CHALMERS

GÖTEBORGS UNIVERSITET

Faktoriseringsalgoritmer och Kryptografi

Examensarbete för kandidatexamen i matematik vid Göteborgs universitet
Kandidatarbete inom civilingenjörsutbildningen vid Chalmers

David Elinder
Eric Lindström
Alexander Schälin
Mikael Strömstedt
Lukas Sundqvist

Institutionen för Matematiska vetenskaper
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2019

Faktoriseringsalgoritmer och Kryptografi

Examensarbete för kandidatexamen i matematik vid Göteborgs universitet
Alexander Schälin Lukas Sundqvist Mikael Strömstedt

Examensarbete för kandidatexamen i matematik inom Matematikprogrammet vid Göteborgs universitet
Eric Lindström

Kandidatarbete i matematik inom civilingenjörsprogrammet Teknisk matematik vid Chalmers
David Elinder

Handledare: Anders Södergren
Olga Balkanova
Examinator: Maria Roginskaya
Ulla Dinger

Institutionen för Matematiska vetenskaper
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2019

Hur säkra är dagens kryptosystem? - en populärvetenskaplig presentation

Stora mängder data om oss själva finns tillgängligt på internet. En del data som vi endast vill ska vara synligt för vissa personer finns i sociala medier. Vi skickar nästan dagligen meddelanden som är avsedda till en eller några behöriga personer. All den här datan måste på något sätt bevaras dold för obehöriga. Ytterligare ett område med känslig information är våra bankkonton. Många individer gör flera transaktioner per dag, några har sina bankuppgifter i sin smart-phone. Våra kontouppgifter måste hållas hemliga för obehöriga som kan övervaka och modifiera vår information. Kanske till och med stjäla våra pengar. Det här tar oss in på kryptering.

Transaktioner och känslig information bör skyddas av kryptering. Kryptering innebär att information omvandlas till en form av svårtydlig data känd som chiffertext. Chiffertexten är det som obehöriga kan få tillgång till men som inte ger dem någon känslig information. Detta bygger dock på att de inte har tillgång till en dold nyckel som används för att få tillbaka den känsliga informationen. Den här nyckeln är oftast ett stort heltal och ska endast vara känd av de som har behörighet till den känsliga informationen. Säkerheten i krypteringen beror främst på två saker. Dels att chiffertexten inte går att omvandla till den känsliga informationen utan nyckeln, men också att nyckeln förblir dold för de som inte har behörighet till informationen.

Det kanske inte är självklart att se när olika kryptosystem används. Men de kryptosystem vi behandlar i den här rapportena används flitligt i praktiken idag. Fastän det är lite mer än 40 år sedan de kom till. Det kryptosystem som används mest i praktiken är RSA kryptering. Kryptosystemet RSA använder väldigt stora beräkningar vilket kräver mycket datorkraft. Därför är det vanligare att RSA används på kortare meddelanden eller som en metod för att kryptera en dold nyckel från ett enklare kryptosystem.

Vårt syfte är att studera två av de kryptosystem som omvandlar information till chiffertext. De här kryptosystemen kan sedan användas av två personer för att skicka säkra meddelanden även om de inte har träffats i verkligheten. Vi kommer att studera hur säkra de här kryptosystemen är. Säkerheten studeras genom att undersöka hur nyckeln ska se ut för att obehöriga inte ska få tillgång till den känsliga informationen om de får tillgång till chiffertexten.

Vårt viktigaste resultat ur ett samhällsperspektiv är att de här kryptosystemen kan anses säkra. Därav behöver man inte vara orolig att förlora sin känsliga information, ifall något av de här kryptosystemen används till att skydda informationen.

För att komma fram till det här resultatet har vi gjort en litteraturstudie. Litteraturstudien har bekantat oss med några av de algoritmer som kan hota säkerheten i kryptosystemen vi har valt att undersöka. Därefter implementerade vi de här algoritmerna i programmeringsspråken C och C#. Till slut undersökte vi hur lång tid det tog för algoritmerna att hitta den dolda nyckeln.

Vi har även studerat en kvantalgoritm vid namn Shors algoritm som teoretiskt kan hota kryptosystemet RSA. Shors algoritm kräver mindre tid för att få tillgång till den dolda nyckeln än någon känd algoritm för klassiska datorer. Kvantalgoritmer kräver dock en kvantdator för att användas. I dagsläget är Shors algoritm inget verkligt hot då det ännu inte finns en tillräckligt kraftfull kvantdator för att utföra Shors algoritm. Men utvecklingen går snabbt framåt och experter ser kvantdatorer som det största hotet mot dagens kryptografi.

