Public-Key Cryptography

EE478 Prof. Hellman

February 1995

These notes are intended to supplement the lectures but are not intended as a stand-alone text. If you try to follow them before the lecture, you are likely to have difficulty in places. Similarly, if you miss a lecture, it will be important to catch up from someone else in the class.

1.0 The Systems Studied

We will study primarily the Diffie-Hellman public-key distribution system, the RSA public-key cryptosystem, system, ElGamal's digital signature system, and the Digital Signature Standard (DSS), which is a variation of ElGamal's system.

Make sure you read the relevant background material on these systems distributed in class, as these notes assume you are familiar with the basic operation of the systems:

- In Diffie and Hellman's "Privacy and Authentication," sections II-E, II-F, III-K, and III-L.
- All of Rivest, Shamir and Adleman's paper.
- NIST's DSS.

Some good references for background are:

- Gallager, *Information Theory and Reliable Communication*, sections 6.3 (Group Theory), 6.4 (Fields and Polynomials) and 6.6 (Galois Fields). Gallager uses an engineering-oriented approach to groups and fields for developing algebraic error-correcting codes, but it is also very useful for public-key cryptography.
- Niven and Zuckerman, *An Introduction to the Theory of Numbers*, is a typical text for a first course in number theory.
- Knuth, *The Art of Computer Programming*, volume 2, section 4.5 covers Euclid's algorithm and factoring methods.
- Bressoud, *Factorization and Primality Testing*, is a good, elementary treatment of many of the number theoretic concepts needed for factoring algorithms, but stops short in his analysis compared to our needs.

2.0 Why RSA Works

In the RSA paper,

$$C = M^E \ \% \ n \tag{1}$$

$$M = C^D \ \% \ n \tag{2}$$

$$D = E^{-1} \ \% \ \phi(n) \tag{3}$$

where $n = pq$ and $\phi(n) = (p-1)(q-1)$ is the number of integers between 1 and $n–1$ which have no common factors with n.[1] $\phi(n)$ is known as Euler's totient function.

The real question is why (1) and (2) are inverse operations. Substituting (1) into (2), we find M must equal $M^{ED}\%n$. This turns out to be true via (3) because arithmetic in the exponent is done modulo $\phi(n)$. The rest of this section is devoted to the math required to show why that is the case. So, I beg your indulgence while we delve into groups, Galois fields, and related topics. They really are useful!

1. Every p^{th} number has p as a factor, so $(p–1)/p$ of the numbers are not ruled out by p. Similarly, $(q–1)/q$ of the numbers are not ruled out by q, leaving a fraction $(p–1)(q–1)/pq$ of the $n=pq$ numbers, for $(p–1)(q–1)$ in all.

Definition: A group is a set of elements and an operator denoted by • such that: The group is closed, the associative law holds, there is an identity element e, and every element has an inverse. If $a \bullet b$ always equals $b \bullet a$ then the group is called Abelian or commutative.

Definition: A field is a set of elements an addition operation and a multiplication operation such that: The elements form an Abelian group under addition, the *non-zero* elements form an Abelian group under multiplication, and the distributive law holds: $(a + b)c = ac + bc$. As usual, the additive identity is denoted by 0, the multiplicative identity by 1, the additive inverse of a by $-a$, and the multiplicative inverse of a by a^{-1}.

Theorem: The set $\{1, 2, ..., n–1\}$ forms a field under arithmetic modulo n if and only if n is prime.

Proof: If you have never seen this theorem before, I suggest you first try some small examples, such as $n = 5$ and $n = 6$, to see that it works before worrying about the proof. For example $2\times3 = 1\%5$ so $2^{-1} = 3$ and $3^{-1} = 2$. It takes a little getting used to that "one-half" is 3, and "one-third" is 2.

Addition modulo n is always an Abelian group $(–a = n–a)$ and the distributive law always holds. Multiplication modulo n is always closed and associative, and 1 is always the multiplicative identity. The only question is whether or not every non-zero element has a multiplicative inverse.

If $n = 6$ then $2\times3=0$ %6, so 2 and 3 cannot have multiplicative inverses. Otherwise we could multiply both sides of $2\times3=0$ by 2^{-1} to find that 3=0. The same idea shows that whenever n is not prime, any number that divides n cannot have a multiplicative inverse. To complete the proof, we need to show that whenever $n = p$, a prime number, then every non-zero element has a multiplicative inverse. We now do that constructively, by developing an algorithm for finding a^{-1} for any non-zero element a. §

(The symbol § denotes the end of a proof or other block, when needed for clarity) Euclid's Algorithm finds the GCD (greatest common divisor) of two numbers, x and y, very rapidly. An extended version of the algorithm (Knuth, volume 2, 2nd edition, page 325) also finds two multipliers u_1 and u_2 such that

$$u_1 x + u_2 y = \mathrm{GCD}(x, y) \equiv u_3$$

Since $\mathrm{GCD}(p,a) = 1$ whenever p is prime and $a \neq 0$ %p (i.e., a is not a multiple of p), we can use the extended algorithm to find solutions to

$$u_1 p + u_2 a = 1$$

in which case $u_2 = a^{-1}$ %p. Henceforth, I only differentiate between the basic and extended versions of Euclid's algorithm when needed for clarity.

The pseudo-program shown below implements Knuth's version of Euclid's algorithm applied to finding multiplicative inverses in modular arithmetic. Once you believe that this algorithm works, you have completed the proof of the above theorem that arithmetic modulo p forms a field. (We do not need u_1, so it could be deleted.)

Inverses in Modular Arithmetic

```
INPUT "Enter A to be inverted"; A
INPUT "Enter modulus N"; N
U = (U1,U2,U3) = (1,0,N)
V = (V1,V2,V3) = (0,1,A)

while(V3≠0)
    Q = INT(U3/V3)
    U = U - QV
    SWAP U,V

if (U3=1)
    PRINT "Inverse = "; U2%N
else
    PRINT "No inverse, GCD="; U3
```

Justification for Basic Euclidean Algorithm: If g divides x and y, it also divides $x+y$ and $x–y$. For example, 5 divides 100 and 15, so it also divides 100–15 and 100+15. Similarly it divides $100–15k$ for any integer k. The above algorithm merely takes the largest value of k which leaves a non-negative answer. Here, $k = 6$ leaves 10 as the remainder. Hence we can reduce the problem of finding GCD(100,15) to the problem of finding GCD(15,10). Because the remainders u_3 (or U3 in the box above) are monotone decreasing, the algorithm must eventually terminate with the u_3 = GCD.

Justification for Extended Euclidean Algorithm: Because the initial values satisfy $u_1 n + u_2 a = u_3$ and $v_1 n + v_2 a = v_3$ any linear combination of the vectors ***u*** and ***v*** also satisfies such an equation. (The algorithm forms such linear combinations.) When $u_3=1$, then $u_2=a^{-1}$%n.

While the following theorem is not needed for our immediate task of proving that arithmetic modulo a prime number p is a field, it will prove useful in the future. It says that Euclid's algorithm runs very fast: 100-digit numbers require at most 479 iteration, and 1000-digit numbers require at most 4,785 iterations. Immediately following it, we return to proving that arithmetic modulo p is a field.

Theorem: If $0 \le u, v < N$, then the number of iterations of the "while loop" in Euclid's algorithm is at most

$$\lceil \log(\sqrt{5}N) \rceil - 2 \qquad \textbf{(4)}$$

where the logarithm is to the base

$$\phi = \frac{\sqrt{5}+1}{2} = 1.618\ldots \qquad \textbf{(5)}$$

ϕ is known as the golden ratio because a rectangle whose sides are in the ratio ϕ:1 has been found in architecture to be "most pleasing" to the eye. §

Proof: The proof goes beyond what we need. If interested, see Knuth, volume 2, 2nd edition, Corollary L, page 343.

Definition: A subgroup is a subset of elements of a group which itself forms a group. The even integers are a subgroup of the integers under addition.

Definition: The order of a group is the number of elements contained therein.

Theorem: The order of a subgroup divides the order of the parent group.

Proof: If interested, see Gallager, page 210, Theorem 6.3.1. It's not too hard to prove using the concept of cosets.

Corollary: $a^{p\text{-}1} = 1 \% p$ provided $a \neq 0 \%$p.

Proof: Let a be a non-zero element under arithmetic modulo p. Because there are only p elements in arithmetic modulo p, the set $\{a^k \% p\}$ must eventually repeat as k goes through the set of positive integers. We can therefore find k and m such that $a^k = a^{k+m}$. Multiplying both sides by $(a^k)^{-1}$ shows that then $a^m = 1$.

You can now show that $\{a^k \% p\}$ forms a subgroup under multiplication modulo p and that the order of the subgroup is the least m such that $a^m = 1$. m must divide the order, p-1, of the parent group, so for some c, $a^{p\text{-}1} = (a^m)^c = 1^c = 1$. §

Corollary: $a^p = a \% p$ for all elements a.

Proof: $0^p = 0 \% p$.

Definition: A Galois field, denoted GF(m), is a field with a finite number m of elements.

Theorem: For every prime number p and integer $k \geq 1$, there is a Galois field GF(p^k). There are no others. When k=1, GF(p) is isomorphic to (indistinguishable from) arithmetic modulo p (i.e., the field can be transformed to arithmetic modulo p by relabelling the elements). When $k>1$, GF(p^k) is isomorphic to arithmetic modulo an irreducible polynomial of degree k. (If you don't know what that means, don't worry. We won't need it. Gallager is a good reference.)

Proof: If interested, see Gallager, page 230, Theorem 6.6.5.

Definition: a is a primitive element of GF(m) if the set $\{a^k\}_{k=0}^{m\text{-}1}$ includes every non-zero field element. (By the above theorem, m must be a power of a prime.)

