PRIMALITY IS IN P

1 Introduction

Manindra Agrawal, Neeraj Kayal, and Nitin Saxena have a paper called "Primes in P" which shows an algorithm, and a proof of correctness, that primality is in P. This is a sketch of that algorithm. We care about the algorithm running in time polynomial in $\log n$, hence we will ignore factors of the form $\log \log n$.

2 History

Given a number n we want to test it for primality. We want the test to use time bounded by a polynomial in $\log n$. Note that the length of the input is $\log n$.

There are very good randomized algorithms for this problem that either (a) are very fast but might make a mistake, or (b) are always correct but if you are unlucky may run along time. In addition there is an algorithm that, assuming the Extended Riemann Hypothesis (well believed by most mathematicians), runs in polynomial time and is always correct. The randomized algorithm is excellent in practice, so the pragmatic need for an algorithm in P is not clear. However, the question of obtaining a deterministic polynomial time algorithm for Primality is still interesting:

1. Intellectually, its worth knowing.
2. There is a train of thought that anything for which there is a randomized poly time algorithm for (like primality), there is also a deterministic poly time algorithm. Nisan and Wigderson have proven that if you assume some reasonable hypothesis then every problem in Randomized Polynomial Time is actually in P. (There has been much follow up work that sharpens what we need to assume.) Hence whenever a problem in randomized polynomial time ends up in P this is further evidence for this train of thought. (We note that the primality algorithm does not use the techniques of NW to convert the randomized algorithm to a deterministic one.)
3. Even though the algorithm is currently not practical the ideas in it may be exploited to obtain a practical algorithm. This is optimistic speculation.
4. Even though the algorithm is for primality the ideas in it may be exploited to obtain a practical algortihm for factoring. This is either optimistic or pessimistic speculation depending on if you want to make or break cryptosystems.

We give an example of a prior randomized algorithm. Our explanation will be incomplete but it will give an idea of the type of algorithm we are concerned with.

Lemma 2.1 *(Fermats little theorem) If n is prime then, for all a, $a^n \equiv a \pmod{n}$.*

Randomized Algorithm for Primality

1. Input(n).
2. Pick s random values of $a \in \{2, 3, \ldots, n-1\}$. Call them $a_1, \ldots, a_s$.
3. For all i, $1 \le i \le n-1$, compute $a_i^n \pmod{n}$ and see if it equals a.
4. If there exists an i such that $a_i^n \not\equiv a_i \pmod{n}$ then output COMP. Otherwise output PRIME.

PROS and CONS of the algorthim.

CON: There are nonprimes n such that, for all a, $a^n \equiv a \pmod{n}$. Hence the algorithm could be wrong. The algorithm can be modified to take care of this.

CON: Possible (even with modification) that the s numbers you pick all have $a_i^n \equiv a_i \pmod{n}$.

PRO: The probability of being that unlucky is at most $\frac{1}{2^s}$.

PRO: If n is prime the algorithm will output PRIME.

CON: Computing $a_i^n \pmod{n}$ seems to take n operations which is alot compared to the length $\log n$.

PRO: By doing repeated squaring one can compute a_i^n in $O(\log n)$ steps. You do a_i^2, $(a_i^2)^2$ (reduce mod n each time).

Our algorithm for primality in P will also use a lemma from number theory similar the one above.

3 A Nice Characterization of Primality

Lemma 3.1 *Assume n is prime. If $0 < i < n$ then n divides $\binom{n}{i}$.*

Proof: $\binom{n}{i}$ is $\frac{n(n-1)\cdots(n-i+1)}{i!}$. Factor the numberator and denominator into primes uniquely. The numerator has a factor of n which is prime, but the denominator does not. Hence when you're done cancelling there is still a factor or n. Hence n divides $\binom{n}{i}$. ∎

Lemma 3.2 *If n is composite and $\gcd(a, n) = 1$ then $(x-a)^n \not\equiv x^n - a \pmod{n}$.*

Proof:

Let q be a prime factor of n. Assume q^k divides n but q^{k+1} does not divide n.

Claim: q^k does not divide $\binom{n}{q} = \frac{n(n-1)\cdots(n-q+1)}{q!}$. This is because q^k is the highest power of q that divides the numerator, and q^1 is the highest power of q that divides the denominator; hence q^{k-1} divides $\binom{n}{q}$ but q^k does not.

Note that

$$(x-a)^n = x^n + \cdots + \binom{n}{q} a^{n-q} x^q + \cdots a^n.$$

We claim that the coefficient of x^q is $\not\equiv 0 \pmod{n}$. We have that q^{k-1} divides $\binom{n}{q}$ but q^k does not. Also q does not divide a (since $\gcd(a,n)=1$). Hence the highest power of q that divides $\binom{n}{q}a^{n-q}$ is q^{k-1}. Since q^k divides n we have that $\binom{n}{q}a^{n-q} \not\equiv 0 \pmod{n}$. ▮

We can now characterize primes in a very strong way.

Theorem 3.3

1. *If n is an odd prime then for all a, $0 < a < n$, $(x-a)^n \equiv x^n - a \pmod{n}$.*
2. *If n is composite then for all a, $\gcd(a,n)=1$, $(x-a)^n \not\equiv x^n - a \pmod{n}$.*

Proof: If n is prime then, by Lemma 3.1, for all i, $1 \le i \le n-1$, $\binom{n}{i} \equiv 0 \pmod{n}$. Hence

$$(x-a)^n = x^n + \sum_{i=1}^{n-1} \binom{n}{i} (-a)^i x^{n-i} - a^n \equiv x^n - a^n \equiv x^n - a \pmod{n}.$$

If n is composite then the statement follows from Lemma 3.2. ▮

4 A First Cut at an Algorithm and How to Improve it

We now look at at an algorithm inspired by this theorem. It will not be correct, but it will set us on the right track.

