RSA Project Report

Justin Day,

Om Bhujel,

Daniel Deda

In the RSA cryptosystem model, the public information (e,n) being:

(5, 8721717648951034180751989933592484457226804450832605710315649207114191)

And the encrypted message being:  
2016279054524078341299256251568889523037078816031686455795312148201129

We were tasked to decrypt the message. For our tools we used mostly Java due to familiarity and problems we faced when trying to implement a large integer representation library in C++. We also used Meataxe, a C package that could do linear algebra within a finite field.

We begin the project with the only factoring method that we know of that is faster than brute force (trial division), Pollard Rho, as we began researching different attacks on RSA that perhaps did not depend on factoring the 70 digit. While Pollard Rho did give us a small 3 digit prime 947, it did not mean that we were done. Firstly being that RSA did not necessarily only have 2 factors, and we checked the remaining 67 digits with the Miller-Rabin primality test, and we are 1/(4^32) certain that it is not a prime. And continuing the Pollard Rho on the remaining 67 digits for a day did not yield anything, and thus we shelved Pollard Rho and tried looking for other methods of breaking RSA.

# **NON-FACTORING ATTACKS ON RSA:**

We know factoring is hard, so our first action was to refer to “Twenty years of attacks on the RSA cryptosystem” [1]. On it there are a variety of different attacks against RSA, but unfortunately we cannot implement most of them. Attacks like Blinding, Hastad’s Broadcast Attack, and Related Message Attacks are not applicable in our case because they require intercepting and resending the RSA messages, but in our case all we were given was expected public information in RSA, and obviously could not repeat or redo the exchange of messages.

There are interesting attacks on how RSA is implemented on the computers, and exploitations, even though they don’t help because mainly we don’t have the access to how the professor created our RSA prompt. For example, there is a Timing attack that time a processors when it is decrypting to recover bits of the private key.

The ones that seemed promising were the low exponent attacks. Low Private Exponent attack if the private key was small. The consideration for not using this method is that small private key’s d were made for smart cards that needed to work efficiently to decrypt. Given the technology now, there is no reason to have a small d. There was no reason to assume or hope that the private d was small.

The other low exponent attack mentioned, the Low Public Exponent attack, seemed like the one that fit the bill. Unfortunately we are unable to implement this attack because of the complexity and the level of knowledge in linear algebra to understand how it works and to use it.

All in all we eventually locked in to breaking this “toy” RSA with factoring methods to find the private key.

**BASIS OF FAST FACTORING METHODS:**

What we found in our research [2] was that factoring methods better than just trial division takes root from Fermat's observation that you can represent composite numbers as a difference of squares. For example, take 91, and represent it as 100 - 9, which is 10^2 - 3^2, which factors out to (10 + 3) \* (10 - 3), which gives the factors of 91 to be 13 and 7. And every odd number N has this property; if N factors into ab, ab also has the property ab = (.5(a + b))^2 - (.5(a - b))^2, thus N can be written as a difference of 2 squares.

Maurice Kraitchk puts out that it would suffice that this difference of squares problem is a multiple of N, such that s^2 = t^2 (mod N), with s != t (mod N). Even though half of the solutions found will be uninteresting, st coprime to N, the other half is and will be the focus and eventually lead into the factors of N.

**SMOOTH NUMBERS:**

A problem is that searching for perfect squares is still too slow, especially when dealing with large numbers with 70 digits, because the squares of the factors will be in the range of the number that is being factored. In “Tale of 2 Sieves” [2], Kraitchk finds a way to create the 2 squares, mainly by finding numbers whose factorization is quite easy and then multiplying numbers together that make a perfect square, based on the exponents of the factorization of these easy to factor numbers. This leads to the search for smooth numbers.

Smooth numbers are numbers that prime factor completely to primes and their powers, where the primes are less than some bound B. This is useful because if we can limit the numbers to consider only numbers that are B smooth, they will be easy to factor. The question then becomes the distribution of such numbers in a given range.

In the beginning of [5], Dickman de Bruijn rho function is an estimation for the counting function of numbers up to n that are y smooth. But this approximation of the counting function is asymptotic, as x grows towards infinity, meaning that for our programs, if we were to bound our smooth numbers, we would need to guess and check the efficacy of a bound rather than have a definitive bound.

