An Efficient Implementation of the AKS Polynomial-Time Primality Proving Algorithm

Chris Rotella

May 6, 2005

Contents

1 History of Primality Testing **3**
1.1 Description of the problem . . . 3
1.2 Previous prime testing algorithms . . . 3

2 The AKS Algorithm **5**
2.1 The motivating idea . . . 5
2.2 Presentation of the algorithm . . . 6

3 Running Time Analysis **9**
3.1 Background . . . 9
3.2 The main theorem . . . 9

4 The Implementation **12**
4.1 Description of implementation details . . . 12
4.2 Bignum library . . . 12
4.2.1 Why GMP? . . . 12
4.2.2 Description of GMP features used . . . 12
4.3 Polynomial library . . . 13
4.3.1 Header file . . . 14
4.3.2 Constructors . . . 14
4.3.3 getCoef . . . 15
4.3.4 isEqual . . . 16
4.3.5 setCoef . . . 16
4.3.6 compact . . . 16
4.3.7 clear . . . 17
4.3.8 Destructor . . . 17
4.3.9 mpz_pX_mod_mult . . . 18
4.3.10 mpz_pX_mod_power . . . 19
4.4 Sieve . . . 20
4.4.1 Header file . . . 20
4.4.2 Constructor . . . 21
4.4.3 isPrime . . . 21
4.4.4 Destructor . . . 22
4.4.5 Implementation notes . . . 22

4.5 The completed implementation 22
4.6 Overall implementation notes 24

5 Empirical Results **25**
5.1 Description of testing environment 25
5.2 Density of the polynomials 25
5.3 Timing results 25
5.3.1 Breakdown 27
5.3.2 Lower bound on maximal a? 27
5.3.3 Comparisons 28
5.4 Profiler results 28
5.4.1 GMP polynomials 28
5.4.2 `unsigned int` polynomials 28

6 Conclusions **32**
6.1 Improvements to AKS and future work 32
6.2 Conclusions 32

A Acknowledgments **34**

Chapter 1

History of Primality Testing

1.1 Description of the problem

A *prime number* is a natural number n that is divisible only by 1 and itself. Prime numbers are the building blocks of the natural numbers since any natural can be uniquely expressed as a product of prime numbers. Numbers that are not prime are called *composite*.

It is natural to ask, given a positive integer n, is n prime or composite? This question may be formalized in the definition of PRIMES: the decision problem of determining whether or not a given integer n is prime. Thus, an algorithm that tests integers for primality solves PRIMES.

1.2 Previous prime testing algorithms

Since primes are central to number theory, they have been the subject of much attention throughout history.

The Sieve of Eratosthenes was one of the first explicit algorithms. It produces a table of numbers that indicates whether each number in the table is prime or composite.

Pratt [9] proved that PRIMES is in NP, meaning that given n and some certificate c, n's primality can be verified in polynomial time using c.

Miller [7] showed that PRIMES in P if the Extended Riemann Hypothesis (ERH) is true. In short, PRIMES's membership in P requires that finding a certificate for a number's primality can be done in polynomial time. However, Miller's argument assumed the Extended Riemann Hypothesis, which is one of the major outstanding problems in mathematics. The Hypothesis is concerned the the zeroes of the Riemann zeta function, and has deep theoretical connections to the density of the primes.

Rabin [10] randomized Miller's approach, thereby eliminating the dependency on ERH, to create an efficient primality test now called Rabin-Miller [2]. While it has a chance of returning positively for a composite, the probability of

error can be made arbitrarily small. Rabin-Miller is very efficient and is used in many applications, including GMP [3].

In 2003, Agarwal, Kayal, and Saxena [1] made their "PRIMES is in P" paper available. They provided a deterministic, polynomial-time primality proving algorithm. They received world-wide press coverage for their finding.

Chapter 2

The AKS Algorithm

2.1 The motivating idea

The AKS Algorithm was motivated by the following lemma [2]:

Lemma 1. *Let $n \geq 2$, and let $a < n$ be an integer that is relatively prime to n. Then*

$$n \text{ is a prime number} \Leftrightarrow (X+a)^n = X^n + a \pmod{n}.$$

Proof. All calculations are done in $\mathbb{Z}_n[X]$. ($\Rightarrow$) We have

$$(X+a)^n = X^n + \sum_{0<i<n} \binom{n}{i} a^i X^{n-i} + a^n \tag{2.1}$$

by the Binomial Theorem. For $0 < i < n$, $\binom{n}{i} \equiv 0 \pmod{n}$. All of the binomial coefficients are thus 0 in Eq. 2.1. Hence,

$$(X+a)^n = X^n + a^n. \tag{2.2}$$

By Fermat's Little Theorem, $a^n = a$ in $\mathbb{Z}_n$. Thus, $(X+a)^n = X^n + a$.

($\Leftarrow$) Assume that n is composite. Choose $p < n$ and $s \geq 1$ such that p is a prime factor of n and p^n divides n, but p^{s+1} doesn't. By Eq. 2.1, the term X^{n-p} has the coefficient

$$\binom{n}{p} \cdot a^p = \frac{n(n-1)(n-2)\cdots(n-p+1)}{p!} \cdot a^p. \tag{2.3}$$

n is divisible by p^s, and the other factors are all relatively prime to p. Therefore, the numerator is divisible by p^s but not p^{s+1}. The denominator is trivially divisible by p. Also, because a and n are relatively prime, a^p is not divisible by p. Thus, we have that $\binom{n}{p} \cdot a^p$ is not divisible by p^s, and by extension, not divisible by n. Therefore, $\binom{n}{p} \cdot a^p \not\equiv 0 \pmod{n}$. Therefore, it is not possible for $(X+a)^n$ to be equal to $X^n + a$. □

This congruence leads to the following algorithm:

Algorithm 2.1 Naive AKS

1 **if** (in $\mathbb{Z}_n[X]$) $(X+1)^n = X^n + 1$ **then**
2   **return** "prime"
3 **return** "composite"

Clearly, Algorithm 2.1 is simple, but is intractable for even modest n, because the number of arithmetic operations is, at best, $O(n)$.[1]

Perhaps the congruence can be salvaged if the number of terms in the polynomial can be reduced. If we compute in $\mathbb{Z}_n[X]$, modulo $X^r - 1$, where r is a "useful" prime. The number of terms in the resulting polynomial will be $O(n \bmod r)$. Computing modulo $X^r - 1$ is very simple: All exponents greater than r are replaced by $n \bmod r$.

The following definition is simply one of convenience. Call r is a "useful" prime if it has the following properties:

1. $\text{GCD}(n, r) = 1$
2. $r - 1$ has a prime factor q such that
   - $q \leq 4\sqrt{r} \log n$
   - $n^{(r-1)/q} \neq 1 \bmod r$

Since the restrictions of equality was relaxed, testing with one value of a is no longer sufficient. However, there is still a polynomial bound number of values of a that must be checked.