Theorem: Every finite field possesses a primitive element.

Proof: If interested, see Gallager, page 226, Theorem 6.6.1. While we will not prove this theorem, we will use it extensively. It is very useful.

Chinese Remainder Theorem: Let $m_1, m_2, \ldots, m_r$ be positive integers that are relatively prime in pairs. Letting $m = \Pi m_i$, there is exactly one integer u between 0 and $m-1$ that satisfies $u_i = u \% m_i$.

Note: The history of this theorem goes back to ancient China, perhaps as early as the third century A.D.

Proof: We will only prove the theorem for r=2, but the ideas are similar for larger r. The following table, which shows u ranging from 0 to 11 and u%2 and u%3 basically gives the proof.

u	0	1	2	3	4	5	6	7	8	9	10	11
u%2	0	1	0	1	0	1	0	1	0	1	0	1
u%3	0	1	2	0	1	2	0	1	2	0	1	2

The proof is by contradiction: Let $\mathbf{v}(u)$ denote the pair $(u\%m_1, u\%m_2)$ and suppose $\mathbf{v}(i) = \mathbf{v}(j)$ with i and j both between 0 and m–1. This is equivalent to $(i\%m_1, i\%m_2) = (j\%m_1, j\%m_2) \Rightarrow (i-j)\%m_1 = (i-j)\%m_2 = 0 \Rightarrow$ both m_1 and m_2 divide $i-j \Rightarrow m = m_1 m_2$ divides $i-j \Rightarrow i-j \geq m$, contradicting the assumption that i and j are both between 0 and m–1. §

Algorithm for Chinese Remainder Theorem: In the RSA system we only need the CRT for m_1 and m_2 being prime numbers, p and q. If we know $x = u\%p$ and $y = u\%q$, we need to be able to compute $z = u\%n$. This is easily done via

$$z = (q^{p-1}x + p^{q-1}y) \% n \quad (6)$$

Justification: We must show that $z\%p=x$ and $z\%q=y$. Since p and q are prime, $q\%p \neq 0$ and $p\%q \neq 0 \Rightarrow q^{p-1}=1\%p$ and $p^{q-1}=1\%q$. Using $p=0\%p$ and $q=0\%q$, we then see that $z\%p=x$ and $z\%q=y$. §

Note: Because exponentiation can be carried out rapidly similar to

$$a^{10} = (a^2)((a^2)^2)^2 \quad (7)$$

the above algorithm is very efficient, even for 100 or 1,000 digit numbers.

We are now in a position to complete the proof that equations (1) and (2) are inverse operations if E and D are related by (3). We start by noting that (3) is equivalent to

$$ED = k\phi(n) + 1. \quad (8)$$

Theorem: If $n = pq$ then, for all $0 \leq M \leq n-1$,

$$M^{k\phi(n)+1} \% n = M. \quad (9)$$

Proof: First consider the case where M is relatively prime to p and q (i.e., $M \neq 0\%p$ and $M \neq 0\%q$). Then, letting X denote $M^{k\phi(n)+1} \% n$, we find that

$$X\%p = M^{k(p-1)(q-1)+1} = [M^{p-1}]^{k(q-1)}M = 1^{k(q-1)}M = M\%p$$

and similarly $X\%q = M\%q$. Using the CRT, this implies that $X = M\%n = M$.

When M is not relatively prime to p or q (i.e., $M=0\%p$ or $M=0\%q$), the proof proceeds similarly. If $M=cp$ then $X\%p=M\%p = 0$ and the above argument still applies. §

3.0 Factoring Algorithms

We can compute our secret key D via equation (3) because we know the factorization of n and can therefore compute $\phi(n) = (p-1)(q-1)$. If an opponent were able to factor n, he would be in the same position as we, and could compute D from our public key (E,n). Factoring large integers was of great interest in pure mathematics even prior to the development of the RSA system, but that innovation has given an added impetus and practical connotation to the work.

3.1 Semi-Exhaustive Search

The simplest factoring algorithm is to try dividing n by all smaller integers. One quickly realizes that the effort can be reduced considerably for two reasons:

- If a prime p does not divide n, then neither does any multiple of p. Hence the search for divisors can be restricted to primes.
- No more than one prime greater than $\sqrt{n}$ can divide n. Hence the search can be further restricted to primes less than $\sqrt{n}$.

Example 1: In trying to factor 103, we must try 2, 3, 5, and 7 (none of which divide 103) and can stop there because $11^2>103$. Therefore 103 is prime.

Example 2: In factoring 6489, we find that 6489 is divisible by 3^2, leaving 721 as the quotient; 721 is divisible by 7, leaving 103; 103 is not divisible by 11 and we can stop there because $11^2>103$. (An extension of the second point above allows us to stop at primes greater than the square root of the quotient left. Of course, here we also knew that 103 was prime from the first example.)

While the above reduction considerably cuts the time to factor a number, the algorithm is still is impractical for all but small examples. If, for example, n is 100 digits long, then the above method can take on the order of 10^{50} trial divisions. (While not all numbers less than $\sqrt{n}$ are prime, the difference in the exponent is not that great, reducing to approximately 10^{48} because approximately $1/(\ln n)$ of the

numbers between 1 and n are prime.) The following theorem is very useful in working with RSA.

Theorem: Let $\pi(x)$ denote the number of primes less than or equal to x. Then

$$\lim_{x \to \infty} \frac{\pi(x)}{x} \ln(x) = 1 \qquad (10)$$

Proof: Beyond the scope of this course. See Knuth, volume 2, 2nd edition, page 366 for a partial discussion. He gives the following table which shows good agreement between the approximation and the actual values.

Table 1: Distribution of Primes

x	$\pi(x)$	$x/\ln(x)$
10^3	168	144.8
10^6	78,498	72,382.4
10^9	50,847,534	48,254,942.4

3.2 Fermat's Method

The method of section 3.1 can run very rapidly (e.g., factoring 2^{1000}), but is typically very slow. Its worst case is when $n=p^2$, in which case the running time is $\sqrt{n}$. It might appear, therefore, that the RSA system would be strongest when $p \approx q$. However, a factoring algorithm due to Fermat makes short work of factoring n in such cases.

The basic idea is to search over $p-q$ since that quantitiy is small when $p \approx q$. Letting $x = (p+q)/2$ and $y = (p-q)/2$,

$$x^2 - y^2 = \frac{(p^2 + q^2 + 2pq) - (p^2 + q^2 - 2pq)}{4} = n \qquad (11)$$

so that $n+y^2$ must be a perfect square, namely x^2.

Hence, in factoring n=391, we can set up a table as shown below and search over y until the square root of $n+y^2$ is an integer. Further reduction in computation is clearly possible by using a tangent (or better) approximation to predict when the last row will hit the next integer, thus allowing us to skip many values of y.

y	0	1	2	3
$n+y^2$	391	392	395	400
$\sqrt{n+y^2}$	19.77	19.80	19.87	20.00

Knuth (volume 2, 2nd edition, page 371) discusses other improvements.

Fermat's algorithm is the reason that most RSA systems choose p and q to be of slightly different lengths, for example 249 and 251 bits if a 500-bit product n is desired. This guarantees that $y=(p-q)/2$ will be at least 2^{500} (much too large to search over), and also protects against any, as yet discovered, algorithms that work if p/q is close to one.

3.3 The Value of Knowing $x^2 = y^2$ Modulo n

Fermat's method searches for a pair (x,y) which satisfies $x^2=n+y^2$. The next several sections describe fast factoring algorithms that are generalizations of Fermat's in that they search for x and y which satisfy $x^2=kn+y^2$, or equivalently $x^2=y^2\%n$. To see why this helps, rewrite $x^2-y^2=kn$ as

$$(x-y)(x+y) = kpq \qquad (12)$$

and note that p must divide either $(x-y)$ or $(x+y)$, and so must q. There are four possibilities:

1. p and q both divide both $(x-y)$, neither divides $(x+y)$, so $GCD(x-y,n)=pq=n$ and $GCD(x+y,n)=1$.
2. neither divides $(x-y)$, p and q both divide both $(x+y)$, so $GCD(x-y,n)=1$ and $GCD(x+y,n)=pq=n$.
3. p divides $(x-y)$ and q divides $(x+y)$, so $GCD(x-y,n)=p$ and $GCD(x+y,n)=q$.
4. q divides $(x-y)$ and p divides $(x+y)$, so $GCD(x-y,n)=q$ and $GCD(x+y,n)=p$.

Theorem: Each of the above four possibilities occurs 1/4 of the time as x and y range over all possible non-zero solutions to $x^2=y^2\%n$ so, if we can find a random such pair, we can factor n half of the time.

Proof: By the CRT, $x^2=y^2\%n \Leftrightarrow x^2=y^2\%p$ and $x^2=y^2\%q \Leftrightarrow x=\pm y\%p$ and $x=\pm y\%q$ where $-y=p-y$ in mod-p arithmetic and $-y=q-y$ in mod-q arithmetic. When $x=y\%p$ and $x=y\%q$ then, by the CRT, $x=y\%n \Leftrightarrow n$ divides $(x-y)$.

Similarly, when x=-y%p and x=-y%q , x=-y%n ⇔ n divides (x+y). In cases #3 and #4, $x \neq y$%n and x=-y%n so the GCD's are as shown.

3.4 Dixon's Method

Although Dixon's method appeared after several related, more efficient, algorithms, I start there because it is the simplest method for generating x and y such that x^2=y^2 %n. The idea is best illustrated by example. Let n=391, so that p=17 and q=23 but, for now, pretend we do not know the factorization.