Sammanfattning

I detta arbete behandlas olika kryptosystem, de underliggande matematiska problem som håller kryptosystemen säkra och de algoritmer som löser dessa problem. De kryptosystem som behandlas är ElGamal och RSA. De underliggande problemen som behöver lösas för att knäcka kryptosystemen är diskreta logaritmproblemet för ElGamal och faktorisering av stora tal för RSA. De lösningsalgoritmer vi diskuterar för att lösa det diskreta logaritmproblemet är en direkt metod och Shanks babystep-giantstep algoritm. För att faktorisera stora tal använder vi en direkt metod, Pollards rho-algoritm, Fermats algoritm, Dixons algoritm, Kedjebråksmetoden och Kvadratiskt såll. Vi analyserar även algoritmer för primtalstest vilka är viktiga för RSA kryptering. De algoritmer för primtalstest som behandlas är en direkt metod, Solovay-Strassens test och Miller-Rabins test. Det resultat vi fick var att dessa kryptosystem kan anses säkra eftersom de på kort tid kan kryptera tal av storleken 10^{1000} och lösningsalgoritmerna med våra implementationer inte kan faktorisera tal av storlek 10^{100} inom rimlig tid. Vi beskriver också en kvantalgoritm, vid namn Shors algoritm, som skulle kunna vara ett framtida hot mot dessa system. Detta ses dock inte som ett problem idag då det än så länge inte finns några tillräckligt kraftfulla kvantdatorer som kan implementera algoritmen på en tillräckligt omfattande skala.

Abstract

In this paper we analyse different cryptosystems, the underlying mathematical problems that keep these cryptosystems safe, and algorithms that solve these problems. The cryptosystems being analysed are ElGamal and RSA. The underlying problems that have to be solved to break the cryptosystems are the discrete logarithm problem for ElGamal and factorization of large integers for RSA. The algorithms we studied that solve the discrete logarithm problem where a trial method and Shank's Babystep-Giantstep algorithm. To factorize large integers we used trial division, Pollard's Rho-algorithm, Fermat's algorithm, Dixon's algorithm, the Continued-fraction Method and the Quadratic Sieve. We also analysed primality tests since they are important in the RSA cryptosystem. The primality tests included in the analysis where a trial method, Solovay-Strassen's test and Miller-Rabin's test. The result we obtained is that these cryptosystems can be considered safe since they can encrypt messages using numbers of the magnitude 10^{1000} while the solution algorithms can not solve the underlying problems, in any reasonable time, for numbers of the magnitude 10^{100}. We also describe a quantum algorithm, called Shor's algorithm, which could be a future threat against these systems. This however, is not considered as an issue today since there are no quantum computers that are powerful enough to implement the algorithm on a wide scale.

Innehåll

1 Inledning **1**
1.1 Syfte och metod . . . 1

2 Bakgrund **1**
2.1 Diffie-Hellman key exchange . . . 1
2.2 ElGamals kryptosystem. . . . 2
2.3 RSA-kryptering . . . 3
2.3.1 Personen-i-mitten-attacken . . . 4

3 Diskreta logaritmproblemet **4**
3.1 Direkt Metod för att lösa DLP . . . 4
3.2 Kollisionsalgoritmen Shanks Babystep-Giantstep . . . 5
3.3 Resultat . . . 6

4 Primtalstest **6**
4.1 Direkt Metod för primtalstest . . . 6
4.2 Solovay-Strassens test . . . 7
4.3 Miller-Rabins test . . . 7
4.3.1 Algoritmen . . . 8
4.4 Resultat . . . 8

5 Faktoriseringsalgoritmer **8**
5.1 Direkt Metod för faktorisering . . . 9
5.2 Pollards rho-algoritm . . . 9
5.3 Fermats algoritm . . . 10
5.4 Dixons algoritm . . . 11
5.4.1 Komplexitet . . . 12
5.5 Kedjebråksmetoden . . . 12
5.5.1 Enkla kedjebråk . . . 12
5.5.2 Idé . . . 13
5.5.3 Algoritmen . . . 13
5.6 Kvadratiskt såll . . . 14
5.6.1 Idé . . . 14
5.6.2 Algoritm . . . 15
5.7 Resultat . . . 16

6 En kvantalgoritm **16**
6.1 En kort introduktion till kvantfysik . . . 16
6.2 En kort introduktion till kvantberäkningar . . . 17
6.3 Kvantfouriertransformen . . . 18
6.3.1 Klassisk diskret fouriertransform . . . 18
6.3.2 Kvantfouriertransform . . . 18
6.4 Shors faktoriseringsalgoritm . . . 18
6.4.1 Shors periodsökaralgoritm . . . 19
6.4.2 Enkel framgångssannolikhet då $r|Q$ . . . 19
6.4.3 Generell framgångssannolikhet då $r \nmid Q$ . . . 19