2.2 Presentation of the algorithm

Algorithm 2.2 The AKS Algorithm

1 **if** ($n = a^b$ for some $a, b \geq 2$) **then return** "composite"
2 $r = 2$
3 **while** ($r < n$) **do**
4   **if** (r divides n) **then return** "composite"
5   **if** (r is a prime number) **then**
6   **if** ($n^i \bmod r \neq 1$ for all $i, 1 \leq i \leq 4\lceil \log n \rceil^2$) **then**
7   **break**
8     $r = r + 1$
9 **if** ($r = n$) **then return** "prime"
10 **for** a **from** 1 **to** $2\lceil\sqrt{r}\rceil \cdot \lceil \log n \rceil$ **do**
11   **if** (in $\mathbb{Z}_n[X]$) $(X+a)^n \bmod (X^r - 1) \neq X^{n \bmod r} + a$ **then**
12   **return** "composite"
13 **return** "prime"

The algorithm is broken into two main sections, the witness search in lines 2-9, and the polynomial check in lines 10-12.

Not all primes need to go the polynomial section to be declared prime. Based on empirical evididence, at line 9, $r = n$ for all prime n, $n < 347$.

[1] See Section 3.1

The time spent checking the polynomials is controlled by r, since the size of the polynomials and the number of multiplications needed to compute them, is bounded by it. Figure 2.1 shows the values of r for $347 \leq n \leq 10000$.

A polynomial upper bound on r is needed to ensure that AKS runs in polynomial time. Since $\rho(n) > \mathrm{ord}_{\rho(n)}(n) > (logn)^2$. Thus, it is not possible for AKS to use less than $O\tilde{}((\log n)^4)$ bit operations [4]. The empirical evidence in Figure 2.1 suggests that the lower bound is higher. While a proof of this conjecture is beyond the scope of this work, it does point to a problem of AKS. As will be discussed below, the time spent multiplying the polynomial dominates the witness search. Thus, the overall running time is controlled by r.

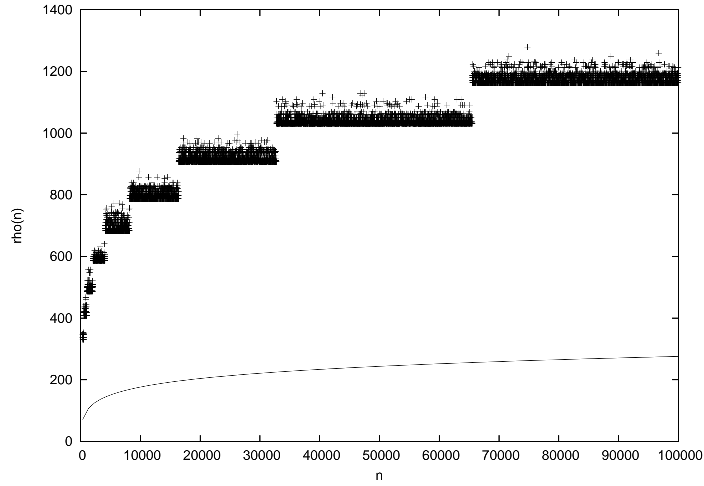

Figure 2.1: The values of $\rho(n)$ and its lower bound $(\log n)^2$, for $347 \leq n \leq 10000$.

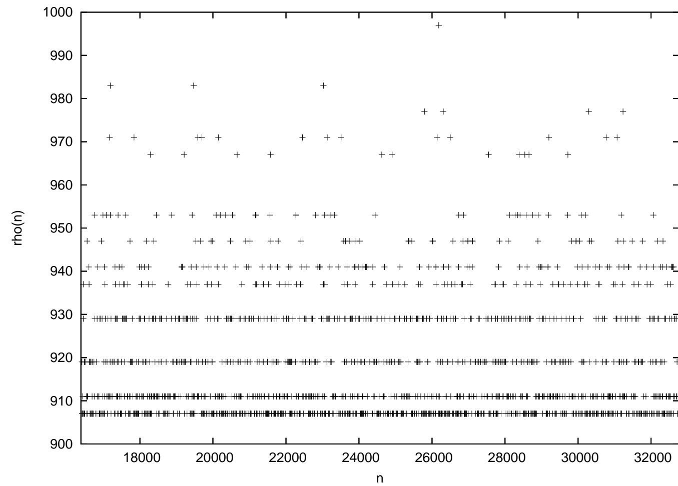

Figure 2.2: The values of $\rho(n)$ for $16384 \leq n \leq 32768$.

Chapter 3

Running Time Analysis

3.1 Background

We begin with the definition of "Big-Oh" notation:

Definition 1. $f(n) \in O(g(n))$ *if there exists* n_0 *and some constant* c *such that for all* $n \geq n_0$, $f(n) \leq c \cdot g(n)$.

If $f(n) \in O(g(n))$ we say that "g is Big-Oh of f." Informally, we can view g as an upper bound for f, for sufficiently large n. Note that constants are ignored in Big-Oh notation; n and $n + 10000$ are both $O(n)$.

In the analysis of algorithms, *polynomial time* means that as the input grows, the algorithm takes a a number of computational steps that is polynomial in the size of the input. Since the input to AKS is a binary number n, the size of it's input is logarithmic in n, its length in bits. More formally, we must show that the AKS algorithm takes $O((\log n)^c)$ computational steps for some constant c. These proofs are based on those in [2].

3.2 The main theorem

Theorem 1. *Algorithm 2.2 runs in polynomial time.*

Proof. All numbers used in the execution are bounded by n^2, and thus have bit length bounded by $2 \log n$. Naively, all arithmetic operations can be completed in $O((\log n)^2)$ bit operations. We will concern ourselves with the number of arithmetic operations needed.

The perfect power test on line 1 requires $O((\log n)^2 \log \log n)$ arithmetic operations, using the standard algorithm.

Lines 3-8 form the search for the smallest witness r. Let $\rho(n)$ be the maximal r for which the loop is executed for a given input n. Assume $\rho(n) = O((\log n)^c)$, for some constant c. One division is required to test whether r divides n. This takes $O(\rho(n))$ operations overall. In line 5, we must test r for primality. To this

end, we build a table of prime numbers up to $2^{\lceil \log r \rceil}$ using a modified version of the Sieve of Eratosthenes. When r reaches $2^i + 1$, for some i, the table is doubled in size. This requires $O(i \cdot 2^i)$ steps. Since

$$\sum_{1 \leq i \leq \lceil \log(\rho(n)) \rceil} i \cdot 2^i \leq \log(\rho(n)) \cdot 2^{\log(\rho(n))+2} + 2 = O(\log(\rho(n)) \cdot \rho(n)),$$

the total number of arithmetic operations for maintaining and updating the table is $O(\log(\rho(n)) \cdot \rho(n))$. The inner loop of the witness test is line 6. Here, $n^i \bmod r$ is calculated for $i = 1, 2, \ldots, 4\lceil \log n \rceil^2$. The number $\lceil \log n \rceil$ can be calculated in $O(\log n)$ steps. The number of multiplications this time modulo r is $O((\log n)^2)$ for one r. Thus, $O((\log n)^2 \cdot \rho(n))$ for all r.

