BTI4202 - Exercise Sheet 3

Jan Fuhrer, Andy Bigler, Joel Häberli

April 19, 2023

Contents

1 Miller-Rabin Primality Test

```kotlin
import java.math.BigInteger
import java.util.*

fun main() {
    MillerRabin()
}

/**
 * @param k number of rounds of testing
 * @param range range of numbers to test for primality
 */
class MillerRabin(private val k: Int = 64, range: IntRange = 1..1000) {
    companion object {
        val ZERO: BigInteger = BigInteger.ZERO
        val ONE: BigInteger = BigInteger.ONE
        val TWO: BigInteger = BigInteger("2")
    }

    init {
        require(k > 0) { "k must be greater than 0" }
        require(range.first > 0) { "range must be greater than 0" }

        val primes = mutableListOf<Int>()
        for (i in range) {
            if (isProbablePrime(i.toBigInteger())) {
                primes.add(i)
            }
        }
        println(primes)
    }

    /**
     * @param n n > 0, an odd number to be tested for primality
     * @return true if number is probably prime, false if number is definitely composite
     */
    private fun isProbablePrime(n: BigInteger): Boolean {
        if (n.compareTo(ONE) == 0) // 1 is not prime
            return false
        if (n.compareTo(TWO) == 0) // 2 is prime
            return true
        if (n.mod(TWO) == ZERO) // Even numbers other than 2 are not prime
            return false

        // First initialize two variables "s" and "d". "s" is a counter for the number
        // of times "d" can be divided by 2, and "d" is initialized to "n-1".
        var s = 0
        var d = n.subtract(ONE)

        // Each time "d" is divided by 2, "s" is incremented by 1.
        // This step is used to write
        // "n-1" as "2^s * d", where "d" is an odd number.
        while (d.mod(TWO) == ZERO) {
            s++
            d = d.divide(TWO)
        }
```

```kotlin

        // Perform "k" iterations of the Miller-Rabin test. In each iteration,
        // a random number "a" is chosen between 2 and "n-1".
        for (i in 0 until k) {
            val a = uniformRandom(TWO, n.subtract(ONE))
            // Checks whether "a" is a witness for the composites of "n".
            // If "x" is equal to 1 or "n-1", then "a" is not a witness.
            var x = a.modPow(d, n)
            if (x == ONE || x == n.subtract(ONE)) continue
            // Repeatedly square "x" modulo "n" "r" times, where "r" is less
            // than "s". If "x" ever becomes equal to 1 during this process,
            // then "a" is a witness for the composites of "n"
            var r = 0
            while (r < s) {
                x = x.modPow(TWO, n)
                if (x == ONE) return false
                if (x == n.subtract(ONE)) break
                r++
            }
            // None of the steps made x equal n-1. Therefore, "a" is a witness
            if (r == s) return false
        }
        return true
    }

    private fun uniformRandom(bottom: BigInteger, top: BigInteger): BigInteger {
        val randomNumber = Random()
        var result: BigInteger
        do {
            result = BigInteger(top.bitLength(), randomNumber)
        } while (result < bottom || result > top)
        return result
    }
}
```

2 Algorithm and Extended Algorithm of Euclid

Task Euclid(48, 174) = 6

a	b	r
48	174	30
30	48	18
18	30	12
12	18	6
6	12	0

Task ExtEuclid(48, 174) = $(6, 11, -3)$

a	b	r	q		d	X	Y
48	174	30	3		6	**11**	**-3**
30	48	18	1		6	-3	2
18	30	12	1		6	2	-1
12	18	6	1		6	-1	1
6	12	0	2	$\rightarrow$	6	1	0

Task MultInv(48, 127) = 45

a	b	r	q	d	X	Y
48	127	31	2	1	**45**	-17
31	48	17	1	1	-17	11
17	31	14	1	1	11	-6
14	17	3	1	1	-6	5
3	14	2	4	1	5	-1
2	3	1	1	1	-1	1
1	2	0	2	1	1	0

3 Square-and-Multiply Algorithm

Task modExp(3, 37, 13) = 3 mod 13

$$
\begin{aligned}
3^{37} &\equiv 3 \cdot 3^{36} && \textit{multiply} \\
&\equiv 3 \cdot 9^{18} && \textit{square} \\
&\equiv 3 \cdot 81^{9} \equiv 3 \cdot 3^{9} && \textit{square} \\
&\equiv 3 \cdot 3 \cdot 3^{8} && \textit{multiply} \\
&\equiv 9 \cdot 9^{4} && \textit{square} \\
&\equiv 9 \cdot 81^{2} \equiv 9 \cdot 3^{2} \equiv 9 \cdot 9 \equiv 81 && \textit{square} \\
&\equiv 3 \pmod{13}
\end{aligned}
\tag{1}
$$

4 Exponentiation with Negative Exponents

Modular exponentiation can be performed with a negative exponent e by finding the modular multiplicative inverse d of $b \pmod{m}$ using the extended Euclidean algorithm.

$$c = b^{e} \pmod{m} = d^{-e} \pmod{m}$$

where $e < 0$ and $b \cdot d \equiv 1 \pmod{m}$