7 Slutsats **20**

Referenser **21**

A Fördjupning **23**
A.1 Kinesiska restklassatsen . . . 23
A.2 Komplexitet . . . 24
A.3 Fast Powering Algorithm . . . 25
A.4 Sats A.4.1 och A.4.2 med bevis . . . 26
A.5 RSA-algoritmen . . . 26
A.6 Försållning . . . 27
A.7 Bevis av proposition 5.3.1 . . . 27
A.8 Översätta meddelanden till tal . . . 27
A.9 Att finna inverser modulo primtal. . . . 28
A.10 Lösning till (5.2.2) . . . 28
A.11 Kollision i Pollards rho-algoritm . . . 28
A.12 Alternativ version av Pollards rho-algoritm . . . 29
A.13 Komplexitet för att finna ett matchande element mellan två listor . . . 31
A.14 Komplexitet för Dixons algoritm . . . 32
A.15 Förbättringar Kedjebråksmetoden . . . 33
A.15.1 Legendresymbolen . . . 33
A.15.2 Övre gräns till primtalsbas . . . 33

B Kvantteori **33**
B.1 Dirac-notation . . . 33
B.2 Unitära matriser . . . 33
B.3 Tensorprodukt . . . 34
B.4 Bevis för fouriertransform . . . 34
B.4.1 F_N är en unitär matris . . . 34
B.4.2 Snabb fouriertransform . . . 34
B.5 Kvantfouriertransform . . . 35
B.6 Sannolikhet för lyckat val av x i Shors algoritm . . . 35
B.7 Sannolikhet för $\text{sgd}(c, r) = 1$ . . . 36
B.8 Bevis av lemma . . . 36
B.9 Analys av $f(x) = \frac{\sin^2\left(\frac{\pi m x}{Q}\right)}{\sin^2\left(\frac{\pi x}{Q}\right)}$ . . . 37
B.10 Komplexitet hos Shors algoritm . . . 37

C Information kring test av algoritmer **38**
C.1 Tal . . . 38
C.2 Testbädd . . . 38

D Kod **40**
D.1 Direkt metod för DLP . . . 40
D.2 Shanks Babystep-Giantstep . . . 41
D.3 Direkt metod för primtalstest & faktorisering . . . 46
D.4 Fermats faktoriseringsalgoritm . . . 47
D.5 Dixons faktoriseringsalgoritm . . . 51
D.6 Pollard-Rho . . . 60
D.7 Kvadratiskt Såll . . . 65
D.8 Kedjebråksmetoden . . . 72
D.9 Solovay-Strassens test . . . 81
D.10 RSA tillsammans med Miller-Rabin . . . 82

Förord

Vi har fört en loggbok, dagbok, och möteslogg under arbetets gång. Arbetet började med en litteraturstudie och genomarbetning av Diffie-hellman, ElGamal och RSA. David behandlade även Shanks-algoritm. Efter det delade gruppen upp sig, Eric fokuserade på kvantdelen och resten betraktade primalitetstest. När detta var klart gjordes en ytterliggare uppdelning. Mikael och Lukas bettraktade Kedjebråk respektive Kvadratiskt såll medans Alexander och David behandlade Pollards rho-algoritm och Fermats algoritm. Eftersom det fanns tid behandlades även Dixons algoritm. Presentationer av teorin från vissa delar gjordes inför gruppen och handledarna för att ge bättre förståelse.

Nedan betecknas vilka avsnitt varje enskild person var huvudansvarig för att skriva, notera att vissa avsnitt delades mellan flera personer.

Alexander Schälin: Populärvetenskaplig presentation, Inledning, 1.1, Bakgrund, 2.1, Faktoriseringsalgoritmer, 5.2, 5.4, A.10, A.11, A.14, D.6, D.9.
Alexander har fört en möteslogg under arbetets gång. Han har även varit delaktig i arbetet med Pollards rho-algoritm, Fermats algoritm, Solovay-Strassens test samt de direkta metoderna.

Lukas Sundqvist: Inledning, 1.1, Diskreta logaritmproblemet, 3.1, 3.3, Primalitetsproblemet, 4.1, 4.2, 5.1, 5.6, Resultatdelarna, 7, App. C, D.1, D.3, D.9, D.7.
Lukas har varit ansvarig för allt tekniskt kring svn-systemet samt satte upp referenssystem, formatering, och kortkommandon i LaTeX. Har mest arbetat med det Kvadratiska Sållet samt skrivande av rapporten. Har även spendarat tid med enklare algoritmer som Solovay-Strassens test och de direkta metoderna.