If a suitable r is found, the **break** statement on line 7 is used, and lines 10-12 are now executed. These lines represent the polynomial check portion of the algorithm. Trivially, $\lceil \sqrt{r} \rceil$ may be computed in time $O(r)$. Now, for all $a, 1 \leq a \leq 2\lceil \sqrt{r} \rceil \cdot \lceil \log n \rceil$, we compute the coefficients of $(X + a)^n \bmod (X^r - 1)$ and compare them to those of $X^{n \bmod r} + a$. Calculating $(X + a)^n \bmod (X^r - 1)$ in $\mathbb{Z}_n[X]/(X^r - 1)$ takes $O(\log n)$ ring multiplications. As discussed above, multiplication is this ring is simple, and amounts to a multiplication of polynomials of degree at most $r-1$ in the ring $\mathbb{Z}_n[X]$ and a polynomial addition, with the coefficients modulo n. Naively, this takes $O(r^2)$ multiplications and additions of elements in $\mathbb{Z}_n$. Thus, for all a, the number of arithmetic operations is bounded by

$$O(\sqrt{\rho(n)}(\log n) \cdot \rho(n)^2 \log n) = O(\rho(n)^{5/2}(\log n)^2) \tag{3.1}$$

All that remains is a proof of the following lemma. □

It is clear then that lines 10-12 dominate the running time of the algorithm, regardless of the actual value of $\rho(n)$.

We will use the following two propositions without proof [2].

Proposition 1. *$\prod_{p \leq 2n} p > 2^n$, for all $n \geq 2$, where the product extends over all primes $p \leq 2n$.*

Proposition 2. *If $p_1, \ldots, p_r$ are distinct prime numbers that all divide n, then $p_1 \cdots p_r$ divides n.*

Lemma 2. *For all $n \geq 2$, there exists a prime number $r \leq 20\lceil \log n \rceil^5$ such that r divides n or r does not divide n and $ord_r(n) > 4\lceil \log n \rceil^2$.*

Proof. This assertion is trivially true for small n, so assume $n \geq 4$. Let

$$\Pi = \prod_{1 \leq i \leq 4L^2} (n^i - 1),$$

where $L = \lceil \log n \rceil$.

Then,

$$\Pi < n^{1+2+\cdots+4L^2} = n^{8L^4+2L^2} < 2^{(\log n) \cdot 10L^4} \leq 2^{10L^5}.$$

By Proposition 1, we have

$$\prod_{r \le 20L^5, r \text{ prime}} r > 2^{10L^5} > \Pi.$$

By Proposition 2, this means that there is some prime number $r \le 20L^5$ that does not divide Π, and thus does not divide any one of the factors $n^i - 1$, $1 \le i \le 4L^2$. Now if r divides n, we are done, since n is not prime. Otherwise, $\mathrm{ord}_r(n)$ is larger than $4\lceil \log n \rceil^2$, since $n^i \not\equiv 1 \pmod{r}$ for $1 \le i \le 4L^2$. □

Therefore, $\rho(n) = O((\log n)^5)$, and Algorithm 2.2 takes $O((\log n)^{14.5})$ arithmetic operations. Using more sophisticated algorithms and analysis, the bound can be reduced to $O\tilde{}((\log n)^{6.5})$. Further, if the Sophie Germain Prime conjecture is assumed, the bound can be reduced to $O\tilde{}((\log n)^5)$.

Bernstein postulates that it may be possible to improve AKS by a factor of two million [4].

Chapter 4

The Implementation

4.1 Description of implementation details

Two implementations were completed, both are functionally equivalent (up to machine limitations). One uses GMP bignums for its integers, the other uses `unsigned ints`. Only the code for the GMP version is presented. See the next section for a description of GMP.

All code was written in C++, however most of GMP is written in C. See [8], [11], and [5].

4.2 Bignum library

A thorough implementation of AKS needs to be able to handle very large integers, including those larger than current 32-bit machines can represent with basic datatypes such as `unsigned int`. Our implementation uses the GNU Multiple Precision Arithmetic Library, or GMP.

4.2.1 Why GMP?

GMP was chosen for this project because it is feature-rich, efficient, and free. It is currently used in commercial applications such as Mathematica.

A custom bignum library was not feasible due to time constraints. Also, we couldn't hope to match the performance of GMP, as many of its functions are hand-crafted in assembly.

4.2.2 Description of GMP features used

Since C++ is used for the implementation, the GMP C++ wrapper class, `mpz_class`, was used as the actual bignums. Overloaded operators are used extensively. However, not all GMP functions are written in C++, so the un-

derlying `mpz_t` struct must be extracted from the `mpz_class` object with the function `get_mpz_t`.

GMP functions used:

- `int mpz_perfect_power_p(mpz_t op)`

  Returns non-zero if `op` is a perfect power.

- `size_t mpz_sizeinbase(mpz_t op, int base)`

  Returns the number of digits of `op` in base `base`. This function is used to calculate logarithms, since GMP has no such functions.

- `int mpz_divisible_p(mpz_t n, mpz_t d)`

  Returns non-zero if `n` is divisible by `d`.

- `void mpz_powm(mpz_t rop, mpz_t base, mpz_t exp, mpz_t mod)`

  Sets `rop` to $\texttt{base}^{\texttt{exp}} \bmod \texttt{mod}$.

- `void mpz_sqrt(mpz_t rop, mpz_t op)`

  Sets `rop` to $\lfloor\sqrt{\texttt{op}}\rfloor$.

- `unsigned long mpz_get_ui(mpz t op)`

  Returns the value of `op` as an `unsigned long`.

- `int mpz_tstbit(mpz_t op, unsigned long int bit_index)`

  Returns the value of bit `bit_index` of `op`.

- `void mpz_setbit(mpz_t rop, unsigned long int bit_index)`

  Sets the value of bit `bit_index` in `rop` to 1.

For more information see [3].

4.3 Polynomial library

The custom polynomial library designed for this implementation is described below.

The term “Library” is a misnomer. The functionality needed is highly specific to AKS. Aside from data structure manipulation, the only two functions need are powering and modular multiplication. Since we are computing in the ring $\mathbb{Z}_n[X]/(X^r - 1)$, the multiplication method is not general. This function is described in detail in Section 4.3.9.

4.3.1 Header file

```
1  class mpz_pX {
2   private:
3    mpz_class **coef; /* Array of coefficients */
4    unsigned int degree;
5
6   public:
7    mpz_pX(); /* constructors */
8    mpz_pX(unsigned int initial_length);
9    mpz_pX(const mpz_pX&);
10
11   friend ostream& operator<<(ostream&, const mpz_pX&);
12   mpz_pX& operator=(const mpz_pX&);
13
14   void setCoef(mpz_class new_coef, unsigned int i);
15   inline mpz_class getCoef(unsigned int i) const;
16
17   inline unsigned int getDegree() const {return degree;};
18
19   int isEqual(mpz_pX p);
20
21   void clear();
22   void compact();
23
24   ~mpz_pX();  /* destructor */
25 };
26
27 void mpz_pX_mod_mult(mpz_pX& rop, const mpz_pX x,
28                      const mpz_pX y, mpz_class mod,
29                      unsigned int polymod);
30
31 void mpz_pX_mod_power(mpz_pX &rop, const mpz_pX& x,
32                       mpz_class power, mpz_class mult_mod,
33                       unsigned int polymod);
```

Polynomials are represented as an array of `mpz_class` pointers.

