val min = heap.min()
                assertNotNull(min)
                assertEquals(i, min.key as Int)
                heap.removeMin()
            }
        }
    }

    @Test
    fun `testHeap reversed inserting 1000 minimums`() {
        assertTimeoutPreemptively(Duration.ofSeconds(5)) {
            val heap = FibonacciHeap()
            val nums = 1000 downTo 1 step 2
            for (i in nums) heap.insert(i)
            for (i in nums.reversed()) {
                val min = heap.min()
                assertNotNull(min)
                assertEquals(i, min.key as Int)
                heap.removeMin()
            }
        }
    }

    @Test
    fun `testHeap shuffled inserting`() {
        assertTimeoutPreemptively(Duration.ofSeconds(5)) {
            val heap = FibonacciHeap()
            val nums = 1..10000
            for (i in nums.shuffled()) heap.insert(i)
            for (i in nums) {
                val min = heap.min()
                assertNotNull(min)
                assertEquals(i, min.key as Int)
                heap.removeMin()
            }
        }
    }

    @Test
    fun `testHeap inserting similar elements`() {
        assertTimeoutPreemptively(Duration.ofSeconds(5)) {
            val heap = FibonacciHeap()
            val nums = arrayOf(3, 3, 3, 4, 4, 4)
            for (i in nums) heap.insert(i)
            for (i in nums) {
                val min = heap.min()
                assertNotNull(min)
                assertEquals(i, min.key as Int)
                heap.removeMin()
            }
        }
    }

    @Test
    fun `testHeap fibonacci algorithm to build child and left-right elements`() {
```

```
        assertTimeoutPreemptively(Duration.ofSeconds(5)) {
            val heap = FibonacciHeap()
            val nums = arrayOf(3, 1, 2)
            for (i in nums) heap.insert(i)
            val min = heap.min()
            // 1 should be min
            assertNotNull(min)
            assertEquals(1, min.key as Int)

            // 2 should be left, 3 - right
            val left = min.left
            val right = min.right
            assertNotNull(left)
            assertNotNull(right)
            assertEquals(2, left.key as Int)
            assertEquals(3, right.key as Int)

            heap.insert(0)
            val zeroShouldBeMin = heap.min()
            assertNotNull(zeroShouldBeMin)
            assertEquals(0, zeroShouldBeMin.key as Int)

            heap.removeMin()

            // 2 should be left, 1 should be min with child 3
            val nextMin = heap.min()
            assertNotNull(nextMin)
            assertNotNull(nextMin.left)
            assertNotNull(nextMin.child)
            assertEquals(1, nextMin.key as Int)
            assertEquals(2, nextMin.left.key as Int)
            assertEquals(3, nextMin.child!!.key as Int)
        }
    }
}
```

Листинг GoldenHeart.kt:

```
import org.junit.jupiter.api.Assertions.assertTrue
import org.junit.jupiter.api.Test
import kotlin.math.roundToInt
import kotlin.math.sqrt
import kotlin.test.assertEquals
import kotlin.test.assertNotEquals


// "Золотое Сердце" плыл через космическую ночь, теперь уже на обычном фотоновом
двигателе.
// Четыре человека, составлявшие его экипаж, чувствовали себя неуютно, зная, что
они вместе
// не по собственной воле и не по простому совпадению, а по странному физическому
принципу --
// как будто отношения между людьми подчиняются тем же законам, что отношения между
атомами
// и молекулами.

data class State<T>(
    val state: T,
    val test: (Boolean) -> ConditionalState<T>
)
```

```
data class ConditionalState<T>(
   val state: T,
   val ifTrue: ((T) -> T) -> ConditionalState<T>,
   val ifFalse: ((T) -> T) -> ConditionalState<T>,
   val then: () -> State<T>,
   val collectResult: () -> T
)

fun <T> initState(state: T): State<T> = State(
   state = state,
   test = { cond ->
       fun createConditionalState(updatedState: T): ConditionalState<T> =
ConditionalState(
           state = updatedState,
           ifTrue = { function ->
               if (cond) createConditionalState(function(updatedState)) else
createConditionalState(updatedState)
           },
           ifFalse = { function ->
               if (!cond) createConditionalState(function(updatedState)) else
createConditionalState(updatedState)
           },
           then = { initState(updatedState) },
           collectResult = { updatedState }
       )
       createConditionalState(state)
   }
)