First, generate a sequence of random variables, uniformly distributed between 1 and n–1=390. To do this, I used a portion of Rand's book "A million random digits with 100,000 normal deviates." (That really is the title!) The following table shows the first six random numbers I found, their squares modulo-n, and the factorizations thereof:

Random x_i	347	111	036	122	175	264
x_i^2%n	372	200	123	026	127	098
factors	$2^2 3^1 31$	$2^3 5^2$	$3^1 41$	$2^1 13$	127	$2^1 7^2$

Using the second and sixth relations above, we find that in mod-391 arithmetic:

$$111^2\ 264^2 = (2^3 5^2)\ (2^1 7^2) = 2^4 5^2 7^2 = (2^2 5^1 7)^2 \qquad (13)$$

111×264=370 and $2^2 5^1 7$=140, so x^2=y^2%n with x=370 and y=140. By the last theorem, this relation gives a 50 percent chance of factoring n. Computing GCD(x±y, n),

GCD(370-140, 391)=23 and GCD(370+140, 391)=17

we find that we are in luck and have factored n into 23×17.

This example contains almost all the ideas needed for Dixon's algorithm, with one key exception: Completely factoring each x_i^2%n would be as hard as factoring n, so a different approach is needed. Instead, we factor x_i^2%n only if it factors into "small" primes. In the above example, there is little point in keeping the equation that results from the third value, 36^2%n =3×41, because we are unlikely to find 41 as a factor of another x_i^2%n, and are therefore unlikely to be able to produce an even exponent by multiplying this equation with another. With this note and the above example as a guide, the following algorithm should make sense.

1. Set i=1.

2. Generate a random variable $x_i \sim U(1, n-1)$ which is statistically independent of all previous x_i's.

Compute x_i^2%n and check if it can be written as a product of small primes $\{p_k\}_{k=1}^K$. (p_1=2, p_2=3, p_3=5, p_4=7, p_5=11, and p_K is the largest "small" prime.) If so, let $\boldsymbol{e}_i$ denote the vector of exponents so that

$$x_i^2 \% n = \prod_{k=1}^{K} p_k^{e_{i,k}} \qquad (14)$$

If x_i^2%n is smooth, go to step #3, else return to the beginning of step #2 with the value of i unchanged.

3. If i=K+10 go to step #4, else set i=i+1 and go to step 2. (As will be seen, the value K+10 is somewhat arbitrary. Any number somewhat greater than K would work.)

4. Set up the K+10 by K, binary matrix whose i^{th} row is the vector of exponents $\boldsymbol{e}_i$ each taken mod-2, and find the first 10 linear dependences. Because addition of exponents corresponds to multiplication, each dependence gives an equation of the form

$$\prod_i x_i^2 = \prod_{k=1}^{K} p_k^{c_{i,k}} \quad \%n \qquad (15)$$

where the exponents $c_{i,k}$ are all even. (A linear dependence gives a result of zero which, in mod-2 arithmetic, equates to an even value.) Hence, each equation of the form (15) corresponds to x^2=y^2%n with

$$x = \prod_i x_i \quad \text{and } y = \prod_{k=1}^{K} p_k^{c_{i,k}/2} \qquad (16)$$

Because each dependence has a 50 percent chance of factoring n, the probability that none of the 10 dependences factors n is 1/1024. (This is why step #3 noted that K+10 was somewhat arbitrary. All that is needed is that the value be approximately equal to K. If it is much smaller, we do not expect any dependences. If it is much larger, we have more dependences than needed. Whether we need K–10 or K+10 does not affect asymptotic performance of the algorithm as $n \to \infty$.). §

3.5 Analysis of Dixon's Method

The effort required by Dixon's factoring method depends on K, the number of "small" primes. If K is small, the prob-

ability that $x_i^2\%n$ is smooth in step #2 is small, and the algorithm labors hard to find each success (i.e., a smooth $x_i^2\%n$). If K is large, the algorithm does not have to work hard for each success, but takes a long time because a large number (approximately K) of successes are needed. The optimal value of K balances the effect of these two conflicting trends, and minimizes the expected computational effort to factor n. In optimizing, we will not worry about factors, such as multiplicative constants, which do not affect the asymptotic form of the required effort. As we shall see, that means neglecting much more than just multiplicative constants.

Ω, the expected computational effort required to factor n can be approximated

$$\Omega \cong \{K \Pr^{-1}(\text{success})[K \ln(n) + \ln(n)]\} + K^3 \qquad \textbf{(17)}$$

The final term, K^3, is the time required to find the dependences by Gaussian elimination on the $K \times K$ binary matrix. Because the matrix will be sparse, faster methods are possible. While improving this term is of importance with more efficient algorithms, it will turn out not to affect the asymptotic performance of Dixon's method.

Turning to the rest of the expression, the first factor of K comes from the need for approximately K successes (e.g., K+10 in the previous section). $\Pr^{-1}$(success) is the average number of iterations of step #2 required per success.

Turning to the two terms in brackets, $K \ln(n)$ is the effort required for trying to divide $x_i^2\%n$ by the K small primes in step #2. This makes the technically incorrect assumption that each small prime will be a single precision variable, but at worst, this term is $K \ln(n) \ln\ln(n)$, and as we shall see in the next section, logarithmic terms can be neglected. The second term in brackets, $\ln(n)$, approximates the time required to compute $x_i^2\%n$. Again, technically, it should be $\ln(n) \ln\ln(n)$, using FFT techniques, but can be neglected for reasons developed in the next section.

From equation (17) we see that the next step is to evaluate Pr(success) in terms of K and n. That requires some background information:

Definition: $\Psi(X,Y)$ is the number of positive integers which are less $\leq X$ and free of prime divisors $>Y$. Thus, for example $\Psi(10,3) = 7$ because 1, 2, 3, 4, 6, 8, and 9 can be written as a product of the numbers {1,2,3}, while 5, 7 and 10 cannot.

Theorem: $\Psi(X,Y)/X = u^{-u+o(u)}$ where $u = \ln(X)/\ln(Y)$, provided $X \geq 10$ and $Y \geq (\ln(X))^{1+\varepsilon}$ for some $\varepsilon > 0$.

Proof: If interested, see N. G. deBruijn, "On the number of positive integers ≤ x and free of prime factors >y," II, Nederl. Akad. Wet. Proc. Ser. A 69 = Indag. Math, vol 38, pp 239-247, 1966.

Note 1: Since the length of a number in digits or bits is proportional to its logarithm, the above theorem says that it is only the ratio of the lengths of X and Y which matter asymptotically. The probability that a random 100 digit number factors into primes of at most 10 digits, or that a random 500 digit number factors into primes of at most 50 digits is approximately 10^{-10}.

Note 2: This theorem combines what is usually two or three steps. I will describe them mostly for completeness so that, if you run into any of the notation, you will not be taken by surprise. Usually it is first shown that

$$\lim_{Y \to \infty} \frac{\Psi(Y^u, Y)}{Y^u} = \rho(u) \qquad \textbf{(18)}$$

$\rho(u)$ is known as Dickman's rho function and, for A=1,2,3… and $A < \rho(u) \leq A+1$, satisfies

$$\rho(u) = \rho(A) - \int_A^u \frac{\rho(v-1)}{v} dv \qquad \textbf{(19)}$$

Equation (19) can, in theory[1], be integrated to obtain $\rho(u)$ for any u because $\rho(u) = 1$ for all $u \leq 1$. (Any number n can be factored into "small" primes if the upper bound on the small primes is bigger than n.) Hence, for example, when $1 < \rho(u) \leq 2$, $\rho(u) = 1 - \ln(u)$. The last step is to show that for any $\varepsilon > 0$

$$\lim_{u \to \infty} (1/u^u)^{1+\varepsilon} \leq \rho(u) \leq (1/u^u)^{1-\varepsilon} \qquad \textbf{(20)}$$

The following table compares several values of $\rho(u)$ with the asymptotic form u^{-u}: Note that, even though $\rho(u)$ becomes very small, its ratio with u^{-u} becomes close to 1.

1. The equation is numerically unstable.

u	1	1.5	2	2.5	3	4	5
$\rho(u)$	1	0.594	0.307	0.130	0.0486	0.0049	0.00035
$u^u\rho(u)$	1	1.09	1.23	1.28	1.31	1.25	1.11

§

With the above background, Pr(success) $\cong u^{-u}$ where $u=\ln(n)/\ln(p_K)$ or, equivalently, $p_K \cong n^{1/u}$. From (10), $K \cong p_K/\ln(p_K)$ so, temporarily assuming that the final K^3 term in (17) is negligible:

$$\Omega \cong [p_K/\ln(p_K)]^2\, u^u \ln(n)]$$

$$= (u^{u+2}\, p_K^{\ 2}) / \ln(n) = (u^{u+2}\, n^{2/u}) / \ln(n) \qquad (21)$$

$$\omega = \log\Omega \cong (u+2)\log u + \frac{2\log n}{u} - \log\log n \qquad (22)$$

(Note: log is the same as ln. My word processor has a glitch whereby I cannot get ln in some places.) To find the minimum, set the derivative of (22) to zero:

$$\frac{u+2}{u} + \log u - \frac{2\log n}{u^2} \cong 0 \qquad (23)$$

The first term is asymptotically negligible, so

$$2\ln(n) \cong u^2 \ln(u). \qquad (24)$$

At first it might seem that we could neglect the $\ln(u)$ since it is so much smaller than u^2. However, that turns out not to work, and the error term suggests that the optimal u is of the form

$$u \cong \sqrt{\frac{\beta\log n}{\log\log n}} \qquad (25)$$

(If we were wrong, the optimal value of β would either be infinite, indicating that (25) grows too slowly, or 0, indicating that it grows too fast.) Plugging (25) into (24) gives

$$2\log n \cong \left[\frac{\beta\log n}{\log\log n}\right]\left[\frac{\beta + \log\log n - \log\log\log n}{2}\right] \qquad (26)$$