4.3.2 Constructors

```
1  mpz_pX::mpz_pX() {
2    degree = 0;
3    coef = (mpz_class**)calloc(1,sizeof(mpz_class*));
4    coef[0] = new mpz_class(0);
5  }
```

The default constructor initializes an array of length 1 and inserts a single

zero element.

```
1   mpz_pX::mpz_pX(unsigned int initial_length) {
2     degree = initial_length;
3     coef = (mpz_class **)calloc(1,sizeof(mpz_class*)*
4                                 (initial_length+1));
5
6     unsigned int i;
7     for(i=0; i<=degree; i++) {
8       coef[i] = new mpz_class(0);
9     }
10  }
```

This constructor is utilized when we wish to allocated a known amount of space for a future polynomial.

```
1   mpz_pX::mpz_pX(const mpz_pX& o) {
2     degree = o.getDegree();
3
4     coef = (mpz_class**)calloc(1,sizeof(mpz_class*)*(degree+1));
5
6     unsigned int i;
7     for(i=0; i<=degree; i++) {
8       coef[i] = new mpz_class(o.getCoef(i));
9     }
10  }
```

The copy constructor constructs a new polynomial by deep copying `o`.

4.3.3 getCoef

```
1   inline mpz_class mpz_pX::getCoef(unsigned int i) const {
2     static mpz_class zero(0);
3     if(i > degree)
4       return zero;
5     return *(coef[i]);
6   }
```

`getCoef` returns the `i`th coefficient of the polynomial. If `i` is greater than the degree, an `mpz_class` object representing 0 is returned. This object is declared static so that every invocation of `getCoef` does not result in a `malloc`. We don't need to worry about memory aliasing, since the coefficients returned are always copied before being placed in another polynomial. Because this function is part of the main computational loop, it is inlined for speed.

4.3.4 isEqual

```
1   int mpz_pX::isEqual(mpz_pX o) {
2     if(o.getDegree() != degree)
3       return 0;
4     unsigned int i;
5     for(i=0; i<=degree; i++)
6       if(o.getCoef(i) != *coef[i])
7         return 0;
8
9     return 1;
10  }
```

Returns 1 if and only if `o` has the same coefficients as the polynomial this function was called on. It Assumes that `compact` has been called on both polynomials, ensuring that the test on Line 2 does not return false negatives.

4.3.5 setCoef

```
1   void mpz_pX::setCoef(mpz_class new_coef, unsigned int i) {
2     if(i < 0)
3       fprintf(stderr, "coef is less than 0\n");
4
5     if(i > degree) {
6
7       unsigned int j;
8       coef = (mpz_class **)realloc(coef, sizeof(mpz_class*)*(i+1));
9       for(j=degree+1; j<i; j++)
10        coef[j] = new mpz_class(0);
11      coef[i] = new mpz_class(new_coef);
12      degree = i;
13    }
14    else {
15      delete coef[i];
16      coef[i] = new mpz_class(new_coef);
17    }
18  }
```

`setCoef` sets the `i`th coefficient to `new_coef`. If `i` is greater than the current degree, the degree is set to `i` and a new array is allocated. Zeroes are inserted where necessary, and a copy of `new_coef` is put in the proper place.

If `i` is less than or equal to the current degree, the old coefficient is removed and deallocated, and `new_coef` is copied and placed into the array.

4.3.6 compact

```
1   void mpz_pX::compact() {
```

```
2     unsigned int i;
3     static mpz_class zero = 0;
4     for(i=degree; i>0; i--) {
5       if (*(coef[i]) != zero)
6         break;
7       delete coef[i];
8     }
9     if(degree != i) {
10      coef = (mpz_class **)realloc(coef,
11                                   sizeof(mpz_class*)*(degree+1),
12                                   sizeof(mpz_class*)*(i+1));
13      degree = i;
14    }
15  }
```

In the course of manipulating the polynomials, the non-zero element of highest degree may be less than `degree`. This function searches the array for the first non-zero element, starting from `degree`. If the degree of that element is less than `degree`, the array is reduced in size and the degree is set to the actual degree.

4.3.7 clear

```
1   void mpz_pX::clear() {
2     unsigned int i;
3     for(i=0; i<=degree; i++) {
4       delete coef[i];
5     }
6
7     coef = (mpz_class **)realloc(coef, sizeof(mpz_class*));
8     degree = 0;
9     coef[0] = new mpz_class(0);
10  }
```

All coefficients are deallocated, and the array is resized to hold one object, which is initialized to zero. The polynomial is in the same state as it was after it was constructed.

4.3.8 Destructor

```
1   mpz_pX::~mpz_pX() {
2     unsigned int i;
3     for(i=0; i<=degree; i++) {
4       delete coef[i];
5     }
6     free(coef);
7   }
```

All coefficients are deallocated, as is the array.

4.3.9 mpz_pX_mod_mult

```
1   void mpz_pX_mod_mult(mpz_pX& rop, const mpz_pX _x,
2                        const mpz_pX _y, mpz_class mod,
3                        unsigned int polymod) {
4     mpz_pX x = _x;
5     mpz_pX y = _y;
6
7     rop.clear();
8
9     unsigned int xdeg = x.getDegree();
10    unsigned int ydeg = y.getDegree();
11    unsigned int maxdeg = xdeg < ydeg ? ydeg : xdeg;
12
13    unsigned int k;
14    for(k=0; k<polymod; k++) {
15      mpz_class sum = 0;
16      unsigned int i;
17      for(i=0; i<=k; i++) {
18        sum += x.getCoef(i)*(y.getCoef(k-i)+
19                             y.getCoef(k+polymod-i));
20      }
21      for(i=k+1; i<=k+polymod; i++) {
22        sum += x.getCoef(i)*y.getCoef(k+polymod-i);
23      }
24
25      rop.setCoef(sum % mod, k);
26
27      if(k>maxdeg && sum==0)
28        break;
29    }
30
31    rop.compact();
32  }
```

To eliminate the possibility of memory aliasing, the arguments `_x` and `_y` are deep copied to local variables `x` and `y` respectively using the copy constructor.

The loops in Lines 17-22 are the inner loops of the implementation. Due to the heavy dependence on `getCoef`, it was declared `inline` to reduce the overhead of repeated function calls.