**POLLARD P-1:**

After getting the idea of how long the pollard’s rho factorization was going to take, we started looking for alternatives and came across pollard’s p-1. It is a number theoretic integer factorization algorithm, invented by John Pollard in 1974, and is described in [7], section 4.

Pollard’s p-1 factorization algorithm finds the factors for which p-1 is power smooth to some boundary.

Let n be a composite integer with prime factor p. By Fermat’s little theorem we know:

*a^k(p-1) = 1 (mod p)*

If a number x is congruent to 1 modulo a factor of n, then the gcd (x-1, n) will be divisible by that factor. Using the same phenomenon, if *gcd(2 ^ B! , n)*  is greater than one and less than p, we will have a factor p.

We select a bound and calculate gcd, and if it gives 1 , we pick the larger B value and run gcd again, and if it gives n back, we choose the smaller B value and run again.

The problem was , it was too time consuming and was neither giving one or N, so we had to look for another solution.

**DIXON FACTORIZATION:**

The Dixon method is described by the creator himself in [6]. We didn’t try to code this factorization method, but it was like a prerequisite to the quadratic sieve. The basic idea of this factorization is also the base for all the fastest factorization algorithms known so far and one of them being quadratic sieve.

i.e *x^2 = y^2 (mod n) and*

*x != +- y (mod n)*

Here, we knew that the number we’re factoring are way larger for this algorithm to come up with the factors, so we moved towards Quadratic Sieve.

**QUADRATIC SIEVE FACTORIZATION:**

Our next, and final goal, was to implement the quadratic sieve program to factor our 67 digit number, made with references to [2], [3] and [4]. We started off by implementing the program without sieving, just trying to get a working factor base up to a smoothness bound. Then, we tried obtaining smooth functions by doing trial division right away as soon as we tried to calculate:



Here, n is the digit we are trying to factor.

This was a mistake because the program would take forever to run, and after about 30 or 35 digit numbers, it would crash.

We knew we needed to optimize the program and make several improvements, so our next option was to try a better approach than trial division. We used our pollard rho factoring algorithm to give us the prime factors of the numbers we were trying to obtain the prime factorization of, but that would lead us to another problem. The pollard rho would give us composite numbers as factors, for instance one time it gave us 1943, which is divisible by 29. This would lead to an error in our program. So we just decided to go straight into sieving.

Using a factor base of primes that n is quadratic residue of, we ran each one of them into the Tonelli-Shanks algorithm to obtain the modular square root t1 and the other modular square root t2 which is equal to p - t1. Then, solving( t1 - sqrt(n) mod p) and (t2 - sqrt(n) mod p) gave us relations of a1 mod p and a2 mod p. Marking these indexes on our relation array denotes the numbers that are divisible by p. These are the numbers that are equal to a1 + i\*p and a2 + i\*p where i is 0, 1, 2, …, etc until the end of our interval. So we did this for all primes in our factor base (to which the Tonelli-Shanks algorithm would give us an answer to because sometimes it wouldn’t give us an answer and in that case we would terminate the Tonelli-Shanks in 1 second and proceed to the next one). We now have our sieving functions and all we need to do now is sieve.

We sieved with these functions and for every entry at a1 + i\*p and a2 + i\*p, we added log(p) to that entry. According to Carl Pomerance’s book titled “Prime Numbers: A Computational Perspective” [4], he referred to this way as a very efficient sieving method. So for these sieving relations we have for all these primes, we add log(p) to every index that we touch. After we finish sieving for all the primes in our factor base, we then check which one of our relation array indexes are greater than a certain threshold, and it is very likely that those numbers are smooth numbers. Pomerance says that a threshold of log( | x^2 - n| ) is suggested, also subtracting 20 or so from it because of the approximations of logarithms. We did some testing and it turns subtracting 25 from log( | x^2 - n| ) was an optimal threshold for us. But, even then, it isn’t guaranteed that all numbers are smooth. So after this, we did trial division on every “smooth” number with all the primes in our factor base, and if the number reduced to 1, it is smooth because it divides by only the primes in our factor base.

Now, we have smooth relations but it was taking us quite a while to check if they were really smooth because we didn’t have a better method than trial division, even though it is the slowest way. We were in the middle of generating our relations, and we got up to 85,000 actual smooth relations with all their factorizations, but we couldn’t finish due to a time constraint.