data class Spaceship(
   val name: String,
   val mass: UInt,
   val engine: Engine,
)

data class Engine(
   val mass: UInt, val power: UInt
)

data class Flight(
   val spaceship: Spaceship,
   var gas: Boolean,
   var brake: Boolean,
   var gasInitTime: Long,
   var brakeInitTime: Long,
   var speed: Double,
)

data class FlightControl(
   val gasInit: (Long) -> Unit,
   val gasStop: (Long) -> Unit,
   val brakeInit: (Long) -> Unit,
   val brakeStop: (Long) -> Unit,
   val speed: (Long) -> Double,
)

fun initFlight(
   spaceship: Spaceship,
): FlightControl {
```

```
    val flight = Flight(spaceship, gas = false, brake = false, gasInitTime = 0,
brakeInitTime = 0, speed = 0.0)

    return FlightControl(gasInit = { timestamp ->
        flight.gas = true
        flight.gasInitTime = timestamp
    }, gasStop = { timestamp ->
        if (!flight.gas) return@FlightControl

        flight.gas = false

        val time = timestamp - flight.gasInitTime
        val acceleration = sqrt(
            spaceship.engine.power.toDouble() / (2.0 * (spaceship.mass +
spaceship.engine.mass).toDouble() * time)
        )
        flight.speed += acceleration * time
    }, brakeInit = { timestamp ->
        flight.brake = true
        flight.brakeInitTime = timestamp
    }, brakeStop = { timestamp ->
        if (!flight.brake) return@FlightControl

        flight.brake = false

        val time = timestamp - flight.brakeInitTime
        val brake = (time / 250.0).roundToInt()
        flight.speed = (flight.speed - brake).coerceAtLeast(0.0)
    }, speed = { timestamp ->
        initState(flight.speed)
            .test(flight.gas)
            .ifTrue { speed ->
                val time = timestamp - flight.gasInitTime
                val acceleration = if (time == 0L) 0.0 else sqrt(
                    spaceship.engine.power.toDouble() / (2.0 * (spaceship.mass +
spaceship.engine.mass).toDouble() * time)
                )
                speed + acceleration * time
            }
            .then()
            .test(flight.brake)
            .ifTrue { speed ->
                val time = timestamp - flight.brakeInitTime
                val brake = (time / 250.0).roundToInt()
                (speed - brake).coerceAtLeast(0.0)
            }
            .collectResult()
    })
}

fun main() {
    val goldenHeart = Spaceship(
        name = "Golden Heart", mass = 100000u, engine = Engine(
            mass = 10000u, power = 99999u
        )
    )

    val flight = initFlight(goldenHeart)
    val initFlightTime = System.currentTimeMillis()
```

```
    Thread {
        while (true) {
            val timestamp = System.currentTimeMillis() - initFlightTime
            println("Current speed ($timestamp): ${flight.speed(timestamp)}m/s")
            Thread.sleep(250)
        }
    }.start()

    Thread {
        while (true) {
            val reader = System.`in`.reader()
            val char = reader.read().toChar()
            val registerTime = System.currentTimeMillis() - initFlightTime
            when (char) {
                'w' -> flight.brakeStop(registerTime).also {
flight.gasInit(registerTime) }
                's' -> flight.gasStop(registerTime).also {
flight.brakeInit(registerTime) }
                'q' -> flight.gasStop(registerTime).also {
flight.brakeStop(registerTime) }
            }
        }
    }.start()
}
class GoldenHeartTest {
    @Test
    fun `testState single ifTrue should be executed when testing true`() {
        val state = initState(10)
        val result = state.test(true)
            .ifTrue { 2 * it }
            .collectResult()

        assertEquals(20, result)
    }

    @Test
    fun `testState single ifFalse should be executed when testing false`() {
        val state = initState(10)
        val result = state.test(false)
            .ifFalse { 2 * it }
            .collectResult()

        assertEquals(20, result)
    }

    @Test
    fun `testState multiple ifTrue should execute all of them`() {
        val state = initState(10)
        val result = state.test(true)
            .ifTrue { 2 * it }
            .ifTrue { 3 * it }
            .collectResult()