As mentioned above, this is not a general modular multiplication function. Special properties of the polynomials and modulus are exploited for efficiency of implementation. For instance, we know that both `x` and `y` will have degree at most $r-1$, and that the intermediate result will be of degree less than $2r-1$

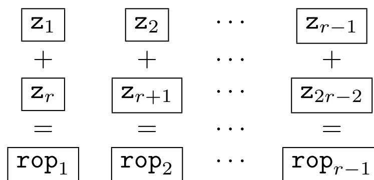

Figure 4.1: A graphical view of the operation performed by `mpz_pX_mod_mult`. `z` is the intermediate product of `x` and `y`.

as well. Since we are computing in $\mathbb{Z}_n[X]/(X^r - 1)$, all coefficients of the form $a_i x^{r+j}$ are added to the coefficient $a_k x^j$. Conceptually, the polynomial is split in half then added together. See Figure 4.1 for a graphical view.

To reduce the number of intermediate bignum objects, only a single integer mod is performed, on Line 25. In the `unsigned int` version of the code, mod operations are carried out at each sum to ensure that there is no overflow.

Because the number of calls to `getCoef` accounts for a large percentage of the real-world time (see Section 5.4), the loops on Lines 17-23 are modified. We pull the term `y.getCoef(k+polymod-i)` out of the second loop, saving $O(\texttt{polymod}^2)$ calls to `x.getCoef(i)` on Line 21 of the unmodified version, shown below.

```
17   for(i=0; i<=k; i++) {
18     sum += x.getCoef(i)*y.getCoef(k-i);
19   }
20   for(i=0; i<=(k+polymod); i++) {
21     sum += x.getCoef(i)*y.getCoef(k+polymod-i);
22   }
```

This change shows greater effect on the `unsigned int` version of the code, since the change in the GMP version introduces more overhead in the form of mallocing intermediate products.

The goal of the test on Line 27 is to save time by exiting the loop early. In the early stages of the exponentiation, the intermediate products will have many fewer terms than `polymod`. Thus, the main loops will be computing zero terms for `k` greater than `maxdeg` (the larger of deg(`x`) and deg(`y`)).

4.3.10 mpz_pX_mod_power

```
1   void mpz_pX_mod_power(mpz_pX &rop, const mpz_pX& x,
2                         mpz_class power, mpz_class mult_mod,
3                         unsigned int poly_mod) {
4
5     rop.clear();
6     rop.setCoef(1,0);
7
8     unsigned int i = mpz_sizeinbase(power.get_mpz_t(),2);
```

```
9     for( ; i >=0; i--) {
10      mpz_pX_mod_mult(rop, rop, rop, mult_mod, poly_mod);
11
12      if(mpz_tstbit(power.get_mpz_t(),i)) {
13        mpz_pX_mod_mult(rop, rop, x, mult_mod, poly_mod);
14      }
15      if(i==0)
16        break;
17    }
18    rop.compact();
19  }
```

The argument `x` is raised to the `power`th power using fast exponentiation. All computations are carried out in $\mathbb{Z}_n[X]/(X^r - 1)$ by using `mpz_pX_mod_mult` as the multiplication function.

The GMP function `mpz_sizeinbase` is used to calculate the logarithm of `power`.

4.4 Sieve

Line 5 of Algorithm 2.2 requires that the current witness candidate r be tested for primality itself. A slightly modified version of the Sieve of Eratosthenes was implemented for this test.

In the course of testing n, a table is built up as each r is tested for primality. The table is initialized to a size of 2. If an r is tested that is greater than the size of the table, the table is doubled and marked according to the Sieve of Eratosthenes.

4.4.1 Header file

```
1   class sieve {
2    private:
3     mpz_t table;
4     unsigned int size;
5
6    public:
7     sieve(); /* constructor */
8     int isPrime(mpz_class r);
9     ~sieve(); /* destructor */
10  };
```

The only data structure needed is an arbitrarily long bit field. If bit i is set to 1, i is composite, otherwise it is prime. Since GMP bignums are essentially bit fields, and they are of arbitrary length, the table is stored as a single `mpz_t` struct. A variable `size`, declared as an `unsigned int`, is used to denote the highest number in the current table.

4.4.2 Constructor

```
1  sieve::sieve() {
2    mpz_init(table);
3    size = 2;
4  }
```

The constructor initializes `table` and sets the size to 2. This indicates that 1 and 2 are prime, since neither are marked. While 1 is not prime in reality, it is marked as such for computation convenience. Doing so has no impact on the output of the sieve, only on the simplicity of the following code.

4.4.3 isPrime

```
1  int sieve::isPrime(mpz_class r) {
2    unsigned int rul = mpz_get_ui(r.get_mpz_t());
3
4    if(size >= rul) { /* just a lookup */
5      return !mpz_tstbit(table,rul);
6    }
7    else {
8      unsigned int oldsize = size;
9      size *= 2;
10
11     unsigned int i;
12     for(i=2; i<=size; i++) {
13       if(!mpz_tstbit(table,i)) {
14         unsigned int j;
15         for(j=i*2; j<=size; j+=i) {
16           mpz_setbit(table,j);
17         }
18       }
19     }
20     return !mpz_tstbit(table,rul);
21   }
22 }
```

The argument `r` has type `mpz_class`. This is for convenience since the main algorithm uses `mpz_class` objects for its integers. The table is indexed using `unsigned int`s, so Line 2 converts `r` appropriately.

Lines 4-6 deal with the case that `rul` is less than the size of the table. Since the table is stored with the composite numbers marked, the logical not of bit `rul` is returned.

The remaining lines build up the table as needed. For every unmarked bit `i`, we mark the `j`th number as composite, starting from `2i`, where `j` is some multiple of `i`.

4.4.4 Destructor

```
1   sieve::~sieve() {
2     mpz_clear(table);
3   }
```

The destructor simply frees the memory occupied by the table.

4.4.5 Implementation notes

Since the table is indexed by `unsigned int`s, the maximum number of elements that can be stored is 2^{32}, or over 4 billion. However, the `r` is logarithmic in n, meaning this implementation will work for any practical value of n.

4.5 The completed implementation

The function `aks` is a complete primality test. It is presented below.

```
1   int aks(mpz_class n) {
2     if(mpz_perfect_power_p(n.get_mpz_t())) {
3       return 0;
4     }
5
6     sieve s;
7
8     mpz_class r = 2;
9     mpz_class logn = mpz_sizeinbase(n.get_mpz_t(),2);
10    mpz_class limit = logn * logn;
11    limit *= 4;
12
13    /* Witness search */
14    while(r<n) {
15      if(mpz_divisible_p(n.get_mpz_t(), r.get_mpz_t())) {
16        return 0;
17      }
18
19      int failed = 0;
20
21      if(s.isPrime(r)) {
22        mpz_class i = 1;
23
24        for( ; i<=limit; i++) {
25          mpz_class res = 0;
26          mpz_powm(res.get_mpz_t(), n.get_mpz_t(),
27                   i.get_mpz_t(), r.get_mpz_t());
28          if(res == 1) {
```

```
29            failed = 1;
30            break;
31          }
32
33        }
34        if(!failed)
35          break;
36      }
37      r++;
38    }
39    if (r == n) {
40      return 1;
41    }
42
43    /* Polynomial check */
44    unsigned int a;
45    mpz_class sqrtr;
46    //actually the floor, add one later to get the ceil
47    mpz_sqrt(sqrtr.get_mpz_t(), r.get_mpz_t());
48    mpz_class polylimit = 2 * (sqrtr+1) * logn;
49
50    unsigned int intr = mpz_get_ui(r.get_mpz_t());
51
52    for(a=1; a<=polylimit; a++) {
53      mpz_class final_size = n % r;
54      mpz_pX compare(mpz_get_ui(final_size.get_mpz_t()));
55      compare.setCoef(1, mpz_get_ui(final_size.get_mpz_t()));
56      compare.setCoef(a, 0);
57      mpz_pX res(intr);
58      mpz_pX base(1);
59      base.setCoef(a,0);
60      base.setCoef(1,1);
61
62      mpz_pX_mod_power(res, base, n, n, intr);
63
64      if(!res.isEqual(compare)) {
65        return 0;
66      }
67    }
68    return 1;
69  }
```