**LOOKING FORWARD:**

If we had more time, we would’ve solved this problem for sure. All we had to do was put our matrix (in field 2 because all we care about is the parity), and put it into Meataxe and get a span for the null space. Meataxe puts the span in a matrix transposed, meaning each solution vector is a row of the matrix. Regardless, as long as the input matrix itself didn’t have a nullity of 0, it was possible to find our 2 squares by multiplying the rows that were part of the solution in each of the null space vectors to find the factors of N. From there, we can check that each of the factors are prime, but the rest should be simple.

Phi(N) is easy to calculate, just the product of N’s primes factors after you subtract one from each of them, and then find the inverse of 5 (mod phi(N)), and the decrypted message is just the message raised to the inverse d (mod N).

Further decrypting the integer message to English plaintext would be simple. Assuming that the first 9 letters were converted without a leading 0, 0’s would become a focal point (the only frequent letter converted into a multiple of 10 would be T). From there any digit before it would have to attach to it, and any number greater than 2 after it would have to be a singular digit, representing letters third to ninth letter. The same of the latter applies in some sense that digits that are greater than 2 are cutoff points, meaning either the digit before them are 1 or 2, making the choices (1 or 2) then x or (1 or 2 x), but it stands to reason that a letter cannot be represented as number greater than 26.

**Quadratic Sieve Source Code:**

import java.math.BigInteger;

import java.util.\*;

import java.io.\*;

public class quadratic\_sieve {

public static BigInteger zero = new BigInteger("0");

public static BigInteger one = new BigInteger("1");

public static BigInteger two = new BigInteger("2");

public static BigInteger powersOfTwo(BigInteger n) {

BigInteger x = new BigInteger("0");

// while even

while(n.mod(two).equals(zero)) {

n = n.divide(two);

x = x.add(one);

}

return x;

}

public static BigInteger shanksTonelli(BigInteger n, BigInteger p, long start) {

// p - 1 = Q \* 2^S

BigInteger S = powersOfTwo(p.subtract(one));

// Q = (p - 1) / 2^S

BigInteger Q = (p.subtract(one)).divide(two.pow(S.intValue()));

// now we need to find a quadratic non-residue

// we will use euler's criterion

BigInteger z = zero;

BigInteger loop = one;

for(loop = one; loop.compareTo(p) == -1 ; loop = loop.add(one)) {

// if (i ^ ( p-1 / 2 ) ) mod p == p - 1

// that means it equals -1 mod p

// which makes i an quadratic non-residue

if(loop.modPow((p.subtract(one)).divide(two), p).equals(p.subtract(one))) {

z = loop;

break;

}

}

// let M = S, c = z^Q, t = n^Q, R = n ^ ( Q+1 / 2) all mod p

BigInteger M = S.mod(p);

BigInteger c = z.modPow(Q, p);

BigInteger t = n.modPow(Q, p);

BigInteger R = n.modPow((Q.add(one)).divide(two), p);

// if t = 0, return r = 0

// if t = 1, return r = R

// give while loop 100ms and if it doesn't compute, return 0

while( ( (System.nanoTime() - start) / 100 ) < 1000) {

if(t.equals(zero))

return zero;

if(t.equals(one))

return R;

int i = 1;

// while t^2^i != 1, increment i

while(!(t.pow(2).pow(i).mod(p).equals(one))) {

// repeated squaring to find the least i , 0 < i < M, such that t ^ 2 ^i = 1 mod p

i++;

}

BigInteger newI = new BigInteger(Integer.toString(i));

// b = c ^ 2 ^ M-i-1 mod p

// same as c\*c ^ M-i-1 mod p

// this line gave me problems when i was

BigInteger b = c.pow((int) Math.pow(2, M.intValue() - i - 1)).mod(p);

M = newI.mod(p);

c = (b.multiply(b)).mod(p);

t = (t.multiply(b).multiply(b)).mod(p);

R = (R.multiply(b)).mod(p);

}

return zero;

}

static BigInteger GCD(BigInteger m, BigInteger n) {

BigInteger zero = new BigInteger("0");

BigInteger r;

// if (m < n), swap m and n

if(m.compareTo(n) == -1) {

BigInteger temp = m;

m = n;

n = temp;

}

// while n != 0

while(!n.equals(zero)) {

r = m.mod(n);

m = n;

n = r;