1. Input(n) (Assume n is odd.)
2. Compute $(x-2)^n \pmod{n}$. If this is $\equiv x^n - 2 \pmod{n}$ then output PRIME. Else output COMP.

PROS: This algorithm is always correct.

CONS: Computing $(x-2)^n$ is slow. Realize that this is a polynomial so, even if we do repeated squaring the degree of the polynomial will get large and computationally unwiedly. Note that at every iteration we are squaring a polynomial of degree d for d as large as $\Theta(n)$. Note more directly that since the answer could be a degree n polynomial we may have to output n things, which takes time $\Omega(n)$. Even if the final answer is small (e.g. $x^n - 2$), intermediary computations may be expensive.

SO, how can we speed this up and keep the PROS? We will compute $((x-a)^n \pmod{(x^r-1)} \pmod{n})$. This means that we replace x^r with 1, x^{r+1} with x, etc. We would do repeated squaring, but after each operation replace all terms of degree $\ge r$. Hence we will do $\log n$ operations where each one is multplying two degree $r-1$ polynomials over Z_n This is $\log n$ operations, each one of which is $O(r^2)$ multiplications mod n. Each such multiplication takes $\log^2 n$ so this takes time $O(r^2 \log^3 n)$. Using Fast Fourier Transforms this can be improved to time $O(r \log^2 n)$

Henceforth we will denote
$(XXX \pmod{(x^r-1)} \pmod{n})$ by
$(XXX \pmod{(x^r-1), n})$.
Here is another attempt

1. Input(n)
2. Let $r = \log n$.
3. Compute $(x-2)^n \pmod{(x^r-1), n}$. See if this is $\equiv x^n - 2 \pmod{(x^r-1), n}$. If yes then output PRIME, else output COMP.

PROS: This is fast. Takes $O(r \log^2 n)$ steps.
PROS: If n is prime, this will output PRIME.
CONS: If n is composite this might be incorrect. Computing mod $x^r - 1$ loses information.

We need to find an r such that computing mod $x^r - 1$ does not lose information.

Here is another attempt.

1. Input(n)
2. Choose an r very carefully (but still poly-log n).
3. Compute $(x-2)^n \pmod{(x^r-1), n}$. See if this is $\equiv x^n - 2 \pmod{(x^r-1), n}$. If yes then output PRIME, else output COMP.

The key to this algorithm is how to pick a special r? If we didn't do the calculation mod $x^r - 1$ then we have the strong statement that (see Lemma 3.2)
if n is composite then

$$(\forall a)[\gcd(a,n) = 1 \Rightarrow (x-a)^n \not\equiv x^n - a \pmod{n}.$$

If we compute $\pmod{(x^r-1)}$ then we lose some information. Perhaps we can make up for this by trying several values of a. Here is another incomplete algorithm based on this idea.

```
1. Input($n$)
2. Choose an $r$ very carefully (but still poly-log $n$).
3. For $a = 1$ to (some bound based on $r$)
       if $(x-a)^n \not\equiv x^n - a \pmod{(x^r-1), n}$ then output COMP.
4. (If you got this far then for all $a$ tested you got $\equiv$.) Output PRIME
```

5 More Lemmas Needed

The first lemma in this section is proven in the paper. The second one is not, but is referenced.

Lemma 5.1 *Assume n is composite and not a power. If r is a prime, q is the largest prime factor of $r-1$, $q \geq 4\sqrt{r}\log n$, and $n^{\frac{r-1}{q}} \not\equiv 1 \pmod{r}$, and then there exists $a \leq 2\sqrt{r}\log n$ such that $(x-a)^n \not\equiv x^n - a \pmod{(x^r-1), n}$.*

Lemma 5.2 *Assume n is composite and not a power. One of the following holds.*

1. *There exists $r \leq (\log n)^6$ such that $\gcd(r,n) \neq 1$ (which is evidence that n is composite).*
2. *There exists $r \leq (\log n)^6$ satisfying Lemma 5.1 (which means that there exists $a \leq 2\sqrt{r}\log n$ such that $(x-a)^n \not\equiv x^n - a \pmod{(x^r-1), n}$, which is evidence that n is composite).*

6 The Final Algorithm

```
1. Input($n$).
2. For $b = 2$ to $\log n$    Test if $n$ is a $b$th power. If it is then output COMP.
3. For $r = 2$ to $(\log n)^6$
     If $\gcd(r,n) \neq 1$ then output COMP
     If $r$ is prime then
        Factor $r-1$
        Let $q$ be the largest prime factor of $r-1$.
        If $q \geq 4\sqrt{r}\log n$ and $n^{\frac{r-1}{q}} \not\equiv 1 \pmod{r}$ then break out of loop.
4. (We have a good value of $r$.)
   For $a = 1$ to $2\sqrt{r}\log n$
     if $(x-a)^n \not\equiv x^n - a \pmod{(x^r-1), n}$ then output COMP.
5. (If you got this far then for all $a$ tested you get $\equiv$.) Output PRIME
```

If n is prime then, by Theorem 3.3.1, the algorithm will return PRIME.

If n is composite then, by Lemma 5.2, the algorithm will return COMP.

The rth iteration of the loop in step 3 takes $O(r)$ steps (this can probably be improved). Hence the entire loop takes $O(\sum_{r=2}^{\log^6 n} r) = O(\log^{12} n)$.

Every iteration of the loop in step 4 takes $O(r\log^2 n)$ steps (using Fast Fourier Transforms to do the multiplication). This is at most $O(\log^{12} n)$.

Hence the entire algorithm takes $O(\log^{12} n)$ steps.

It is thought that r need not go as high as $\log^6 n$ and hence this should work much better in practice.