Line 2 performs the perfect power test, returning 0 if `n` is a perfect power and, thus, not prime.

The upper bound for i on Line 6 of Algorithm 2.2 is computed on Lines 9-11. Lines 24-33 actually implement that statement in the pseudo-code. If

the current value of `r` fails the test, we break and move on to the next `r`. If exponentiation yielded no values equal to 1, we break out of the witness search since we have found our AKS witness.

Once a suitable witness has be found, we begin the polynomial check portion of the algorithm. Again, we must compute the limit for the main loop. This is done on Lines 44-48.

Lines 53-50 construct the various objects needed to compute $(X+a)^n$ and compare it to $X^{n \bmod r} + a$. Line 62 computes $(X+a)^n$. If the result does not equal $X^{n \bmod r} + a$, `n` is not prime and 0 is returned. If all values of `a` pass the test, `n` is prime and 1 is returned.

4.6 Overall implementation notes

Both implementations are complete, and no false positives or negatives were found in testing.

The `unsigned int` version will fail if the intermediate products of the polynomial multiplication become greater than 2^{32}. The modulus is taken for every intermediate product, to delay this effect. Even so, overflow may occur for $2n^2 > 2^{32}$, or $n > 2^{15.5}$ with the version above. If the inner loops are rearranged to the naive version presented in Section 4.3.9, the effect may begin at $n^2 > 2^{32}$, or $n > 2^{16}$.

The two implementations have the limitation that r must be a machine-sized number. This is due to both the polynomials and the sieve using `unsigned int`s as indicies. Neither of these problems are easily fixed. While a linked-list implementation of the polynomials would remove the explicit limitation, one would still have to store r coefficients somehow, since the degree of the polynomial reaches r for even modest n (see Section 5.2). Having that many allocated objects is sure to exploit some bug in the operating system, and available memory will be an issue. A linked-list implementation will be slower than the current version also. Similary, the sieve would have to be re-architected to remove the limitation. Again, any change to this effect will have a large impact on the running time. Luckily, since r is logarithmic in n, this cataclysmic n must be very large for its witness to exceed 2^{32}.

Chapter 5

Empirical Results

5.1 Description of testing environment

All code was written in C++ and compiled with `g++` 3.3 (Apple Computer, Inc. build 1666), the GNU Complier Collection's C++ compiler.

The test system is a Power Macintosh G5 with dual 2.0 GHz processors and 1.5 GB of RAM.

The flags used to configure GMP were `--prefix=/usr/local --enable-cxx --enable-mpfr --disable-shared`. The compiler flag used to build GMP was `-O3`.

Code written for this project was compiled with the flag `-O6`. The algorithms used for comparison were also compiled in this manner, except for `mpz_probab_prime_p`, as it is a part of GMP.

5.2 Density of the polynomials

When representing polynomials, the choice of the data structure used should be driven by the number of non-zero elements and the desire to balance memory usage and execution speed.

Because speed is of chief concern for this implementation, arrays are used to store the coefficients, as explained above. Figure 5.1 shows the average density of the polynomials during the exponentiation. Figure 5.2 shows the average maximum density of the polynomials during exponentiation.

5.3 Timing results

To be of practical use to those interested prime-proving algorithms, AKS must perform competitively to current state-of-the-art methods.

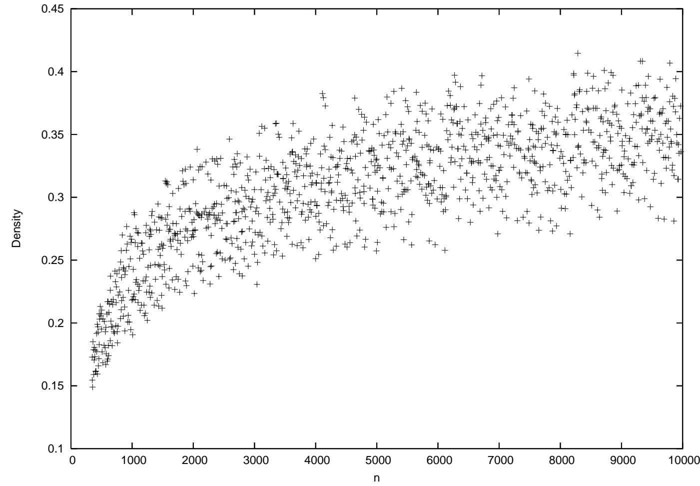

Figure 5.1: The average density of the polynomials in testing n from primality. Density is calculated as average number of non-zero coefficients divided by r.

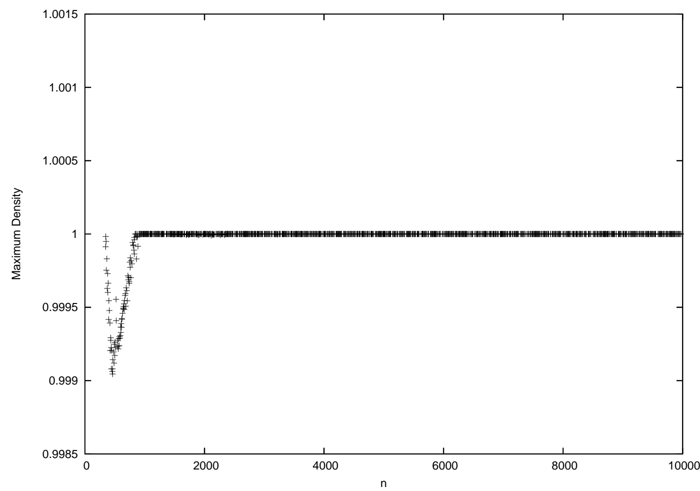

Figure 5.2: The average maxium density of the polynomials in testing n from primality. Density is calculated as average number of non-zero coefficients divided by r.