}

return m;

}

static BigInteger pollardRho(BigInteger n) {

if(n.mod(two).equals(zero))

return two;

BigInteger five = new BigInteger("5");

if(n.mod(five).equals(zero))

return five;

BigInteger one = new BigInteger("1");

BigInteger base = two;

BigInteger constant = one;

BigInteger t = base;

BigInteger h = base;

// while d doesn't change, do steps

while(true) {

// g(x) = (x^2 + 1) mod n

// x = g(x)

t = ((t.pow(2)).add(constant)).mod(n);

// y = g(g(y))

h = ((h.pow(2)).add(constant)).mod(n);

h = ((h.pow(2)).add(constant)).mod(n);

// compute |x-y|

BigInteger abs = ((t.subtract(h)).abs());

BigInteger d = GCD(abs, n);

if(d.compareTo(one) == 1)

return d;

else if(d.equals(n)) {

base = base.multiply(two);

constant = constant.multiply(two);

t = base;

h = base;

}

else continue;

}

}

// method to clear array

static int[] clearAry(int[] ary) {

for(int i = 0; i < ary.length; i++) {

ary[i] = 0;

}

return ary;

}

static void factorize(ArrayList<BigInteger> factorBase, BigInteger n, BufferedWriter outFile) throws IOException {

// if n is less than 0, first vector element is 1

if(n.compareTo(zero) == -1)

outFile.write("1" + " ");

else

outFile.write("0" + " ");

for(int i = 0; i < factorBase.size(); i++) {

BigInteger factor = factorBase.get(i);

if(n.mod(factor).equals(zero)) {

int count = 0;

while(n.mod(factor).equals(zero)) {

n = n.divide(factor);

count++;

}

outFile.write(count%2 + " ");

}

else {

outFile.write("0" + " ");

}

}

outFile.write("\n\n");

}

static BigInteger smoothNumber(ArrayList<BigInteger> factorBase, BigInteger n) {

for(int i = 0; i < factorBase.size(); i++) {

BigInteger factor = factorBase.get(i);

if(n.mod(factor).equals(zero)) {

while(n.mod(factor).equals(zero))

n = n.divide(factor);

}

}

return n;

}

static int[] findPrimeFactorization(ArrayList<BigInteger> factorBase, BigInteger n){

int[] factors = new int[factorBase.size()];

for(int i = 0; i < factorBase.size(); i++) {

BigInteger factor = factorBase.get(i);

if(n.mod(factor).equals(zero)) {

factors[i]++;

n = n.divide(factorBase.get(i));

}

}

return factors;

}

public static void main(String[] args) throws IOException {

Scanner inFile = new Scanner(new FileReader(args[0]));

//BufferedWriter outSieve = new BufferedWriter(new FileWriter(args[1]));

BufferedWriter outSmooth = new BufferedWriter(new FileWriter(args[1]));

BufferedWriter smooth\_factorizations = new BufferedWriter(new FileWriter(args[2]));

BigInteger modulus = new BigInteger("8721717648951034180751989933592484457226804450832605710315649207114191");

// 947 and 9209839122440374002906008377605580208264841025166426304451583112053 are factors of modulus ^

//BigInteger factor1 = new BigInteger("9209839122440374002906008377605580208264841025166426304451583112053");

BigInteger factor1 = new BigInteger("9209839122440374002906008377605580208264841025166426304451583112053");

// bound = exp( (1/2) (log N log log N) ^ 1/2 )

double bound = 3000000;

BigInteger bigIntN = new BigInteger("9209839122440374002906008377605580208264841025166426304451583112053");

// System.out.println(bound);

// smoothness bound is 362 million as calculated above

// now we need to get all primes up to bound

// we will use the sieve of eratosthenes algorithm

boolean[] primes = new boolean[(int)bound + 1];

for(int i = 0; i < primes.length; i++)

primes[i] = true;

// from i = 2 to sqrt(bound)

for(int i = 2; i < (int)Math.sqrt(bound) + 1; i++) {

if(primes[i] == true) {

int j = i\*i;

while(j < bound) {

primes[j] = false;

j += i;

}

}

}

// factor base arraylist to store primes in factor base

// i used an arraylist to reduce space taken by an array using

// unnecessary values like composite numbers or primes that aren't in the factor base