The numerator of the second term in brackets can be reduced asymptotically to $\ln\ln(n)$, cancelling the denominator of the first term in brackets so, for the optimal β,

$$\beta^*\ln(n) = 4\ln(n) \;\Rightarrow\; \beta^* = 4 \qquad (27)$$

Substituting (27) into (24) and then using that in (22) while deleting terms that are asymptotically negligible (e.g., $u+2 \cong u$, and $\ln(n)+\ln\ln(n) \cong \ln(n)$)

$$\omega \cong 2\sqrt{\frac{\log n}{\log\log n}}\;\frac{1}{2}(\log 4 + \log\log n - \log\log\log n) + 2(\log n)\sqrt{\frac{\log\log n}{4\log n}} - \log\log n \qquad (28)$$

$$\omega \cong \sqrt{4(\log n)\log\log n} \qquad (29)$$

$$\Omega \cong \exp\sqrt{4(\log n)\log\log n} \qquad (30)$$

This is of the form $\exp\sqrt{k(\log n)\log\log n}$ with k=4. We now justify our assumption that K^3 in (17) is negligible compared to the first term. That reduces to

$$K^3 << K^2(\log n)\, u^u \qquad (31)$$

which is true because $K << u^u$. (I'll leave it to you to check that.)

Note 1: The next section presents a simipler derivation of the $\exp\sqrt{k(\log n)\log\log n}$ behavior, but pulls the bound K on the largest small prime "out of a hat." The derivation of this section gives some understanding of why that bound works so well.

Note 2: Knuth, vol. 2, 2nd edition, page 383, has what might be a simpler proof for the continued fraction method, but I believe it has the wrong value of k. (Knuth has k=4, while the continued fraction method should have k=2 for reasons soon to be discussed.) In going over Knuth's derivation, it appears that he has not optimized properly over K. He uses $K = (1/2)[\ln(n)\ \ln\ln(n)]^{1/2}$ while, following his approach, I found the optimal value to be $[1/8^{1/2}][\ln(n)\ \ln\ln(n)]^{1/2}$. I would appreciate any feedback you may have. (e.g., Did I make a mistake?) Knuth has a simple bound on Pr(success) that is trivial to derive and yet seems to be adequate for the derivation. He gives it as an exercise, and so will I:

Exercise: Prove that Pr(success) > $K^u/(u!\ n)$.

Note 3: While the above proof is not rigorous, Dixon's algorithm can be proved rigorously to have computation time of the form $\exp\sqrt{k(\log n)\log\log n}$. That is the main reason it is of interest, even though it has k=4 while earlier algorithms had smaller values of k. The most difficult part of

making the proof rigorous is to change the theorem about the density of smooth numbers within the set $[1, n-1]$, so that it deals with the density of smooth numbers within the set of quadratic residues modulo-n.

Note 4: For rigorous proofs with smaller values of k, see Vallée ["Generation of Elements with Small Modular Squares and Provably Fast Integer Factoring Algorithms," *Mathematics of Computation*, **56**, pp. 823-849, 1991] and her reference to work of Lenstra and Pomerance.)

Note 5: Richard Schroeppel was the first to observe that $\exp\sqrt{k(\log n)\log\log n}$ was a possible running time for factoring algorithms. Such algorithms already existed (e.g., the continued fraction method has such a running time with k=2), but no one had shown they were subexponential. Schroeppel also was the first to recognize that sieving could be applied to greatly reduce the run-time to k=1. (Sieving will be discussed later. It usually halves k.) Most of the literature I have read shortchanges Schroeppel's contribution, probably due to his prejudice against publishing papers.

3.6 Background for a Simpler Derivation

There is a simpler derivation of equation (30) which I now present using the notation of Coppersmith, Odlyzko and Schroeppel (denoted COS hereafter; D. Coppersmith, A. Odlykzo, and R. Schroeppel, "Discrete Logarithms in GF(p)," *Algorithmica*, v. 1, n. 1, 1986, pp. 1-16). This derivation depends critically on COS's Theorem 1 which I paraphrase below, without the epsilons and deltas that make it rigorously correct, but harder to understand.

Definition: Let $L(c) = \exp\{c[1+o(1)][\ln(n)\ln\ln(n)]^{1/2}\}$ so that $L(c_1)L(c_2) = L(c_1+c_2)$, $L(c)^k = L(kc)$, $L(c_1)+L(c_2) = L(\max\{c_1,c_2\})$, etc.

Theorem: If $p_K = L(\beta)$ and a "success" occurs when a number of size n^α is smooth wrt p_K then

$$\Pr(\text{success}) = L(-\alpha/2\beta). \qquad (32)$$

Note: While Dixon's algorithm only needs $\alpha = 1$ because the size of $x^2 \% n$ is comparable to the size of n when x is $U(1,n-1)$. COS include α as a parameter because other, better algorithms will produce numbers comparable to $n^{1/2}$, corresponding to α=0.5, or even smaller.

Proof: See COS paper. It follows directly from the fact that $\Psi(X,Y) \cong u^{-u}$ where $u = \ln(X)/\ln(Y)$.

Theorem: Factors of ln(n) to any power can be neglected in the expression for Ω, the effort to factor a number or compute a discrete log using Dixon's method or any other method which results in Ω being of the form L(k) for some finite k. Similarly, factors of p_K and K are interchangeable. All the other methods we shall study meet this condition.

Proof: Ω=L(k) is super-polynomial in ln(n). Neglecting a polynomial factor therefore does not change the asymptotic form. Similarly $p_K \cong K\ln(K)$ and the factor of ln(K) can be neglected without changing the asymptotic form. §

3.7 Application to Dixon's Algorithm

With this background, refer back to (17) which gives the effort for Dixon's algorithm:

$$\Omega \cong \{K\,\Pr^{-1}(\text{success})[K\ln(n)+\ln(n)]\} + K^3 \qquad (17)$$

Using the above simplifications, this reduces to

$$\Omega \cong K^2\Pr^{-1}(\text{success}) + K^3 \qquad (33)$$

so, letting $p_K = L(\beta)$,

$$\Omega = L(2\beta)L[1/(2\beta)] + L(3\beta) \qquad (34)$$

since α=1 for Dixon's method (the residues are O(n)). This reduces to

$$\Omega = L[2\beta+1/(2\beta)] + L(3\beta). \qquad (35)$$

Optimizing over p_K is now the same as optimizing over β and is easy: Set the derivative of $[2\beta+1/(2\beta)]$ to zero to find $\beta^*=1/2$ and $\Omega = L(2) + L(1.5) = L(2) = \exp\{2[\ln(n)\ln\ln(n)]^{1/2}\}$. We also see that the K^3 term which corresponds to Gaussian elimination is negligible.

3.8 Continued Fraction Method

Several advances in factoring algorithms have focussed on improving Pr(success) by using carefully chosen, rather than random values, for the x_i's which are squared mod-n in equation (14). Continued fractions (CF) approximate irrational numbers very rapidly. For example, 22/7 = 3.143 is

the first CF approximation to $\pi = 3.142$, with both rounded to three decimal places.

See Knuth, vol. 2, 2nd edition, section 4.5.3 for a detailed exposition of CF's and pages 380-384 for their application to factoring via the Morrison-Brillhart CF factoring algorithm. Here, I present the main steps without proof.

- The numerator x_i and denominator d_i of the ith CF expansion for $\sqrt{n}$ can be easily computed modulo-n.
- Because $(x_i/d_i)^2 \approx n$, it is not surprising that $x_i^2\%n$ is small. It can be shown to be $O(\sqrt{n})$, instead of $O(n)$ for the $x_i^2\%n$ of Dixon's method.
- The analysis proceeds almost exactly as for Dixon's algorithm except that u, the ratio of $\ln(x_i^2\%n)$ to $\ln(p_K)$, is now half as big for the same p_K.
- Because Pr(success) $\cong u^{-u}$, we can replace n by $n^{1/2}$ in the analysis.
- Because $\ln(n)\ln\ln(n) \cong 0.5\ln(n)\ln\ln(n)$, the optimal k is reduced from 4 to 2.

As explained in Knuth, it is sometimes better to use the CF expansion for $\sqrt{kn}$ with k being a small integer, instead of the CF expansion for $\sqrt{n}$.

The CF method has almost exactly the same form for Ω as Dixon's method. The only difference is that α=1/2 because the residues are $O(n^{1/2})$. Hence (17) becomes

$$\Omega = L(2\beta)L[1/(4\beta)] + L(3\beta)$$

$$= L[2\beta+1/(4\beta)] + L(3\beta) \quad (36)$$

The first term is minimized at $\beta^*=1/8^{1/2}$ and yields $\Omega = L(2^{1/2})+L[(9/8)^{1/2}] = L(2^{1/2})$.

4.0 Discrete Logarithms

There is a close connection between factoring and finding discrete logarithms. About fifteen years ago, when I was first starting out in this area of cryptography, there were no direct connections (i.e., no algorithm that was invented for factoring could be applied directly to discrete logs or *vice versa*), but the $\Psi(X,Y)$ function came up in both places. Since then, several algorithms have been transferred directly from one problem to the other. I give this history as evidence to support my hunch that the two problems are equivalent.

This section describes the algorithm of fifteen years ago that led to the above conjecture. It has the same running time as Dixon's except n, the number to be factored, is replaced by p, the prime number which is the modulus for the discrete logs. Hence $\Omega = \exp\sqrt{4(\log p)\log\log p}$. (Later we will improve from k=4 to k=1, the same as was done for factoring.)