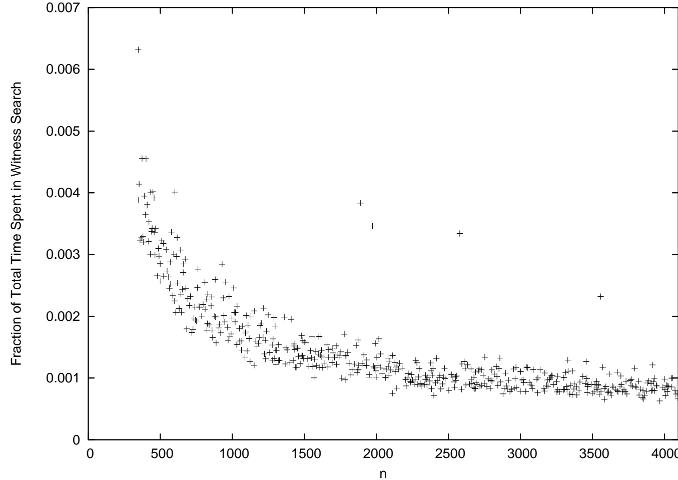

Figure 5.3: The amount of time spent in the witness search divided by the total time for prime n, $347 \leq n \leq 4096$.

5.3.1 Breakdown

As discussed in Section 3.2, asymptotically, the number of steps needed for the witness search is dominated by the number needed for the polynomial exponentiation.

This is evident in the empirical evidence. Figure 5.3 shows the fraction of the time that is spent during the witness test portion of the algorithm for the `unsigned int` version of the code. For nearly all prime $n \geq 347$, the amount of time spent searching for r is less than one-half of one percent. For all prime $n < 347$, the witness search is superset of trial division, stopping at n instead of $\sqrt{n}$.

5.3.2 Lower bound on maximal a?

Given the AKS algorithm, it is natural to ask, what is the smallest a for which a composite n fails the polynomial check? Currently, the answer is unknown: we have yet to detect a composite n that passes the witness test. That number may be as large as 400 million, to ensure to that $n < 20\lceil \log n \rceil^5$.

This interesting result combined with the Figure 5.3 suggests that AKS may be viewed as a relatively fast *compositeness* test for small n.

5.3.3 Comparisons

We compared the AKS implementation against the naive algorithm, the sieve described in Section 4.4, and the GMP function `mpz_probab_prime_p`, which preforms trial divisions and then the Rabin-Miller test. The naive algorithm of trial divisions up to $\sqrt{n}$ is also included.

Litt [6] was able to calculate the first 1,716,050,469 primes in just under one hour and thirty minutes, finding over 300,000 primes per second. He implemented his prime finder with a sieve, using sophisticated paging techniques in conjunction with memory-saving bit arrays.

For consistency, only times for primes are plotted. See Figures 5.4-5.7.

GMP polynomials

The version of the code that used GMP polynomials was significantly slower than the `unsigned int` version. See Figure 5.8.

5.4 Profiler results

To help improve and measure our implementation's efficiency, the code was profiled with Apple Computer's Shark tool.

5.4.1 GMP polynomials

As discussed in Section 5.3.3, the GMP version of the polynomials is very slow. The profiler shows that over thirty percent of the time is spent initializing `mpz_class` objects, and that over twenty percent of the time used is in `malloc` and seventeen percent is in `free`. It was this observation that lead to the creation of the `unsigned int` version.

A speedup could be realized if a custom memory allocated was written. For instance, all integers used in the execution are bounded by n^2, and all polynomial arrays are bounded by r. We conjecture that a custom allocator is necessary to make the GMP version tractable for even $n > 1000$.

5.4.2 `unsigned int` polynomials

The profiler shows that over 98.7 percent of the execution time is spent in `mpz_pX_mod_mult`, and that less than one percent is spent in memory-management functions.

The inlined function `getCoef` is the major bottleneck. Shark reported that 85.9 percent of the samples were taken testing if the requesting coefficient is greater than the current degree.

We attempted to realize a speedup by allocating arrays of size $2r$ and initializing all coefficients to zero. Thus, `degree` is no longer the size of the array, but rather the actual degree of the polynomial. The check is then no longer needed in `getCoef`, we can simply return the requested coefficient. If that coefficient

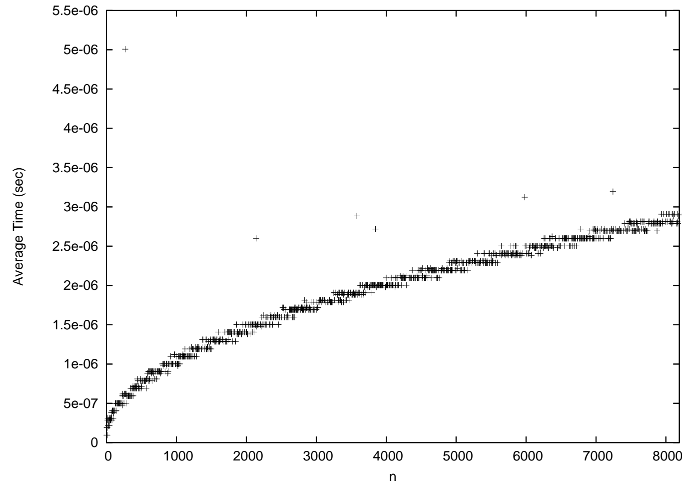

Figure 5.4: The time needed to test n for primality in seconds using the naive algorithm. The outliers are artifacts of CPU scheduling. The reported time is the average over 10 trials.

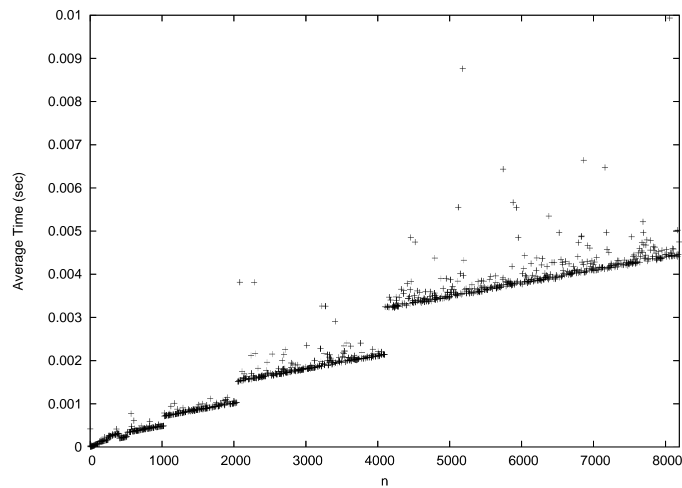

Figure 5.5: The time needed to test n for primality in seconds using the sieve. The outliers are artifacts of CPU scheduling. The reported time is the average over 2 trials.

