A MOST CURIOUS ALGORITHM

BRUNO ZIKI KONGAWI

Abstract. In which we demonstrate an algorithm for computing primes in Kotlin.

We begin with the set $\{1,\ 2\}$ and the constant $S = 4$ from the set of natural numbers.

Step 1: We generate all binary partitions of the set.
Step 2: For every binary partition, we generate (whole) positive exponent combinations such that S is the maximum exponent of any given element of any of the two partition subsets.
Step 3: For every binary partition, every element of every partition subset is raised to the power of every (whole) positive exponent less or equal to S (stemming from the set of exponent combinations of Step 2).
Step 4: For every binary partition raised to a set of exponents, we multiply all elements of the each of the subsets (respectively).
Step 5: We collect the sums and the absolute values of the differences of every two products of Step 4.
Step 6: From the results of Step 5, we retain only those that are between the biggest element of the starting set (from Step 1) and its square.
Step 7: We find the smallest from the retained elements (from Step 6) that is not in the starting set (from Step 1).
Step 8: We create a new set by adjoining the element from Step 7 to the starting set from Step 1.
Step 9: We use new set from Step 8 (as a starting set) to start over from Step 1 (with S unchanged).

We say that every collected element from Step 6 is prime, as can be asserted from the Kotlin program below.

```kotlin
/* AMostCuriousAlgorithm.kt */

import kotlin.math.absoluteValue


fun main(args: Array<String>) {
    val results = mutableListOf<Long>()
    val current = mutableListOf(1L, 2L)
    results.add(current.max()!!)
    val resultsCount = HashMap<Long, Int>()
    repeat((0 until 7).count()) {
        val foundPrimes = computePrimes(current)
        foundPrimes.forEach { prime ->
            if (resultsCount.containsKey(prime)) {
                resultsCount[prime] = resultsCount[prime]!! + 1
            } else {
```

```
                resultsCount[prime] = 1
            }
        }
        val distinctPrimes = foundPrimes.distinct()
        println("Found primes: $distinctPrimes")
        results.addAll(distinctPrimes)
        val differenceMin = distinctPrimes.difference(current).min()!!
        current.add(differenceMin)
    }
    println("Hello primes: ${results.distinct().sorted()}")
    println("Occurence counts: ${resultsCount.toSortedMap()}")
    println("found composites: ${results.filter { !it.isPrime() }}")
}

fun Long.isPrime() = this > 1L && (2L..(this / 2)).all { this % it != 0L }

fun computePrimes(seedPrimes: List<Long>, maxExponent: Int = 4): List<Long> =
    if (seedPrimes.isEmpty()) emptyList()
    else {
        seedPrimes.binaryPartitions()
            .applyExponents(seedPrimes.size.exponentCombinations(maxExponent))
            .toSumsAndDifferences()
            .filter {
                val max = seedPrimes.max() ?: 1
                it in LongRange(max + 1, max.pow(2) - 1)
            }
            .sorted()
    }

fun <T> List<T>.difference(other: List<T>): List<T> =
    (this subtract other).toList()

fun List<List<List<Long>>>.toSumsAndDifferences(): List<Long> =
    if (isEmpty()) emptyList()
    else {
        val sumAndDifferences = mutableListOf<Long>()
        forEach {
            val firstSuccessiveProduct = it.first().reduce { acc, l -> acc * l }
            val secondSuccessiveProduct = it.last().reduce { acc, l -> acc * l }
            val sum = firstSuccessiveProduct + secondSuccessiveProduct
            sumAndDifferences.add(sum)
            val diff = (firstSuccessiveProduct - secondSuccessiveProduct).absoluteValue
            sumAndDifferences.add(diff)
        }
        sumAndDifferences
    }

fun List<List<List<Long>>>.applyExponents(exponents: List<List<Int>>): List<List<List<Long>>> =
    if (isEmpty()) emptyList()
    else {
```

```
        val primesRaisedToAPower: MutableList<List<List<Long>>> = mutableListOf()
        forEach { binaryPartitions ->
            val firstList = binaryPartitions.first()
            val secondList = binaryPartitions.last()
            exponents.forEach {
                val exponentListPair = it.split(firstList.size)
                val firstExponentList = exponentListPair.first
                val secondExponentList = exponentListPair.second
                primesRaisedToAPower.add(
                    listOf(
                        firstList.mapToPower(firstExponentList),
                        secondList.mapToPower(secondExponentList)
                    )
                )
            }
        }
        primesRaisedToAPower
    }

fun List<Long>.mapToPower(exponents: List<Int>): List<Long> =
    mapIndexed { index, item -> item.pow(exponents[index]) }

fun Long.pow(exp: Int): Long =
    if (exp <= 1) this
    else {
        var product = this
        repeat((1 until exp).count()) {
            product *= this
        }
        product
    }

fun <T> List<T>.split(n: Int): Pair<List<T>, List<T>> = Pair(take(n), drop(n))

fun Int.exponentCombinations(maxExponent: Int): List<List<Int>> =
    if (this < 1) emptyList()
    else {
        val exponents = (1..maxExponent).toList()
        val seedExponents: MutableList<List<Int>> = mutableListOf()
        repeat((1..this).count()) {
            seedExponents.add(exponents)
        }
        combinations(this, seedExponents.flatten().toList()).distinct()
    }

fun <T> List<T>.binaryPartitions(): List<List<List<T>>> =
    if (size < 2) emptyList()
    else {
        val binaryPartitions: MutableList<List<List<T>>> = mutableListOf()
        (1..this.size / 2).forEach { splitIndex ->
            binaryPartitions.addAll(group(listOf(splitIndex, this.size - splitIndex), this))
        }
        binaryPartitions
```

```
    }

fun <T> group(sizes: List<Int>, list: List<T>): List<List<List<T>>> =
    if (sizes.isEmpty()) listOf(emptyList())
    else combinations(sizes.first(), list).flatMap { combination ->
        val filteredList = list.filterNot { combination.contains(it) }
        group(sizes.tail(), filteredList).map { it + listOf(
    combination) }
    }

fun <T> combinations(n: Int, list: List<T>): List<List<T>> =
    if (n == 0) listOf(emptyList())
    else list.flatMapTails { subList ->
        combinations(n - 1, subList.tail()).map { (it + subList.first
    ()) }
    }

fun <T> List<T>.flatMapTails(f: (List<T>) -> (List<List<T>>)): List<
    List<T>> =
    if (isEmpty()) emptyList()
    else f(this) + this.tail().flatMapTails(f)

fun <T> List<T>.tail(): List<T> = drop(1)
```

