

THE CONTINUED FRACTION FACTORING ALGORITHM

Dustin Moody

In 1931, D.H. Lehmer and R.E. Powers came up with an algorithm to factor large integers which relies on continued fractions. However, the method was difficult to implement with only paper and pencil. In 1975, M. Morrison and J. Brillhart published a paper detailing how the continued fraction algorithm could be efficiently programmed onto a computer. The algorithm has since become known as CFRAC. Using the most modern technology available, they were able to factor the seventh Fermat number with CFRAC:

$$2^{2^7} + 1 = 59649589127497217 * 5704689200685129054721$$

Until the discovery of the quadratic sieve, CFRAC was generally regarded as the fastest factoring algorithm. For my project, I implemented the basic CFRAC algorithm in SAGE.

One of the common approaches to factoring a large number N is the following. Suppose we can find two integers x and y such that $x^2 \equiv y^2 \pmod{N}$. Then $(x - y)(x + y) \equiv 0 \pmod{N}$, and we hope $\gcd(x-y, N)$ yields a nontrivial factor of N . For example, let $N=1147$. It is easy to see that $34^2 \equiv 1156 \equiv 9 \equiv 3^2 \pmod{1147}$. We find $\gcd(34-3, 1147) = 31$ and we have found a factor of N . The challenge for large integers is to find two squares congruent to each other mod N . The idea behind CFRAC is to use the continued fraction expansion of \sqrt{N} to find such a pair of squares. We now give a summary of how the algorithm works.

We first expand \sqrt{N} (or \sqrt{kN} for some suitably chosen $k \geq 1$) into a simple continued fraction $\sqrt{kN} = [q_0, q_1, q_2, \dots, q_{n-1}, \frac{\sqrt{kN} + P_n}{Q_n}]$. If $\frac{A_n}{B_n}$ is the N th convergent, then it is a well-known identity that

$$A_{n-1}^2 - kNB_{n-1}^2 = (-1)^n Q_n$$

and so $A_{n-1}^2 \equiv (-1)^n Q_n \pmod{N}$. We then look for a subset Q_i of $\{Q_1, Q_2, \dots, Q_n\}$ such that $\prod_i (-1)^i Q_i$ is a square. If this is not possible, we expand \sqrt{kN} further. This yields the congruence

$$A^2 \equiv \prod_i A_{i-1}^2 \equiv \prod_i (-1)^i Q_i \equiv Q^2 \pmod{N}.$$

We then compute $D = \gcd(A-Q, N)$, and if $1 < D < N$, we have found a nontrivial factor of N . For complete details, see Morrison and Brillhart's original paper.

As an example, let's factor $N = 13290059$, with $k=1$. Using recurrence relations, it is easy to compute $Q_1 = 4034, Q_2 = 3257, Q_3 = 1555, \dots$ where all computations are done mod N . Continuing the expansion, we eventually find that $Q_5 = 2 * 5^2 * 41, Q_{22} = 41 * 113, Q_{23} = 2 * 113$ hence