The algorithm is variously credited to Western and Miller, to Merkle, and to Adleman for the following reasons: In the 1960's, Western and Miller were faced with the problem of compiling tables of indices (the mathematician's name for discrete logs) and primitive roots (also known as primitive elements). The problem is to fit this data into a book, instead of several encyclopedias.

Primitive roots are not hard to tabulate: If α is one primitive root mod-p, then all primitive roots can be expressed as α^k where k is any integer relatively prime to p–1. Hence, for each prime p, all Western and Miller had to do was list one primitive root (usually the smallest) and the factorization of p–1. The user could quickly compute all other primitive roots from this data.

The problem was not so easily solved for indices and, at first, it might appear that they would have to list a table of size p–1 for each prime p. Western and Miller recognized, however, that publishing the indices for a set of small primes would allow the computation of the indices for all numbers which were a product of those small prime (smooth wrt to those primes). If, for example, they published the indices for 2 and 3 to some base (any primitive element α will do), then you could also compute the index of $18 = 2\times3^2$ from the relation

$$\log(18) = \log(2) + 2\log(3) \ \% \ (p-1)$$

since arithmetic in the exponent is modulo p–1.

If the number y whose index is sought is not of this form, then compute $y' = y\alpha^z \% p$ for random values of z until y' is smooth with respect to the set of small primes, {2,3} in this simple case. Now you can compute $\log(y')$ and subtract z (modulo p–1) to get $\log(y)$. Western and Miller did not analyze the run-time of their algorithm and did not seem to know it was subexponential.

Merkle independently derived the same algorithm while working on his Ph.D. thesis under my supervision in ap-

proximately 1977. He did not analyze the run-time either. Worse yet, another of my students did analyze the run-time, but made an error which indicated that, asymptotically Merkle's algorithm (as we called it then) was inferior to others. I did not catch this error either.

Leonard Adleman, the A of RSA, also derived the algorithm and showed the run-time should be of the form $\exp\sqrt{k(\log n)\log\log n}$ [L. Adleman, "A subexponential algorithm for the discrete logarithm problem with applications to cryptography," *Proc. 20th IEEE Foundations of Computer Science Symposium*, pp. 55-60, 1979].

The algorithm works in two stages: first compute the logs of the small primes $\{\pi_k = \log(p_k)\}_{k=1}^K$ and then use these as described above to compute the log of an arbitrary element. To find the $\{\pi_k\}$:

- choose a sequence of random x_i's uniformly distributed between 1 and $p-1$
- compute $y_i = \alpha^x \% p$ and check if y_i is smooth with respect to the set of small primes $\{p_k\}_{k=1}^K$.

Each success produces an equation similar in form to (14)

$$\alpha^{x_i} \% p = \prod_{k=1}^{K} p_k^{e_{i,k}} \quad (37)$$

which gives an equation in terms of the unknown $\{\pi_k\}$:

$$x_i = \left(\sum_{k=1}^{K} e_{i,k}\pi_k\right) \% \; p-1 \quad (38)$$

Once we have K independent equations, we can solve for the $\{\pi_k\}$. It takes approximately K successes for this to happen. Hence this first part of the algorithm is exactly equivalent in effort to that required for Dixon's method of factoring a number n, provided n is the approximately the same size as p. The optimal value of u and other quantities predicted in section 3.5 are all directly applicable.

As indicated earlier, the second phase of the algorithm computes $y' = y\alpha^k \% p$ until y' is smooth, in which case $\log(y) = \log(y') - k \% (p-1)$. This part of the algorithm takes $1/K$ as much effort as the first part, and is thus negligible. For this reason, if a number of users share a common modulus p in a Diffie-Hellman key exchange system or an El-Gamal signature system, the cryptanalyst can do a large precomputation once and then break individual keys much more rapidly. This is the danger of a "shared modulus system" with the digital signature standard (DSS).

The algorithm can be illustrated by a small example. Take $p = 229$ and $\alpha = 6$ as the base of the logs. (6 is the smallest primitive element in GF(229).) Instead of generating random x_i's, I used $x_i = 100, 101, \ldots$ (These seemed as if they should work as well as truly random values.) I took $p_K = 7$, rather than optimizing over this parameter as in section 3.5 (see exercise below). Here is the resultant table, keeping only the successes:

x	100	105	106	107
$6^x \% p$	180	32	192	7
factored	$2^2 3^2 5$	2^5	$2^6 3$	7

We see that $\log(7) = 107$ and $\log(2) = 5^{-1}105 = 21$. Then

$$\log(3) = 106 - 6\log(2) = 208$$

$$\log(5) = 100 - 2(21+208) = 98$$

Exercise: Check the four discrete logs found above.

Exercise: I used $p_K = 7$. What is the optimal value predicted in section 3.5? Optimize numerically (precisely) over p_K. What is the optimal value now? How does the actual optimal effort compare to the predicted optimal effort Ω? This is a non-trivial, but doable exercise. If any one does it, please show it to me for possible inclusion in a later version of these notes.

4.1 Shank's Method for Discrete Logs

Shank's method has a running time of approximately $p^{1/2}$. Although much slower than the Western-Miller-Merkle-Adleman algorithm, it is a useful time-memory tradeoff whose main idea finds application in other areas, including a very clever factoring algorithm due to Pollard. The trick is to represent x as a high and low order part

$$x = x_1 M + x_0 \quad (39)$$

where M is an integer $\approx n^{1/2}$. Since $0 \le x \le p-1$

$$0 \le x_0 < M - 1 \approx n^{1/2} \quad (40)$$

$$0 \le x_1 \le [\,(p-1)/M\,] \approx n^{1/2} \quad (41)$$

From (34), $y = \alpha^x$ can be restated as

$$y\alpha^{-x_0} = \alpha^{x_1 M} \quad (42)$$

We therefore construct a table $\{\alpha^{x_1 M}\}_{x_1=0}^{(p-1)/M}$ and sort it. We then compute $y\alpha^{-x_0}$ for $x_0 = 0, 1, 2, \ldots$ until a match is found with a value in the table. (Sorting the table allows matching to proceed rapidly.)

A match must be found before $x_0 = M$, so the time and memory required are each approximately $n^{1/2}$. A little thought shows that a more general time-memory tradeoff is possible so long as the time-memory product $TM = n$ and $M \le T$. (If $M > T$ then the time required for computing the table dominates. Because memory is more expensive than time, it is hard to imagine such a situation.)

5.0 Schroeppel's Algorithm

5.1 Fermat Numbers

The great seventeenth century mathematician, Pierre de Fermat, noticed that 2^N+1 was prime for $N = 0, 1, 2, 4, 8$ and 16. He then conjectured that all numbers of the form $2^{2^\wedge n}+1$ were prime. These are now known as the Fermat numbers, and are indexed by n, so $F_0=3$, $F_1=5$, $F_2=17$, etc.

Although Fermat established the primality of $F_0=3$ through $F_4=65537$, none of the succeeding Fermat numbers that have been tested have been found to be prime. While Fermat's conjecture had to wait until the next century to be proved false by Euler, it was well within Fermat's mathematical and computational abilities to prove that F_5 is not prime. By Fermat's own theorem ($a^{p-1}\%p = 1$ for all $a \ne 0$ if p is prime), he could have shown that F_5 was not prime by computing $3^{(2^\wedge 32)}\%(2^{32}+1)$ and showing that the answer was not 1. This takes only 32 multiplications mod-n.

Interestingly, the test fails if 2 is used as the base, instead of 3. Fermat may have tried 2 as the obvious base and been misled. Numbers such that $b^{n-1}\%n=1$ are called pseudo-primes to the base b. Pseudo-primes to the base 2 are also called just pseudo-primes. Pseudo-primes are rare, with the first five being 341, 561, 645, 1105, 1387.

A group of mathematicians and computer scientists has an informal competition going to see who can factor the largest numbers and attention naturally focuses on numbers of special form, such as the Fermat numbers. $F_7=2^{128}+1$ held out until 1970, when Morrison and Brillhart used their CF method to show that its two factors are 59649589127497217 and 5704689200685129054721. Interest then gravitated to factoring $F_8=2^{256}+1$. Using $\exp[2\ln(n)\ln\ln(n)]^{1/2}$ as the time required by the CF method, approximately 4E18 operations would be needed to factor F_8, a computation that would take over 100,000 years at 1 µsec per operation. Schroeppel's algorithm cut the expected running time to $\exp[\ln(n)\ln\ln(n)]^{1/2}$ and requires only half a year at 1 µsec per operation. While this was an extremely useful improvement, for reasons explained in Section 10.0, Schroeppel lost the race to factor F_8 to a usually much weaker competitor.

5.2 Schroeppel's Algorithm for Factoring

Let m be the best integer approximation to $n^{1/2}$. For example, if $n = 197209$, then $n^{1/2} = 444.0822$ and $m = 444$. Because $(n^{1/2}+\varepsilon) - n = 2\varepsilon n^{1/2}+\varepsilon^2$ and the error in the integer approximation is less than 1/2, it follows that $m^2 - n < n^{1/2}+0.25$. Similarly, provided $|A| << n^{1/2}$, $|(m+A)^2 - n| = O(n^{1/2})$, giving us relations similar to that for the CF method, with the x_i replaced by $m+A$. Again using $n = 197209$ as an example, the following table results:

A	0	1	2	3	4
$(m+A)^2-n$	–73	816	1707	2600	3495
factored	$-1^1 73$	$2^4 3^1 17$	$3^1 569$	$2^3 5^2 13$	$3^1 5^1 233$

Note that –1 has been added as a new "prime" in the factorizations for the following reason: As A becomes larger, the values of $(m+A)^2-n$ also become larger and are less likely to be smooth. Instead of using $A = 0, 1, 2, \ldots A_{max}$, it makes sense to use positive and negative values which are bounded in magnitude by $A_{max}/2$. Continuing the above table in this manner we find:

A	–1	–2	–3	–4
$(m+A)^2-n$	–960	–1845	–2728	–3609
factored	$-1^1 2^6 3^1 5$	$-1^1 3^2 5^1 41$	$-1^1 2^3 11^1 31$	$-1^1 3^2 401$

As anticipated, we get more smooth results this way. All that is required is to add –1 to the factor base and require the final result to have an even power of –1 as well even powers of the small primes.