ArrayList<BigInteger> factorBase = new ArrayList<BigInteger>();

for(int i = 2; i < primes.length; i++) {

if(primes[i]) {

BigInteger bigIntI = new BigInteger(Integer.toString(i));

if(bigIntN.modPow((bigIntI.subtract(one)).divide(two), bigIntI).equals(one))

factorBase.add(bigIntI);

}

}

// 2097483647

int intervals = factorBase.size()\*9500;

int[] relations = new int[intervals];

int[] relations\_negative = new int[intervals];

BigInteger x = bigIntN.sqrt().add(one);

/\*

System.out.print("Factor base for " + bigIntN + ": ");

for(int i = 0; i < factorBase.size(); i++)

System.out.print(factorBase.get(i) + " ");

\*/

System.out.println("\nNumber of prime factors: " + factorBase.size() + "\n");

for(int i = 0; i < relations.length ; i++) {

relations[i] = 0;

relations\_negative[i] = 0;

}

int smooth\_count = 0;

// keep track of end offset of sieving functions

int[] count1\_offset = new int[factorBase.size()];

int[] count2\_offset = new int[factorBase.size()];

int[] count1\_negative\_offset = new int[factorBase.size()];

int[] count2\_negative\_offset = new int[factorBase.size()];

int count1 = 0;

int count2 = 0;

int count1\_negative = 0;

int count2\_negative = 0;

// this function calculates the sieving relations for each prime

// i put all of these into a text file to make it quicker to run the program

// because it would take just 5 minutes to run all this because sometimes the Tonelli-Shanks program

// wouldn't return a square root so we had to wait for a second for it to finish. And over 100,000 primes,

// there was definitely a little bit of waiting

// the for loop aftet this block of code reads what this code outputs to a text file into 4 arrays

// and those arrays are the starting indexes for the sieving function for the first iteration

/\*

for(int i = 0; i < factorBase.size(); i++) {

System.out.println("HERE1 " + i);

BigInteger p = factorBase.get(i);

long start = System.nanoTime();

BigInteger root1 = shanksTonelli(bigIntN, p, start);

if(root1.equals(zero))

continue;

BigInteger root2 = p.subtract(root1);

BigInteger sieve1 = root1.subtract(x).mod(p);

BigInteger sieve2 = root2.subtract(x).mod(p);

count1 = sieve1.intValue();

count2 = sieve2.intValue();

count1\_negative = Math.abs(p.intValue()-count1);

count2\_negative = Math.abs(p.intValue()-count2);

count1\_offset[i] = count1;

count2\_offset[i] = count2;

count1\_negative\_offset[i] = count1\_negative;

count2\_negative\_offset[i] = count2\_negative;

}

\*/

for(int i = 0; i < factorBase.size(); i++) {

if(inFile.hasNext())

count1\_offset[i] = inFile.nextInt();

if(inFile.hasNext())

count2\_offset[i] = inFile.nextInt();

if(inFile.hasNext())

count1\_negative\_offset[i] = inFile.nextInt();

if(inFile.hasNext())

count2\_negative\_offset[i] = inFile.nextInt();

}

System.out.println(count1);

int interval\_count = 0;

// we want to generate factorBase size + 1 smooth relations to guarantee a solution to the null space

while(smooth\_count < factorBase.size() + 1) {

System.out.println("Smooth = " + smooth\_count);

for(int i = 0; i < factorBase.size(); i++) {

// load offset from previous iteration

count1 = count1\_offset[i];

count2 = count2\_offset[i];

count1\_negative = count1\_negative\_offset[i];

count2\_negative = count2\_negative\_offset[i];

// if tonelli shanks didnt return an answer, you cant sieve so next relation

if(count1 == -1) {

continue;

}

// while not at end of relation array

while(count1 < relations.length && count2 < relations.length && count1\_negative < relations.length && count2\_negative < relations.length) {

BigInteger p = factorBase.get(i);

// log p

int log = (int)Math.ceil(Math.log10(p.intValue()));

// add log p

relations[count1] += log;

// add log p

if(!p.equals(two))

relations[count2] += log;

// a1 + i\*p

count1 += p.intValue();

// a2 + i\*p

count2 += p.intValue();

// same for negatives

relations\_negative[count1\_negative] += log;

if(!p.equals(two))

relations\_negative[count2\_negative] += log;

count1\_negative += p.intValue();

count2\_negative += p.intValue();

