In this report, we will discuss the deterministic polynomial-time prime-testing algorithm discovered by Agrawal, Kayal and Saxena, and present an implementation of this algorithm in the Haskell programming language.

It is often useful to be able to determine whether an integer is prime, especially for cryptographic applications. For example, RSA, a common public-key cryptosystem needs two large random prime numbers. In order to generate them, we can get keep getting random numbers until we get one that is prime. Since only a fraction of these random numbers will be prime, it is important that we are able to test for their primality in an efficient way. There are two types of such algorithms: Probabilistic and non-probabilistic.

The probabilistic algorithms, as their name suggest, work in a probabilistic fashion, which means that the output has a certain probability of being wrong. The non-probabilistic, or deterministic, algorithms, on the other hand are guaranteed to be right. Until recently, all the known deterministic prime-testing algorithms ran in exponential time, meaning that as the input increases linearly, the run-time of the algorithm increases exponentially. This means that it becomes infeasible to determine whether an large integer is prime using these algorithms. The probabilistic algorithms, on the other hand, are a lot faster, and are able to run in polynomial time. However, we can never be sure if the result is correct, although we can get arbitrarily confident, by repeating the algorithm as many times as needed. This might not be enough for some applications.

In 2002, Agrawal, Kayal and Saxena, students at the Indian Institute of Technology Kanpur, discovered a deterministic algorithm for primality-testing that runs in $\tilde{O}(\log^{12} n)$, which is polynomial. The basic principle of this algorithm is that $(X + a)^n \equiv X^n + a \pmod{n}$ where $a \in \mathcal{Z}$, $n \in \mathcal{N}$ and $(a, n) = 1$ if and only if $n$ is prime. So, testing if $n$ can be done relatively quickly if we are able to reduce the number of coefficients terms. This is done by checking if $(X + a)^n \equiv X^n + a \pmod{X^r - 1, n}$ is true for some well chosen $r$. The complete proof of correctness of this algorithm is given in [1].

The algorithm is the following:

```
$isPrime(n) :=$
if $isPerfectPower(n)$
  then return $False$
fi
$r \leftarrow min(rorderModnr > 4(logn)(logn))$
if $1 < (a, n) < n$ for some $a \leq r$
  then return $False$
```

```
fi
if $n \le r$
  then return $True$
fi
for $a = 1$ to $\lfloor 2\sqrt{\phi(r)}\log n \rfloor$ do
    if $(X+a)^n \not\equiv X^n + a \pmod{X^r - 1, n}$
      then return $False$
    fi
od
return $True$
```

Our implementation is able to determine that 101 is a prime number, as we can see:

```
-->>> isPrime 103

True
```

Similarly, it is able to determine that 323 is not a prime number:

```
-->>> isPrime 323

False
```

We can also use it to print out a list of all the primes between 100 and 500, by testing every integers in that range, outputting only the ones that are prime:

```
-->>> fst $ unzip $ filter (snd) [(a, isPrime a) | a <- [100..500]]

[101,103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,
197,199,211,223,227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,
311,313,317,331,337,347,349,353,359,367,373,379,383,389,397,401,409,419,421,
431,433,439,443,449,457,461,463,467,479,487,491,499]
```

This operation takes almost 7 minutes, on an AMD Athlon 2200+ computer, which shows that despite the fact that it runs in polynomial time, it has large "hidden constants" that do not show up in the asymptotic upper bound of its run time. Therefore, it is a lot faster to use other algorithms, unless we are dealing with extremely large integers. However, we should note that our implementation is not very optimized, and could by a lot faster, by using more efficient algorithms and data structures for the polynomial arithmetic. For instance, we found by profiling the code that polynomial multiplication is what takes the majority of the runtime. As we will see from the source code, the algorithm we use for multiplication is $O(m * n)$ (where $m$ and $n$ are the number of terms in each of the two polynomials we are multiplying). It is however possible to do multiplication in (almost) $O(n)$, by using Discrete Fourier Transforms.