Schroeppel went one step further and suggested using residues of the form $(m+A)(m+B)-n$ and including the $(m+A)$ or $(m+B)$ terms as new "primes," just as we added –1. This allows $A_{\max}$ to be reduced to approximately the square root of its old value, further reducing the size of the residues and increasing the chance of their being smooth.

As a small example, again take n = 197209 and $m = 444$, and let *A and B* range from –2 to +2, so that the matrix of residues $(m+A)(m+B)-n$ is:

```
       B=-2   B=-1   B= 0   B=+1   B=+2
A=-2  -1845  -1403   -961   -519    -77
A=-1          -960   -517    -74    369
A= 0                  -73    371    815
A=+1                         816   1261
A=+2                               1707
```

The first row (A=–2), first column (B=–2) entry is

$$(m+A)(m+B)-n = (444-2)(444-2) - 197209 = -1845.$$

Check a few of the other values to make sure you understand what they represent. Now divide the above matrix of residues by the first seven primes {2, 3, 5, 7, 11, 13, 17} as shown below. If the residue is not smooth wrt 17 [e.g., $-1845 = -(2^0 3^2\ 5^1 41)$], seven stars are printed. If the residue is smooth, the vector of seven exponents is printed (e.g., $-77=-(2^0 3^0\ 5^0 7^1 11^1 13^0 17^{01})$]. In either case, a plus or minus sign precedes the entry to indicate whether the residue is positive or negative. (If the residue is smooth, a minus sign means the "prime" –1 occurs to the first power.)

```
-******* -******* -******* -******* -0001100
         -6110000 -******* -******* +*******
                  -******* +******* +*******
                           +4100001 +*******
                                    +*******
```

$A_{\max}$=2 is too small to allow factoring n, but the approach can be outlined. The (2,2) entry above corresponds to

$$(m-1)^2 = -1^1 2^6 3^1 5^1\ \%n \quad \textbf{(43)}$$

while the (1,5) entry corresponds to

$$(m-2)^1(m+2)^1 = -1^1 7^1 11^1\ \%n \quad \textbf{(44)}$$

In (38), the "primes" $m-1$ and 2 are raised to even powers while –1, 3 and 5 have odd powers, so we would seek other smooth residues which also have –1, 3 and 5 to odd powers and multiply to obtain all even exponents. In (39), the primes $(m-2)$, $(m+2)$, (–1), 7, and 11 all have odd powers, and require cancellation by other smooth residues which also have these primes to odd powers.

There are $K+2+2A_{\max}$ primes, including –1 and the $(m+j)$'s, so we need approximately that many successes (smooth residues) before we can obtain an equation of the form

$$\prod_{j=-A_{max}}^{A_{max}} (m+j)^{E_j} = \prod_{k=0}^{K} p_k^{e_k} \quad \textbf{(45)}$$

with both vectors of exponents $\boldsymbol{E}$ and $\boldsymbol{e}$ having only even components. Once this happens, we have $X^2=Y^2$ %n with

$$X = \prod_{j=-A_{max}}^{A_{max}} (m+j)^{E_j/2} \quad \textbf{(46)}$$

$$Y = \prod_{k=0}^{K} p_k^{e_k/2} \quad \textbf{(47)}$$

so that GCD($X-Y$, n) = p or q with probability 50 percent.

5.3 Sieving

Thus far, Schroeppel's algorithm has no advantage over the CF method. Both give residues that are $O(n^{1/2})$ instead of $O(n)$. But, for the same value of K, Schroeppel's algorithm has $K+2+2A_{\max}$ primes, while CF has only K. The advantage of Schroeppel's method only comes into play when sieving is used. This is similar to the famous "sieve of Eratosthenes" for finding all primes in the set $[1,N]$. Applied to Schroeppel's algorithm, sieving works by noting that, if p_k divides $(m+A)(m+B)-n$, then it also divides

$$(m+A+p_k)(m+B) - n = [(m+A)(m+B) - n] + p_k(m+B)$$

It is therefore unnecessary to try dividing all residues by each p_k. Once we find one residue which is divisible by p_k and which corresponds to (A,B), we know that all residues

corresponding to $(A+ip_k,B+jp_k)$ also are divisible by p_k. When $p_k \approx 100$, this saves 99 percent of the effort. Since trial division by the small primes was the most time consuming part of the algorithm (Dixon's, CF, or Schroeppel's), this is a major advance. There are efficient algorithms to locate the first residue divisible by each prime p_k, but that is a fine point we shall skip.

Sieving reduces the effort to obtain K equations to

$$\Omega = K\,\mathrm{Pr}^{-1}(\text{success}) \sum_{k=1}^{K} 1/p_k \tag{48}$$

The sum can be approximated by an integral which evaluates to less than O(ln(K)) and can thus be neglected. Since α is still 1/2,

$$\Omega = L(\beta)L[1/(4\beta)] = L[\beta+1/(4\beta)] \tag{49}$$

Optimizing, we find $\beta^* = 1/2$ and $\Omega = L(1)$. We need to check that the time to solve the K equations does not dominate the effort. If we use Gaussian elimination, $K^3=L(3\beta^*)=L(1.5)$ and does dominate. Using sparse matrix techniques, the time drops to $K^2=L(2\beta^*)=L(1)$, the same as to obtain the equations. While solving the equations is asymptotically as difficult as solving them, according to Schroeppel, this is not true for problems of current interest where n is 100 to 1000 bits long.

Since $\Omega = \exp\sqrt{k\,(\log n)\log\log n}$, sieving effectively cuts the length of n in half by halving k. Consequently, if an algorithm without sieving can factor a 128-bit number, if sieving can be used with the algorithm, it can factor approximately 256-bit numbers. As shown above, Schroeppel's algorithm allows sieving. No one has found a way to apply sieving to the CF method. Hence Schroeppel's attempt to factor F_8 when CF could factor F_7, a number half as large.

5.4 Extension to Discrete Logarithms

The COS paper already referenced (Coppersmith, Odlyzko and Schroeppel) extended Schroeppel's factoring algorithm to compute discrete logarithms.[1] In addition to finding the discrete logarithms $\{\pi_k\}_{k=1}^{K}$ of the conventional primes $\{2,3,\ldots p_K\}$, they also find $\pi_0=\log(-1)$ and $\{\upsilon_j=\log(-m+j)\}$ where j ranges from $-A_{\max}$ to $+A_{\max}$. For example, equations (38) and (39) yield the equations:

$$2\upsilon_{-1} = \pi_0 + 6\pi_1 + \pi_2 + \pi_3 \ \%\ 197208 \tag{50}$$

$$\upsilon_{-2} + \upsilon_2 = \pi_0 + \pi_4 + \pi_5 \ \%\ 197208 \tag{51}$$

Once we have slightly more than $K+2+2A_{\max}$ successes, we are able to solve for the logs of all the "primes" involved: $\{\pi_k\}_{k=0}^{K}$ and $\{\upsilon_j\}$. The next part is tricky because the technique of section 4.0 produces y' values which are O(n) instead of O($n^{1/2}$), which is what we need if we are to get k=1 in $\exp\sqrt{k\,(\log n)\log\log n}$ as the effort required. COS explain how to handle this problem in detail in section 6 of their paper. I will only summarize the key idea.

As in section 4.0, generate $y' = y\alpha^k \% p$ until y' is "relatively smooth," that is smooth wrt a larger set of primes $\{p_k\}_{k=1}^{M}$. Since the set $\{p_k\}_{k=1}^{K}$ is the set of small primes, the new set will be called the set of small and medium primes. COS use

$$p_M = \exp\left(\sqrt{4\,(\log p)\log\log p}\right) \tag{52}$$

Lenstra's elliptic curve method of factoring is used to detect when y' is relatively smooth because its running time to do this is $\exp[2\ln(p_M)\ln\ln(p_M)]^{1/2}$. While the runtime of Lenstra's algorithm is the same as Schroeppel's (or the soon-to-be -discussed quadratic sieve) when n, the number to be factored, is the product of two primes, each of approximately size $n^{1/2}$, it is much faster when the prime factors are more unevenly distributed. In particular, here we are looking for y' such that it has a number of prime factors all less than p_M, which is much less than $p^{1/2}$. Hence, Lenstra's algorithm is only run long enough to factor y' if it is relatively smooth. If it has not factored y' by that time, its factorization is of no interest and may be abandoned.

Thus, after a reasonable time, COS obtain a y' which is the product of small and medium size primes. We already know the discrete logs of the small primes, but need to find those of the medium primes involved. To do this they invoke a clever trick which allows the discrete log of any

1. Here's what *really* happened. Coppersmith and Odlyzko figured out how to extend Schroeppel's factoring algorithm to discrete logs. They called Schroeppel, and he said "Doesn't everyone already know that?" [Answer: No. I certainly didn't! Nor presumably did Adleman, who extended Dixon's algorithm to logs, but did not mention the possibility of extending Schroeppel's, with its much better performance.] Coppersmith and Odlyzko then added Schroeppel as a coauthor. Even though Schroeppel seems to have an aversion to publishing, this is how he came to "publish" a paper.

medium size number (including each of the medium sized primes involved in the factorization of y') to be found rapidly enough that this time does not change the asymptotic form of the run-time. This trick is clever, but complicated, so I refer you to the original paper for details.