}

// save offset of relations for next iterations

// so we know where to start sieving for the next interval

count1\_offset[i] = factorBase.get(i).intValue() - (relations.length - 1 - count1) - 1;

count2\_offset[i] = factorBase.get(i).intValue() - (relations.length - 1 - count2) - 1;

count1\_negative\_offset[i] = factorBase.get(i).intValue() - (relations.length - 1 - count1\_negative) - 1;

count2\_negative\_offset[i] = factorBase.get(i).intValue() - (relations.length - 1 - count2\_negative) - 1;

}

for(int i = 0; i < relations.length; i++) {

// threshold

int threshold = (int)Math.log10(Math.abs((i\*i) - bigIntN.doubleValue())) - 20;

// if index is over threshold

if(relations[i] > threshold){

// calculate sequence relation

BigInteger bigIntI = new BigInteger(Integer.toString(i));

BigInteger intervalsBig = new BigInteger(Integer.toString(intervals));

BigInteger intervalCountBig = new BigInteger(Integer.toString(interval\_count));

bigIntI = bigIntI.add( intervalsBig.multiply(intervalCountBig) );

BigInteger sequence\_n = ((x.add(bigIntI)).pow(2)).subtract(bigIntN);

// if its actually smooth

if(quadratic\_sieve.smoothNumber(factorBase, sequence\_n).equals(one)) {

// factorize it to text file

quadratic\_sieve.factorize(factorBase, sequence\_n, smooth\_factorizations);

// increase smooth count

smooth\_count++;

outSmooth.write(bigIntI + "\n");

}

}

}

//outSmooth.write("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");

for(int i = 0; i < relations\_negative.length; i++) {

//int threshold = (int)Math.log(i\*i - bigIntN.intValue());

int threshold = (int)Math.log10(Math.abs((i\*i) - bigIntN.doubleValue())) - 20;

if(relations\_negative[i] > threshold){

BigInteger bigIntI = new BigInteger(Integer.toString(-1\*i));

BigInteger intervalsBig = new BigInteger(Integer.toString(intervals));

BigInteger intervalCountBig = new BigInteger(Integer.toString(interval\_count));

bigIntI = bigIntI.add( intervalsBig.multiply(intervalCountBig) );

BigInteger sequence\_n = ((x.add(bigIntI)).pow(2)).subtract(bigIntN);

if(quadratic\_sieve.smoothNumber(factorBase, sequence\_n).abs().equals(one)) {

quadratic\_sieve.factorize(factorBase, sequence\_n, smooth\_factorizations);

smooth\_count++;

outSmooth.write("-" + bigIntI + "\n");

}

}

}

// go to next interval

interval\_count++;

// clear relation array for next interval

for(int i = 0; i < relations.length; i++) {

relations[i] = 0;

relations\_negative[i] = 0;

}

}

outSmooth.write("\n\n\n" + smooth\_count + " smooth relations");

outSmooth.close();

smooth\_factorizations.close();

//outSieve.close();

}

}

### 

### 

### References

1. Bone, D. “Twenty years of attacks on the RSA cryptosystem”. Notices of the American Mathematical Society, Vol. 46 (1999), pp. 203–213.
2. C, Pomerance. “A Tale of Two Sieves”. Notices of the American Mathematical Society, Vol. 43 (1996), pp. 1473-85.
3. Gerver, Joseph L. “Factoring Large Numbers with a Quadratic Sieve.” Mathematics of Computation, Vol. 41 (1983) , pp. 287–94.
4. Crandall, R. E., & Pomerance, C. “Chapter 6.1 Quadratic Sieve Factorization Method.” Prime Numbers: A Computational Perspective (2nd ed.)(2005), pp. 261-276. Springer.
5. Granville, A. “Smooth Numbers: Computational Number Theory and Beyond.” Algorithmic Number Theory, Vol. 44 (2008). MSRI Publications.
6. Dixon, J. D. “Asymptotically Fast Factorization of Integers.” Mathematics of Computation, Vol. 36 (1981), pp. 255–60.
7. Montgomery, P. L. “Speeding the Pollard and Elliptic Curve Methods of Factorization.” Mathematics of Computation, Vol. 48 (1987), pp. 243–64.