Presuming Kotlin is installed on a modern machine, compiling and running this program outputs the following:

```
Found primes: [3]
Found primes: [5, 7]
Found primes: [7, 11, 13, 17, 19, 23]
Found primes: [11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
Found primes: [13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
     71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113]
Found primes: [17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 73,
     79, 83, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149,
    151, 157, 163, 167]
Found primes: [19, 29, 31, 37, 41, 43, 53, 61, 67, 71, 73, 83, 89, 97,
     103, 107, 109, 113, 131, 137, 139, 149, 151, 157, 167, 173, 179,
    181, 191, 193, 197, 199, 227, 229, 233, 239, 241, 251, 257, 263,
    269, 271, 277, 281, 283]
Hello primes: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
     53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
    127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191,
    193, 197, 199, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271,
    277, 281, 283]
Occurence counts: {3=16, 5=28, 7=88, 11=104, 13=156, 17=184, 19=216,
    23=184, 29=136, 31=108, 37=136, 41=144, 43=112, 47=116, 53=68,
    59=60, 61=84, 67=92, 71=100, 73=100, 79=76, 83=88, 89=88, 97=116,
    101=56, 103=124, 107=96, 109=88, 113=104, 127=24, 131=60, 137=52,
    139=44, 149=36, 151=40, 157=36, 163=24, 167=40, 173=12, 179=36,
    181=48, 191=12, 193=36, 197=24, 199=24, 227=12, 229=48, 233=12,
    239=12, 241=36, 251=36, 257=24, 263=24, 269=12, 271=44, 277=12,
    281=24, 283=12}
found composites: []
```

Assuming the starting set has at least two elements, we conjecture that the algorithm always yields primes for any $S > 0$, provided that the exponents are natural numbers (greater than zero) and the starting set contains all primes below the greatest of the set (which must also be prime) regardless of the presence of the unit 1; hence, the set $\{2,\ \ 3\}$ can also be used as the starting set. Further, we hypothesize—beyond the limitations of this program (in terms of space or time complexity)—that the greater the upper bound S, the more exhaustive the algorithm.

An ulterior, more elaborate manuscript (with few variants of this program) explicating the reasoning and correctness of this algorithm shall follow. A first draft paper has been submitted to the Annals of Mathematics for review.

Email address: `bruno.ziki.kongawi@gmail.com`