6.0 Quadratic Sieve

The quadratic sieve (QS) factors a composite number n in $\exp\sqrt{(\log n)\log\log n}$ steps, the same as Schroeppel's algorithm. Schroeppel's came first and the father of the QS, Prof. Carl Pomerance, gives credit where it is due by noting: "The QS can also be viewed as a natural outgrowth of Schroeppel's algorithm." Yet, much of the literature I have seen overlooks this parentage.

We have already seen the QS in the very beginning of section 5.2. I just didn't tell you that's what it was. It is the algorithm which uses $(m+A)^2-n$ as the residues tested for smoothness. We already showed that these residues are $O(n^{1/2})$ instead of $O(n)$, so we only need to show that sieving is possible to establish the claimed run-time. Because

$$(m+A+p_k)^2-n = [(m+A)^2-n] + 2p_k(m+A) \quad \textbf{(53)}$$

if $(m+A)^2-n$ is divisible by p_k, then so is $(m+A+p_k)^2-n$. That is the essence of the proof!

For evidence of this truth, examine the tables at the bottom of page 11 and the top of page 12. Note that 2 appears as a factor in the residues corresponding to $A = -3, -1, 1$, and 3. Equation (46) shows that *all* odd values of A will have 2 as a factor if one does. And it goes further. Because $2^4=16$ is a factor when $A = 1$, it will also be a factor whenever A is of the form $16i+1$. Similarly, because 2^3 is a factor when $A = \pm 3$, it will also be a factor for all A of the form $8i\pm 3$.

This last point demonstrates that a prime or a power of a prime ($8=2^3$ in the above example) can divide more than one sequence of *A's* and lead to a practical improvement (as opposed to changing the asymptotic form). This is because most smooth numbers are divisible by higher powers of the first few primes. For example, the first three numbers greater than 9000 that are smooth wrt 50, are $9009=3^2 7^1 11^1 13$, $9016=2^3 7^2 23$, and $9020=2^2 5^1 11^1 41$. But, if 2 and 3 are restricted to occur to at most the first power, then $9025 = 5^2 41$ is the first to be smooth wrt 50, and even it is divisible by 5^2.

If no sequences of A's are divisible by 2^3, then n is likely to be harder to factor than if (as for 197209) two sequences of such A's exist. Three sequences divisible by 2^3 would be even better. Hence, some numbers will be easier to factor than others. This might not seem to help when we are given a particular n to factor, until it is realized that we can use the QS to factor kn where k is a small multiplier. Because the QS does not favor small factors, it is as likely to find p or q as the small multiplier. While a few extra quadratic congruences are needed, this does not change the run-time even by 1 percent for the range of numbers of usual interest. So, before running the QS, a search is usually done over small values of k until a value kn is found with a larger than average number of sequences divisible by higher powers of the very small primes. (I will not go into how to do that, but there are fast methods.)

7.0 Elliptic Curves

In 1985, H. W. Lenstra, Jr. introduced a new factoring method based on elliptic curves (EC). Its running time to find a factor p is $\exp[2\ln(p)\ln\ln(p)]^{1/2}$. If $n = pq$ with p and q both near $n^{1/2}$, then this is $L(1) = \exp[\ln(n)\ln\ln(n)]^{1/2}$, the same as for Schroeppel's algorithm or QS. Random composite numbers usually break up more unevenly, for example $p\approx n^{1/3}$ and $q\approx n^{2/3}$, in which case the running time would be cut to $\exp[(2/3)\ln(n)\ln\ln(n)]^{1/2} = L[(2/3)^{1/2}]$.

For this reason, an RSA key generator would avoid unequal size factors and pick $\ln(p)\approx\ln(q)$ (but not too close because of Fermat's method which searches over $p-q$). While this ability to choose n at will in an RSA system makes the EC method of little use there, as noted in section 5.4, it is useful in finding discrete logarithms (in the part where we had to find if y' was relatively smooth) and in many other problems.

I will provide only a brief outline of how and why the EC method works. If you want to learn more, Stephens' paper in the 1985 Crypto proceedings (pp. 409-416) provides a starting point.

The key idea here, as with many other algorithms, is that computations mod-n preserve the answer mod-p, provided that p is a factor of n. For example, if n=15 and p=5, then $3^4 \%5 = (3^4 \%15) \%5$ because, whenever we subtract or add a multiple of 15 in mod-15 arithmetic, we also subtract or add three times that multiple of 5. Hence the answer mod-5 is not affected. This is helpful because, in factoring n, we know n but not p.

This idea was used in section 3.3 and all succeeding sec-

tions which make use of the value of finding $x^2=y^2 \% n$. For then $x^2=y^2 \% p \Rightarrow x=\pm y \% p \Rightarrow$ either $(x+y)$ or $(x-y)$ is a multiple of p and GCD($x\pm y$, n) is likely to be p.

Before describing Lenstra's EC algorithm, it is necessary to consider an earlier factoring algorithm due to Pollard, known as Pollard's p–1 method (to distinguish it from Pollard's ρ or Monte Carlo method, which we will look at briefly in seciton 9.0).

7.1 Pollard's *p*–1 Method

This method of factoring is extremely rapid if p–1 has only small prime factors. RSA key generators often avoid such values of p and q for this reason, but the method is still of interest for naturally occurring numbers, where there is a chance p will be of this form. It is also of interest because of its connection to Lenstra's EC factoring algorithm.

Theorem: If $n=pq$ and p–1 is smooth wrt p_K while q–1 is not, then GCD(a^E–1,n) = p for most a provided

$$E = \prod_{k=1}^{K} p_k^{c_k} \tag{54}$$

$$c_k = \lfloor (\log n) / (\log p_k) \rfloor . \tag{55}$$

Proof: As with many other important results, the foundation lies in Fermat's Theorem: $a^{p-1}=1\% p$ so that $(a^{k(p-1)}-1)$ is a multiple of p. Because E is guaranteed to be a multiple of p–1 but not of q–1, a^E–1 will be a multiple of p but probably not of q. I say "probably" because a might be of low order in GF(q), but that has low probability.§

The proof is so simple that a small example helps to make it believable: You are told that n = 402,257 is the product of two primes, p and q, and that p–1 is smooth wrt 5. Take

$$E = 2^{18}3^{11}5^{8} = (262{,}144) \times (177{,}147) \times (390{,}625)$$
$$= 18{,}139{,}852{,}800{,}000{,}000.$$

E is a multiple of p–1 because 2^{19}, 3^{12} and 5^9 are each greater than n. Note that here, where three primes are used, E is approximately n^3. In general, when p_K is the upper bound on the factors of p–1, E will be approximately n^K.

Using a=2, we find $2^E\% n$ = 397,062 and Euclid's algorithm rapidly finds GCD(397062–1, 402257) = 433. The other factor of n is then found to be 929.

Now we can see why the method works: $p-1 = 432 = 2^4 3^3$ divides $E = 2^{18}3^{11}5^8$ while $q-1 = 928 = 2^5 29$ does not. Hence $2^E\% p$ =1 but $2^E\% q$ does not, so p divides 2^E–1, but q does not. While we do not know p and therefore cannot work in mod-p arithmetic, mod-n is just as good since it preserves arithmetic mod-p. We also rely on the fact that $2^E\% n$

- can be computed rapidly, in at most $2\log_2(E)$ multiplications mod-n
- is a reasonable size. (In normal arithmetic 2^E would have been 2,000 terrabytes long for this small example.)

7.2 Analysis of Pollard's *p*–1 Method

While Pollard's p–1 method only factors a small subset of integers, we can assign it a figure of merit based on the expected effort required to factor an integer near n. For application to the RSA system, we assume the number to be factored is composed of two approximately equal-sized, randomly chosen prime factors. We further assume that the probability that $p-1$ is smooth wrt p_K is the same as the probability that a random number of the same size is smooth. There is ample experimental, but little theoretical, justification for this latter assumption.

Since p_K is a parameter that can be chosen by the cryptanalyst to minimize his expected effort, we optimize over it, just as in section 3.5. Letting Ω denote the expected effort,

$$\Omega \cong [\ln(E)\ \ln(n) + \ln^2(n)]\mathrm{Pr}^{-1}(\mathrm{success})$$

because

- $\ln(E)$ is the number of multiplications mod-n required to compute $2^E\% n$.
- $\ln(n)$ is the effort per multiplication mod-n. (Really $\ln(n)\ln\ln(n)$ due to the FFT, but even $\ln(n)$ will be negligible.)
- $\ln^2(n)$ is the (negligible) time to run Euclid's algorithm.
- Pr^{-1}(success) is the expected number of trials before expecting to achieve one success (factor a number near n).

Since $E \cong n^K$

$$\Omega \cong K\ \ln^2(n)\ \mathrm{Pr}^{-1}(\mathrm{success}) \tag{56}$$

Using the approach and terminology of section 3.6, and again letting $p_K \cong K = \mathrm{L}(\beta)$,

$$\Omega = L(\beta)\, L[1/(4\beta)] \quad (57)$$

α=1/2 because the number which has to be smooth is $(p-1) \approx n^{1/2}$. Equation (57) is identical to (53), so after optimization, the expected effort is L(1) or

$$\Omega \cong \exp \sqrt{(\log n) \log\log n} \quad (58)$$

the same as for Schroeppel's algorithm or QS.