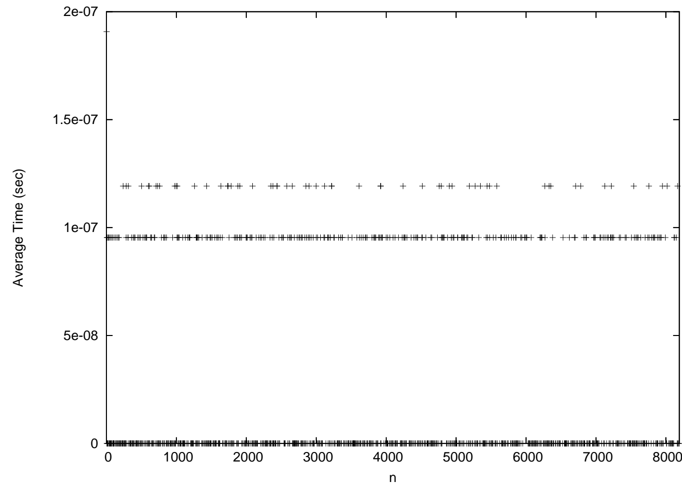

Figure 5.6: The time needed to test n for primality in seconds using `mpz_probab_prime_p`. The reported time is the average over 10 trials.

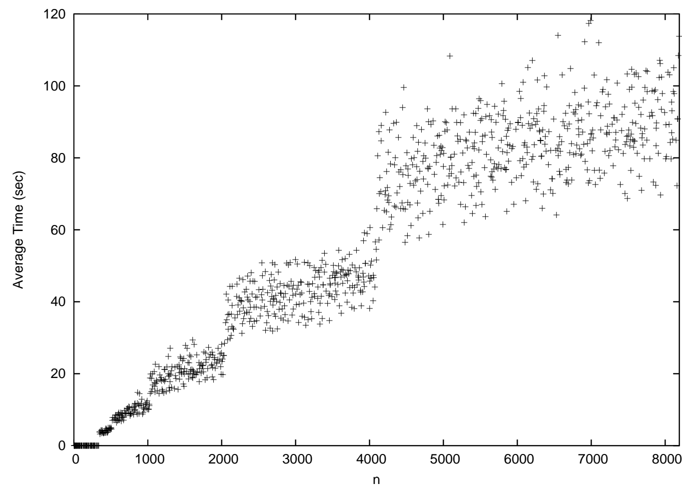

Figure 5.7: The time needed to test n for primality in seconds using AKS with `unsigned int`s. The reported time is the average over 2 trials.

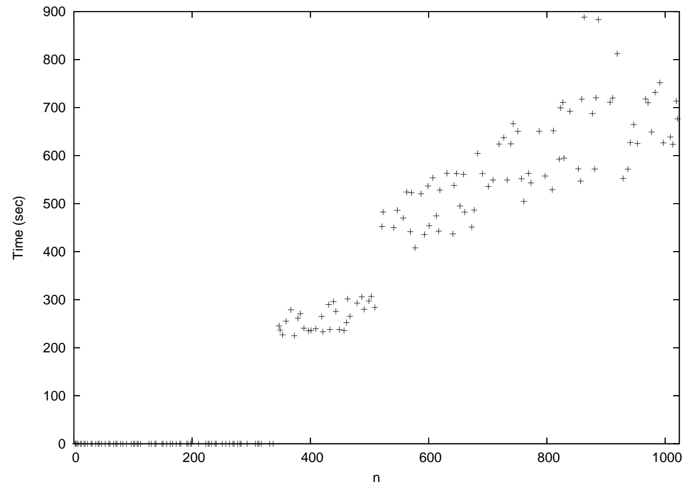

Figure 5.8: The time needed to test n for primality in seconds using AKS with GMP bignums.

has degree larger than `degree`, zero is returned by the nature of the initialization. In practice, this method showed no speedup. Instead, it showed a small decrease in performance.

Chapter 6

Conclusions

6.1 Improvements to AKS and future work

After [1] was released, many researchers began to search for improvements in the running time.

Lenstra and Pomerance were able to modify AKS to achieve a bound of $\tilde{O}((\log n)^6)$ bit operations, which is the theoretical lower bound of AKS [4].

Cheng provided a test based on AKS and the work of others that has running time $O((\log n)^{4+o(1)})$. [4] calls this the AKS-Berrizbeitia-Cheng-Bernstein-Mihailescu-Avanzi test.

Work is underway to modify AKS by removing polynomials completely using linear recurrence sequences [4]. This research is likely to provide a significant real-world speedup.

6.2 Conclusions

In its original version, the algorithm of Agarwal, Kayal, and Saxena is too slow to be of practical use. The hidden constant factors and the actual asymptotic bounds conspire to make the algorithm intractable for even modest inputs.

The terms "polynomial time" or "exponential time" should be used carefully when assessing the actual performance of an algorithm. As we have shown, in the realm of asymptotic analysis and Complexity Theory, AKS is superior to older algorithms such as trial division or Rabin-Miller. Since they are either not polynomial time or not deterministic. However, in using these algorithms for actual primality-proving, it is clear that AKS is not superior.

For the inputs tested here, AKS was several orders of magnitude slower than other algorithms. The difference is so great, that for n large enough that the asymptotics begin to factor, AKS would likely use a large amount of memory. At these n, it may be necessary to re-architect the implementation, which would render it even slower.

At least in its original form, the AKS algorithm, while it is elegant and relatively simple, should be viewed as an algorithmic curiosity rather than an algorithm to be used for actual primality proving.

Appendix A

Acknowledgments

First, I'd like to thank my thesis advisor Klaus Sutner. Without his patience, guidance, and humor, this project would not have been possible.

I'd like to thank Mark Stehlik for his understanding of initial stumbles and for his years of unparralelled academic advising.

Thanks to Manindra Agrawal, Neeraj Kayal and Nitin Saxena for their beautiful algorithm.

Finally, I'd like to thank my friends and family for putting up with me all these years.

References

[1] Agrawal, M., Kayal, N., and Saxena, N. PRIMES is in P. Preprint, available at `http://www.cse.iitk.ac.in/news/primality_v3.pdf`, 2003.

[2] Gietzfelbinger, M. *Primality Testing in Polynomial Time: From Randomized Algorithms to "PRIMES is in P"*. Lecture Notes in Computer Science. Springer, 2004.

[3] Granlund, T. GNU MP: The GNU Multiple Precision Arithmetic Library. Available at `http://www.swox.com/gmp/gmp-man-4.1.4.pdf`, 2004.

[4] Granville, A. It is easy to determine whether a given integer is prime. *Bull. Amer. Math. Soc. 42* (2005), 3–38.

[5] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language*, 2nd ed. Prentice Hall PTR, 1998.

[6] Litt, S. Fun with prime numbers. Available at `http://www.troubleshooters.com/codecorn/primenumbers/primenumbers.htm`, 2004.

[7] Miller, G. Riemann's hypothesis and test for primality. *J. Comput. Syst. Sci. 13* (1976), 300–317.

[8] Pohl, I. *C++ Distilled*. Addison-Wesley, 1997.

[9] Pratt, V. Every prime has a succinct certificate. *SIAM J. Comput. 4* (1975), 214–220.

[10] Rabin, M. Probabilistic algorithm for testing primality. *J. Number Theory 12* (1980), 128–138.

[11] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, 1997.