CRYPTOGRAPHY TOPIC VIII
FACTORING BY TRIAL DIVISION

PAUL L. BAILEY

1. Sieve of Eratosthenes

Eratosthenes (276 BC to 194 BC) was a mathematician in Classical Greece who devised the following method for finding all primes less than t. The algorithm can be used to test for primality of a given number, or to collect a set of primes.

(1) Select a top t.
(2) Let $n = 2$.
(3) If n is not marked, it is prime.
(4) Mark all multiples of n which are less than t.
(5) Increment n.
(6) Go to **(3)**.

We implement this in software as a C++ class, with the following class definition. The remainder of the software for this class may be found at `http://www.saumag.edu/pbailey`. Note that the array `siv` only keeps track of the odd marks.

```
typedef unsigned __int32 UNT;
typedef unsigned __int8  BIT;

class Primer
{ private:
  UNT*  pri;    // array of primes
  UNT   cnt;    // number of primes
  BIT*  siv;    // array of marks
  UNT   sze;    // number of marks
  public:
  Primer(void);
  Primer(UNT top);
  ~Primer(void);
  UNT   operator[](UNT k);      // returns k'th prime
  UNT   Count(void) { return cnt; }
  UNT   Top(void)   { if (sze) return sze*2-1; return 0; }
  void  Clear(void);
  void  Generate(UNT t);        // generate pri and siv
  void  List(void);             // list primes
  private:
  void  mark(UNT i);        // if (siv[i]) // 2^i+3 is composite
};
```

Date: November 13, 2007.

2. Trial Division

Suppose we wish to construct a 32-bit RSA modulus, with relatively large primes. Then, we may run the Sieve of Eratosthenes to locate two large 16-bit primes and take their product.

The RSA public key is (n, e), where n is the RSA modulus and e is the RSA encryption exponent. To break RSA, it suffices to factor the RSA modulus $n = pq$, and then run the Euclidean algorithm to find the RSA decryption exponent d so that $ed \equiv 1 \pmod{(p-1)(q-1)}$.

Any 32-bit integer n can easily be factored using the results of the sieve and the algorithm know as *trial division*, which involves testing each prime to see if it is a factor of n. Now if n is not prime, it has a prime factor which is less than or equal to $\sqrt{n}$. Thus the trial division algorithm is

(1) Set $q = \sqrt{n}$
(2) Set $k = 0$
(3) Set $p = P(k)$, where $P(0) = 2$ and $P(k)$ is the k^{th} prime
(4) If $p > q$, then n is prime
(5) If $n \bmod p > 0$, p is not a factor of n; increment k and go to **(3)**
(6) Otherwise, p is a factor of n; record this fact, and set $n = n/p$.
(7) Go to **(3)**

Assuming that we have a good square root function, trial division can be coded as follows. The primes we find are placed in the array `fac`. The maximum number of primes dividing a 32-bit number is 32; we declare the array to be of size 33 so that the array can be null-terminated.

```
Primer primer(0x10000); // Construct all 16 bit primes

UNT factor(UNT n,UNT* fac) // return 0 on failure, or # of primes
{ UNT p=0,q=sqrt(n);
  int k=0,cnt=0;
  if (n<2) return 0;
  while (n>1 && p<q)
  { p = primer[k++];
    if (p==0) break;      // past top! (shouldn't happen)
    if (n%p) continue;    // p is not a factor
    while (n%p==0)        // pull out all p's
    { fac[cnt++] = p;
      n/=p; }
    q=sqrt(n); }          // reset q
  // what is left of n is prime (unless n==1 or p==0)
  if (n>1)
  { fac[cnt++]=n; }
  fac[cnt]=0; // null-terminated
  return p?cnt:0; }
```

3. Square roots

It remains to construct a square root function for integers. We wish this function to return $\lfloor\sqrt{n}\rfloor$, the greatest integer which is less than or equal to the real positive square root of a positive integer n.

We use Newton's method to accomplish this. So, we consider the function $f(x) = x^2 - n$. The unique positive zero of f is the positive square root of n.

Let x_0 be a real number close to but greater than $\sqrt{n}$; this seeds an iterative algorithm.

Newton's method requires us to find the line tangent to the graph of the function f at the point $(x_i, f(x_i))$. Then, we set x_{i+1} to be the x-coordinate of the x-intercept of this line. In the case of our given f, which is concave up, we will have $x_{i+1} > \sqrt{n}$. So, the sequence (x_i) converges to $\sqrt{n}$ from above.

Since $f'(x) = 2x$, the tangent line at $(x_i, f(x_i))$ is

$$y = 2x_i(x - x_i) + f(x_i) = 2x_i(x - x_i) + x_i^2 - n = 2x_i x - (x_i^2 + n).$$

Setting $y = 0$, solving for x, setting x_{i+1} to the solution, and simplifying, we have

$$x_{i+1} = \frac{1}{2}\Big(x_i + \frac{n}{x_i}\Big).$$

In our case, we take the integer part of x_i at each stage, so that eventually, $x_i \leq \sqrt{n}$. When $x_i < \sqrt{n}$, the concavity of the graph of f forces x_{i+1} to be bigger than x_i; we will use this as a test for completion.

It remains to determine a simple and efficient way to find a seed value x_0 for the algorithm. The simplest solution is to let x_0 be the smallest power of two which is greater than $\sqrt{n}$. The smallest exponent k such that 2^k is greater than n is obtained by finding the highest bit of n and adding one. The smallest exponent of two which greater than $\sqrt{n}$ is then the least integer greater than or equal to $k/2$. This can be computed by examining the bits of n, and rotating the return variable (because left bit rotation is equivalent to multiplying by two).

This produces two functions, which we code thusly.

```
UNT bitmax(UNT u)   // |_ log_2(u) _| + 1
{ UNT k=0;
  while (u) { u>>=1; k++; }
  return k; }

UNT sqrt(UNT a) /* Newton's Method */
{ UNT x,y;
  if (a==0) return 0;
  if (a<=3) return 1;
  x = 1 << (bitmax(a)+2)/2;
  while (1)
  { y=(x+(a/x))/2;
    if (y>=x) break;
    x=y; }
  return x; }
```

4. Testing

Finally, we need some code to test and/or run our new factoring function. This code needs to declare the array of factors, and print the results. We have chosen to allow the user to enter numbers in a loop, and then factor them. To grab the extra bit of an unsigned integer, a text to unsigned integer conversion function is included.

```
UNT atou(STR buf)
{ UNT u=0;
  char chr;
  while (chr = *buf++)
  { if (chr < '0' || chr > '9') break;
    u*=10;
    u+=(chr-'0'); }
  return u; }

void printfactor(UNT n,UNT* fac)
{ int k=0;
  printf("%u = ",n);
  while (fac[k])
  { if (k) printf("*");
    printf("%u",fac[k]);
    k++; }
  printf("\n"); }

void testfactor(void)
{ char buf[80];
  UNT fac[33]; // Null terminated array of factors
  UNT n;
  while (1)
  { printf("Number: ");
    gets(buf);
    n=atou(buf);
    if (n<2) break;
    factor(n,fac);
    printfactor(n,fac); } }
```

Notice that if you run this code and try several 32-bit numbers (less than $2^{32} = 4294967296$), you will eventually find primes which are greater than 16 bits (greater than $2^{16} = 65536$). They were proven to be prime, because they have no prime factors less than or equal to their square root.

To use such primes for RSA, our RSA code would need to be modified to handle moduli of at least 64-bits.

Department of Mathematics and CSci, Southern Arkansas University
E-mail address: plbailey@saumag.edu