While Pollard's method might at first appear to be comparable with Schroeppel or QS, it has a major difference. Here Ω is only an *expected* value. We do very little effort trying to factor each number n, and have a very small probability of factoring it. If we need to factor a specific n, this is inferior to Schroeppel or QS where n is factored with probability close to 1. (Removing that inferiority is the genius behind Lenstra's elliptic curve (EC) method, discussed next.) This disadvantage of Pollard's method is also an advantage if viewed from another angle. Suppose Pr(success) = 1E-3 and Ω = 1E3 years on the computer available to us. If run for one year, Pollard's method has a 10^{-3} chance of success, while Schroeppel or QS has an infinitesmal chance of success. (The probability of finding a dependence relation in K/1000 random binary vectors of length K is almost zero. When K = 1,000,000, this probability is upper bounded by $2^{-999,000} = 10^{-300,729}$, much smaller than the 10^{-3} Pr(success) for Pollard's method.)

7.3 Extension to Elliptic Curves

In Pollard's p–1 method, $x_i = \alpha x_{i-1}$ defines a useful recurrence relation because

- the period is either p–1 or a submultiple thereof.
- x_i can be computed in time proportional to a polynomial in ln(i). Hence we can take E to be the huge number specified in (54) and (55).

Elliptic curves mod-p are a recurrence relation on pairs of integers $\mathbf{x} = (x_1,x_2)$ with the following properties:

- for a fixed p, there are many elliptic curves, corresponding to the choice of two parameters a and b in the recurrence relation.
- if a and b are chosen randomly, the period is well modeled as a $U(p-2p^{1/2}, p+2p^{1/2})$ random variable.
- the ith point on an elliptic curve mod-p can be computed in time proportional to a polynomial in ln(i).

Elliptic curves have all the properties which made Pollard's p–1 method have Ω = L(1) as its expected effort so, by varying a and b with n fixed, elliptic curves give us many chances to factor a fixed $n = pq$ with effort Ω = L(1). Instead of having just one chance to factor n, and having to average over many values of n, elliptic curves give us many independent chances to factor n, each with the same effort and probability of success as Pollard's p–1 method.

8.0 Number Field Sieves

Number field sieves (NFS) are a new method of factoring based on algebraic number fields. This method, developed in 1990 by Lenstra, Lenstra, Manasse and Pollard, has the distinction of being the first to break the L(k) bound. Its effort to factor a number n is of the form

$$\Omega = \exp\{c\, [\ln(n)]^{1/3} [\ln\ln(n)]^{2/3}\} \quad (59)$$

The NFS was first applied to factoring the ninth Fermat number $F_9 = 2^{512} + 1$. For numbers of this form ($r^s + e$ with r and e not too large), the NFS is much faster than in general. The "special" NFS has c = 1.526, while the general NFS has c = 1.902.

Because of the larger value of c, NFS does not provide any savings over QS until $n \approx 2^{512}$. Factoring a 256-bit number with NFS will actually be *slower* than with QS. But, just as factoring went from $c = 2^{1/2}$ for CF to c=1 for QS, prudence would dictate assuming that improvements to NFS will reduce c.

While the progress over the last five years has been slow, cutting c for the general NFS from its intial value of 2.080 to 1.902 today, if c can be reduced to 1, an RSA system would need at least a 4000 bit and preferably an 8000 bit key size as seen from the table below. These developments emphasize a point I have made repeatedly: Safety margins of at least a factor of 2, and preferably 4 or greater, are prudent when setting key size. While also true for conventional systems, such as DES, this is particularly for public key systems because their greater mathematical structure is likely to admit greater mathematical insight.

Method	2^{512}	2^{1024}	2^{2048}	2^{4096}	2^{8192}
QS	7E19	4E29	1E44	2E65	2E96
NFS c=2	1E20	1E27	4E36	1E48	2E65
NFS c=1	1E10	4E13	2E18	3E24	4E32

I will distribute other notes and papers on the NFS.

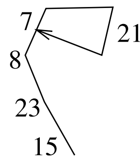
9.0 Pollard's ρ Method

Pollard's ρ method grows out of the "birthday problem," which states that, if approximately $p^{1/2}$ random values are chosen mod-p, then an overlap becomes highly likely. (This is more evidence in favor of my statement earlier in the quarter that a significant fraction of the important results in cryptography stem from the birthday problem!) This method is sometimes also called "Monte-Carlo" factoring because of the randomness involved.

The ρ appellation applies because a many-to-one recurrence relation on a finite set must eventually cycle and is therefore be shaped like a ρ. For example, the mapping

$$x_{i+1} = x_i^2 + 1 \ \% \ p \qquad (60)$$

with p=29 and x_0=15 produces the sequence 15, 23, 8, 7, 21, 7, 21, 7, etc. as depicted in the following picture:



Normally there will be many more points, and the ρ can be made much smoother.

While there is no proof that (60) can be modeled as if it were random, experiments show that is a good model. In particular, the average time before (60) cycles is approximately $p^{1/2}$ for most values of p. There is nothing special about x^2+1%p and any other pseudo-random, many-to-one mapping mod-p would also work.

Of course, in factoring n we do not know p, so the recurrence must be computed mod-n. But that preserves the result mod-p. For example, if n=28913=29×997, and x_0=15 then the x_i sequence will start: 15, 226, 22164, 11027, 15565, 7199, 13506, 28833, 6401, 3081, 9098. While this has not yet cycled, it has cycled modulo the unknown factor 29, where the sequence is as before: 15, 23, 8, 7, 21, 7, 21, 7, 21, 7, 21. Because the other factor of n is so much larger, we do not expect that the sequence has started to cycle modulo-997, and it has not as shown in Table 2 below.

Because the period of the recurrence relation mod-29 is two, $x_{i+2} - x_i$ will be a multiple of 29 once we are in the loop, and GCD($x_{i+2} - x_i$, n) = 29. For example, 9098–6401 = 2697 = 29×93, so GCD(2697,28913) = 29.

In this example we knew the period mod-p was two, but in real life we will only know that the period is O($p^{1/2}$). Therefore, if we suspect that n has a factor p which is less than P, we can compute x_m for $m = 5P^{1/2}$, or any other value which is not too large but which is likely to be out of the "tail" and into the "loop" of the ρ in mod-p arithmetic.

Table 2. Pollard's ρ Method: n=28913=29×997

x%28913	x%29	x%997
226	23	226
22164	8	230
11027	7	60
15565	21	610
719	7	220
13506	21	545
28833	7	917
6401	21	419
3081	7	90
9098	21	125

Then compute

$$X = \prod_{i=1}^{5\sqrt{P}} (x_{m+i} - x_m) \quad \% \ n \qquad (61)$$

Again the factor of 5 is typical, not a hard and fast rule. If we are right that the tail and loop are each less than $5P^{1/2}$ in mod-p arithmetic, then at least one of the terms (x_{m+i}–x_m) will equal zero mod-p and X will be divisible by p. Hence GCD(X, n) will equal p. It is seen that the effort required is O($p^{1/2}$) so that Pollard's ρ method typically finds the smallest prime divisor of n first. Because $p^{1/2}$ grows so rapidly, this method is only useful for finding moderate-sized prime divisors of n. However, it extends our ability to find small prime divisors compared to trial division, whose complexity grows proportionately to p.

As evidence of the value of Pollard's ρ method, it is worth noting that Schroeppel developed his algorithm in an effort to factor $F_8 = 2^{256}+1$, but was beaten to the punch by Brent using Pollard's ρ method! This was somewhat surprising because, in the worst case, the ρ method requires an exponential number of operations, O($n^{1/4}$), to factor n. The only reason it worked on F_8 is that F_8 is a 78 digit number

which breaks very unevenly into a 16 digit factor (1238926361552897) and a 62 digit factor (93461639715357977769163558199606896584051237541638188580280321). Hence the effort of Pollard's ρ method was only $O(10^8)$.

Whenever it is suspected or hoped that *n* splits into such unequally sized factors, Lenstra's EC method described in section 7.3 should be used instead of Pollard's ρ method. (The EC method had not yet been developed when Brent tried to factor F_8.) If used on a carefully chosen RSA *n* of 78 digits with two roughly equal length factors, Pollard's ρ method would require $O(10^{39})$ operations and be clearly infeasible, while Lenstra's EC method requires only $O(10^{13})$ operations.

As evidence of the efficacy of the EC method, I factored F_8 in only two hours on my NeXT workstation (25 MHz 68040 processor) using Mathematica. Pretending I did not know the answer, I ran the program for one hour with an estimate of 10^{10} for the smaller factor. When that failed, I increased the estimate to 10^{20}, ran for another hour, and achieved success. See Knuth, volume 2, 2nd edition, pages 369-371 for more details on Pollard's ρ method.

10.0 Floyd's Cycle Finder

Floyd's cycle finding algorithm is extremely useful in cryptography, including in factoring algorithms that ned to find a repeat in a finite sequence. (Floyd is Prof. Robert Floyd of our own CS department.)

Given a recurrence relation x_i=f(x_{i-1}) on a finite set, it is clear that the *x* values must eventually repeat. Often the problem is stated in terms of halting a finite-state, pseudo-random number generator before it starts to cycle. The problem solved so elegantly by Floyd is to detect when cycling first happens, using only a few words of memory and just several times the computation needed just to reach the point of cycling.

Floyd showed that if we define the auxiliary sequence y_i=f[f(y_{i-1})] and stop the process at the first *N* for which $x_N=y_N$ then it is guaranteed that the $\{x_i\}$ sequence has not yet repeated. Conversely, Floyd's algorithm finds two equal *x* values, x_N and x_{2N}, with at most three times the number of iterations required to start cycling and with only two words of memory. See Knuth, vol 2, 2nd edition, page 4, for the proof that it really works.