We will now thoroughly comment the code.
First, we import the modules that we need.

```
import List
```

The First step of the algorithm is to determine whether $n$ is a perfect power. To do that, we just need to verify that $n^{1/k}$ is an integer for some $k \in \mathcal{Z}, 2 < k < \log_2 n$. Please note that to find out if $n^{1/k}$ is an integer, we compare two double precision floating point numbers. This is because of the lack of arbitrary precision real number support in the Haskell programming language. Implementing such a data structure would have been well outside the scope of this project. Because of this, our implementation will now work on numbers greater than some value, which should not really be a problem, because of the time required to test even 3 digit numbers.

```
isPerfectPower :: Integer -> Bool
isPerfectPower n = not $ null $
   filter filt $ [n'**(1/k) | k <- [2..logBase 2 n']]
    where n' = fromInteger(n)::Double
  filt x = (fromInteger(truncate x)::Double) == x
```

The next step in the algorithm is to find the smallest $r$ such that $order_r(n) > 4\log^2 n$

```
orderMod :: Integer -> Integer -> Integer
orderMod m n = if null l then 0 else snd $ head l
    where l = filter (\x -> (fst x) == 1) $ zip [mod (n^k) m | k <- [1..m]]
             [1..]

getR :: Integer -> Integer
getR n = snd $ head $ filter (\x -> (fst x) > (truncate $ 4 * (log n')^2))
              [(orderMod k n, k) | k <- [1..]]
            where n' = fromInteger(n)::Double
```

Then, we need to verify that $1 < (a, n) < n$ for all $a \in \mathcal{Z}, a \leq r$.

```
checkGcds :: Integer -> Bool
checkGcds n = not $ null $ filter (\x -> (x /= 1) && (x < n))
            [gcd n k | k <- [1..getR n]]
```

The last part of the algorithm deals with polynomial arithmetic, so we have to implement several operations on polynomials.
We represent polynomial as lists of tuples. Each tuple represents a term of the polymomial. The first element of a tuple represents the order of the term. The second one represents its coefficient. So, for example, $X^4 + 2X + 3$ is represented as `[(4,1), (1,2), (0,3)]`.

```
data Poly = Poly [(Integer, Integer)] deriving Show

mkPoly :: [(Integer, Integer)] -> Poly
mkPoly l = Poly (mkPoly' l)
    where
    mkPoly' [] = []
    mkPoly' (x:xs)
 | (snd x) == 0 = mkPoly' xs
 | otherwise = x : mkPoly' xs
```

Polynomial addition and subtraction are done by simply adding or subtracting the coefficient of the corresponding terms.

```
addPoly :: Poly -> Poly -> Poly
addPoly (Poly a) (Poly b) = mkPoly (addPoly' a b)
    where
    addPoly' [] b = b
    addPoly' a [] = a
    addPoly' xl@((xExp, xCo):xs) yl@((yExp, yCo):ys) =
case compare xExp yExp of
     EQ -> (xExp, xCo + yCo) : addPoly' xs ys
     LT -> (yExp, yCo) : addPoly' xl ys
     GT -> (xExp, xCo) : addPoly' xs yl

subPoly :: Poly -> Poly -> Poly
subPoly (Poly a) (Poly b) = mkPoly (subPoly' a b)
    where
    subPoly' [] b = [(e, -c) | (e, c) <- b]
    subPoly' a [] = a
    subPoly' xl@((xExp, xCo):xs) yl@((yExp, yCo):ys) =
case compare xExp yExp of
     EQ -> (xExp, xCo - yCo) : subPoly' xs ys
     LT -> (yExp, -yCo) : subPoly' xl ys
     GT -> (xExp, xCo) : subPoly' xs yl
```