        assertEquals(60, result)
    }

    @Test
    fun `testState multiple ifFalse should execute all of them`() {
```

```
        val state = initState(10)
        val result = state.test(false)
            .ifFalse { 2 * it }
            .ifFalse { 3 * it }
            .collectResult()

        assertEquals(60, result)
    }

    @Test
    fun `testState multiple ifTrue and ifFalse should execute only true branch when
testing true`() {
        val state = initState(10)
        val result = state.test(true)
            .ifTrue { it }
            .ifFalse { 2 * it }
            .ifTrue { 3 * it }
            .ifFalse { 4 * it }
            .collectResult()

        assertEquals(30, result)
    }

    @Test
    fun `testState multiple ifTrue and ifFalse should execute only false branch when
testing false`() {
        val state = initState(10)
        val result = state.test(false)
            .ifTrue { it }
            .ifFalse { 2 * it }
            .ifTrue { 3 * it }
            .ifFalse { 4 * it }
            .collectResult()

        assertEquals(80, result)
    }

    @Test
    fun `testState then() should allow to construct new state`() {
        val state = initState(10)
        val result = state.test(true)
            .ifTrue { 2 * it }
            .ifFalse { 3 * it }
            .then().test(false)
            .ifTrue { 2 * it }
            .ifFalse { 3 * it }
            .collectResult()

        assertEquals(60, result)
    }

    @Test
    fun `testState similar states should be compared`() {
        val state = initState(10)
        val state2 = initState(10)

        assertEquals(state.state, state2.state)
        assertNotEquals(state.test, state2.test)
```

```
        val state3 = state2.test(true)
            .ifTrue { it }
            .then()

        assertEquals(state.state, state3.state)
        assertNotEquals(state.test, state3.test)
    }

    @Test
    fun `testFlight gas initialization`() {
        val spaceship = Spaceship("Apollo", 1000u, Engine(500u, 2000u))
        val flightControl = initFlight(spaceship)

        flightControl.gasInit(1000L)
        assertTrue(flightControl.speed(1000L) == 0.0)
    }

    @Test
    fun `testFlight gas should increase speed`() {
        val spaceship = Spaceship("Apollo", 1000u, Engine(500u, 2000u))
        val flightControl = initFlight(spaceship)

        flightControl.gasInit(1000L)
        flightControl.gasStop(2000L)

        val expectedSpeed = sqrt(2000.0 / (2.0 * 1500.0 * 1000)) * 1000
        assertEquals(expectedSpeed, flightControl.speed(2000L), 0.01)
    }

    @Test
    fun `testFlight brake initialization`() {
        val spaceship = Spaceship("Apollo", 1000u, Engine(500u, 2000u))
        val flightControl = initFlight(spaceship)

        flightControl.brakeInit(3000L)
        assertTrue(flightControl.speed(3000L) == 0.0)
    }

    @Test
    fun `testFlight brake should decrease speed`() {
        val spaceship = Spaceship("Apollo", 1000u, Engine(500u, 2000u))
        val flightControl = initFlight(spaceship)

        flightControl.gasInit(1000L)
        flightControl.gasStop(2000L)
        val initialSpeed = flightControl.speed(2000L)

        flightControl.brakeInit(3000L)
        flightControl.brakeStop(4000L)
        val brakeEffect = (1000 / 250.0).roundToInt()

        assertEquals((initialSpeed - brakeEffect).coerceAtLeast(0.0),
flightControl.speed(4000L), 0.01)
    }
}
```

Выводы

Во время выполнения данной лабораторной работы я ознакомился с основами тестирования ПО на примере модельного тестирования. Я реализовал модульное тестирование функции arccos(x) при разложении в степенной ряд, обеспечил достаточное тестовое покрытие, охватывающее ключевые случаи работы функции. Модульное тестирование проводилось по принципу сравнения с эталонным значением. Кроме того, я провел тестирование программного модуля для работы с Фибоначчиевой кучей. Здесь я ознакомился с возможностью задавать таймаут для теста. Также я разработал доменную модель с применением практик функционального программирования и провел тестирование для данной модели, что позволило убедиться в её корректности и соответствии исходным требованиям. Также автор лабораторной работы считает, что функциональный код намного проще тестировать из-за отсутствия side-cause и side-effects. Таким образом, я ознакомился с возможностями библиотеки JUnit, принципами модульного тестирования и составлением тестового покрытия.