$$Q_5 Q_{22} Q_{23} \equiv 6769401 \equiv 46330^2 \pmod{N}$$

Computing the corresponding A_{i-1} , we find that


```
#####
#
# SAGE: System for Algebra and Geometry Experimentation
#
# Copyright (C) 2009 Dustin Moody <dbm25@math.washington.edu>
#
# Distributed under the terms of the GNU General Public License (GPL)
#
# This code is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# General Public License for more details.
#
# The full text of the GPL is available at:
#
# http://www.gnu.org/licenses/
#####
```

```
def cfactor(N,k=1,L=80,B=65):
    """
```

```
        Continued fraction factoring algorithm, as described in M. A. Morrison and J. Brillhart, A method of
        factoring and the factorization of F_7, Math. Comp. 29, 1975. (183-205)
        We first expand vN, or vkN for some suitably chosen k=1, into a simple continued fraction
vkN=[q_0,q_1,q_2,...,q_(n-1),v(kN+P_n)/Q_n ].
        If A_n/B_n is the Nth convergent, then it is a well-known identity that A_(n-1)^2-kNB_(n-1)^2=(-1)^n
Q_n
        and so A_(n-1)^2=(-1)^n Q_n mod N. We then look for a subset Q_i of {Q_1,Q_2,...,Q_n} such that
?_i!(-1)^i Q_i ? is a square. If this is not possible, we expand vkN further.
        This yields the congruence A^2=?_i!A_(i-1)^2=?_i!(-1)^i Q_i=Q^2 ?? mod N.
        We then compute D = gcd(A-Q,N), and if 1<D<N, we have found a nontrivial factor of N
        Note--this program is not very fast.
```

```
        The parameter L is the bound for how far we compute the continued fraction expansion of
vkN=[q_0,q_1,q_2,...,q_(L-1),v(kN+P_n)/Q_L].
        The parameter B is an upper bound for the primes in the factor base.
```

```
Example:
sage: p1=next_prime(10^6+500)
sage: p2=next_prime(10^5+500)
sage: cfactor(p1*p2,B=1000,L=1600)
100501
```

```
    """
```

```
    g=floor(sqrt(k*N))
    global setFB
```

```
    # Initialize variables in continued fraction expansion
    A=[]
    Q=[]
    r=[]
    P=[]
    q=[]
```

```

A.append(0)
A.append(1)
Q.append(0)
Q.append(k*N)
r.append(0)
r.append(g)
P.append(0)
P.append(0)
P.append(0)
Q.append(1)
q.append(0)
q.append(0)

#Find continued fraction of sqrt(k*N)
for i in range(0,L+1):
    q.append(floor((g+P[i+2])/Q[i+2]))
    r.append(g+P[i+2]-q[i+2]*Q[i+2])
    A.append((q[i+2]*A[i+1]+A[i]) % N)
    P.append(g-r[i+2])
    Q.append(Q[i+1]+q[i+2]*(r[i+2]-r[i+1]))
    # Check if expansion is periodic too quickly.
    if Q.count(Q[len(Q)-1])>3:
        print("Try a different k")
        return
    # Check if a square is found in expansion without needing to find relations.
    if is_square(Q[i+3]):
        if (i % 2)==1 and gcd(A[i+2]-sqrt(Q[i+3]),N)>1 and gcd(A[i+2]-sqrt(Q[i+3]),N)<N:
            return -1,gcd(A[i+2]-sqrt(Q[i+3]),N)

#Build factor base
FB=[2]
Pr=primes(3,B)
for p in Pr:
    ls=legendre_symbol(N,p)
    if ls==0:
        return p
        break
    if ls==1:
        FB.append(p)
setFB=Set(FB)

#Find FB smooth Q[i]
F=[]
G=[]
for i in range(3,len(Q)-1):
    if issmooth(Q[i]):
        F.append((-1)^i*Q[i])
        G.append(i)

V=VectorSpace(GF(2),len(FB)+1+len(F))
f=[]
for i in range(0,len(F)):
    e=[]
    e.append((1-sgn(F[i]))/2)
    for j in range(0,len(FB)):

```

```

        e.append(F[i].ord(FB[j]) % 2)
    for j in range(0,len(F)):
        if j==i:
            e.append(1)
        else:
            e.append(0)
    f.append(V(e))

#Find relations which produce a square mod N
rr=len(FB)
while rr>0:
    flg=0
    for i in range(len(F)):
        if f[i][rr]==1 and flg>0:
            f[i]=f[i]+f[flg]
        if f[i][rr]==1 and flg==0:
            flg=i
    rr=rr-1

for ff in f:
    mx=ff.support()[0]
    if mx>(len(FB)+1):
        a=1
        qq=1
        for w in ff.support():
            a=a*A[Q.index(abs(F[w-len(FB)-1]))-1] % N
            qq=qq*F[w-len(FB)-1]
        qs=abs(qq).sqrt()
        g1=gcd(a+qs,N)
        if g1>1 and g1<N:
            return g1
        g2=gcd(a-qs,N)
        if g2>1 and g2<N:
            return g2
    print("Try again")
return

def issmooth(d):
    """
    Checks if a prime d is FB-smooth
    """
    dp=Set(d.prime_factors())
    if dp.intersection(setFB)==dp:
        return True
    else:
        return False

def f(p,k,N):
    """
    Auxiliary function to try and determine a good value of k
    """
    if p==2:
        if k%2==0:
            return 1/3
        if (k*N)%4==3:

```

```

        return 1/3
    if (k*N)%8==5:
        return 2/3
    if (k*N)%8==1:
        return 4/3
    if (k%p)==0:
        return 1/(p+1)
    return (2*p)/(p^2-1)

def F(k,N):
    """
    Auxiliary function to determine a good value of k. See Brillhart and Morrison's paper for details.
    """
    tot=0
    for p in Pr:
        tot=tot+f(p,k,N)
    return (tot-(log(k))/2).n()

def factorbase(Bd):
    """
    Builds factor base for use in determining a good value of k.
    """
    global Pr
    Pr=[]
    for p in primes(1,Bd):
        Pr.append(p)

```