Multiplication is done using the “naive algorithm” used when doing multiplication by hand. We multiply the first polynomial by each term of the second one, and add the intermediary results together. As we have mentioned before, this is $O(m * n)$, and could be improved, by using Discrete Fourier Transforms. Since this is the function that uses the most run time in the whole program, it should be the first one we would have to improve, would we want to increase the performance of the whole program.

```
mulPoly :: Poly -> Poly -> Poly
mulPoly (Poly a) (Poly b)
    | null a = Poly []
    | null b = Poly []
    | otherwise = foldr1 addPoly [mulPoly' e c b | (e, c) <- a]
    where mulPoly' ex cx ls = mkPoly [(ex + e, cx * c) | (e, c) <- ls]
```

Just like for multiplication, we use the same algorithm we would use when dividing polynomials by hand. We should note that this never returns, if the polynomial we are dividing by has degree zero. This is not a problem, because we avoid doing it.

```
divPoly :: Poly -> Poly -> (Poly, Poly)
divPoly a b = divPoly' a b $ mkPoly []
    where
    grExp (Poly ((a,_):_)) = a
    grCo (Poly ((_,a):_)) = a
    divPoly' r@(Poly r') b q
        | (not $ null r') && (grExp r - grExp b >= 0) = divPoly' newR b newQ
        | otherwise = (q, r)
where
t = mkPoly [(grExp r - grExp b, div (grCo r) (grCo b))]
newR = subPoly  r (mulPoly t b)
newQ = addPoly q t
```

To do polynomial modulo, we just take the remainder from the division.

```
modPoly :: Poly -> Poly -> Poly
modPoly a m = snd $ divPoly a m
```

In the case that we need to take a polynomial modulo a constant, all we need to do is take the modulo of each of the coefficients of the polynomial.

```
modIPoly :: Poly -> Integer -> Poly
modIPoly (Poly a) n = mkPoly $ map (\x -> (fst x, mod (snd x) n)) a

powerModIPoly :: Poly -> Integer -> Poly -> Integer -> Poly
--powerModIPoly a p mp m = foldr1 (\x y -> modIPoly (modPoly (mulPoly x y) mp) m) $ genericTake p $
powerModIPoly a p m mn
    | p == 0 = Poly []
    | p == 1 = a
    | mod p 2 == 0 = let xn2m = powerModIPoly a (div p 2) m mn in modIPoly (modPoly (mulPoly xn2m x
    | otherwise = let xn2m = powerModIPoly a (div p 2) m mn
      in modIPoly (modPoly (mulPoly a $ mulPoly xn2m xn2m) m) mn

isZeroPoly :: Poly -> Bool
isZeroPoly (Poly a) = null a

eulerPhi :: Integer -> Integer
eulerPhi n =  sum $ snd $ unzip $ zip (filter (== 1) [gcd k n | k <- [1..n]]) $ repeat 1

isPrime :: Integer -> Bool
isPrime n = if isPerfectPower n then False
    else let r = getR n in
     if checkGcds n then False
     else case n <= r of
     True -> True
     False -> let n' = fromInteger(n)::Double
  phiR = fromInteger(eulerPhi(r))::Double
  maxA = truncate(2 * sqrt(phiR) * log (n'))
  xr1 = mkPoly [(r,1), (0,-1)]
  xna a = mkPoly [(n,1),(0,a)]
  xan a = powerModIPoly (mkPoly [(1,1), (0,a)]) n xr1 n
  p1 a = xan a
  p2 a = modIPoly (modPoly (xna a) xr1) n
  in null $ filter (not . isZeroPoly) [modIPoly (subPoly (p1 a) (p2 a)) n | a <- [1..maxA]]

main = putStr $ (show $ fst $ unzip $ filter (snd) [(a, isPrime a) | a <- [100..500]]) ++ "\n"
--main = putStr $ show $ isPrime 211
```

**Bibliography**

[1] Manindra Agrawal, Neeraj Kayal, Nitin Saxena, PRIMES is in P.