Eric Lindström: Kapitel 6, A.1, Kapitel B.
Eric har tillsammans med gruppen gått igenom grunderna i kryptografi och sedan fördjupat sig i Shors algoritmen. För denna uppgift var det nödvändigt att först förstå grunderna i kvantfysik, kvantdatorer, kvantfouriertransformen och flera mindre kvantalgoritmer.

Mikael Strömstedt: 2.3, 4.3, 5.5, A.4, A.5, A.6, A.15, C.2, D.8, D.10.
Under arbetets gång upptäckte Mikael ett försållningssystem vilket kunde utnyttjas för att minska körtiden för slumpmässig framtagning av stora primtal. Senare insåg vi att detta sättet redan är vedertaget. Mikael har implementerat ett fullt fungerande RSA-kryptosystem, och varit ansvarig för att föra gruppens dagbok.

David Elinder: Sammanfattning, Abstract, Inledning, 2.2, 3.2, 5.2, 5.3, 5.4, A.2, A.3, A.7, A.8, A.9, A.12, A.13, D.2, D.4, D.5, D.6.
Under arbetets gång har David tagit fram många koder för implementationen av dessa algoritmer. För att spara plats har endast de bästa och senaste algoritmerna presenterats i Appendix och majoriteten utelämnats. Den alternativa versionen av Pollards rho-algoritm i Appendix A.12 ansågs först inte tillräckligt utförlig och en mer detaljerad version skrevs samman av David men denna utelämnades för att spara tid.

Vi vill tacka våra handledare Anders Södergren och Olga Balkanova, som har varit till stor hjälp och väglett oss under arbetets gång.

Notation	Förklaring	Referens
$\mathbb{F}_p$	Den ändliga kroppen med p element.	[1]
$\mathbb{F}_p^*$	Den multiplikativa delgruppen av kroppen $\mathbb{F}_p$.	[1]
$\left(\frac{b}{n}\right)$	Jacobisymbolen.	[2, s.47]
$\mathcal{O}(\cdot)$	Övre gräns för komplexitet.	A.2.1
$\Omega(\cdot)$	Undre gräns för komplexitet.	A.2.1
$\Theta(\cdot)$	Intervall för komplexitet.	A.2.1

1 Inledning

Hemlig kommunikation är en viktig del av många människors vardag. Det används när du skriver in dina bankuppgifter online och när du skickar e-post eller SMS. Symmetriska kryptosystem, där nyckeln för kryptering och dekryptering är densamma, är beroende av att nyckeln är hemlig och att de två kommunicerande parterna kan utväxla nyckeln på ett säkert sätt. Om du aldrig träffar den du kommunicerar med finns det dock ett annat sätt att skicka nyckeln osedd, via asymmetrisk kryptering.

Gemensamt för asymmetriska system är att det finns två nycklar, en publik för att kryptera och en privat för att dekryptera. Diffie-Hellmans algoritm var det första exemplet på asymmetrisk kryptering, och kom ut år 1976 (se [3, s.59] och [4]). Två år senare följde RSA vilket används flitigt idag, t.ex. när man säkert ansluter till en hemsida via SSL/HTTPS [5]. Även Diffie-Hellman används vid anslutning via SSL/HTTPS, i ssh-sessioner, och VPN anslutningar [6]. Däremot kan Diffie-Hellman endast skapa en dold, gemensam nyckel och inte kryptera och dekryptera meddelanden. Diffie-Hellman är därmed inte ett kryptosystem.

Efter RSA följde ElGamals kryptosystem år 1985 [7]. ElGamals kryptosystem är en förlängning av Diffie-Hellman. Även om RSA var det första kryptosystemet som använde de koncept som behandlades av i Diffie och Hellmans artikel så kommer vi se att ElGamals system är mycket närmare Diffie-Hellmans faktiska implementation av dessa idéer.

Säkerheten i de asymmetriska kryptosystemen ElGamal och RSA bygger på hur svårt det är att finna den privata nyckeln givet den publika nyckeln som alla har tillgång till. Vi kommer främst att studera när de här kryptosystemen kan anses vara säkra.

Genom förbättringar i datorers beräkningskraft har det som tidigare varit omöjligt blivit rutin, så frågan är om det vi anser säkert idag fortfarande är det om 30 år. Kryptosystemet RSA-129 uppskattades år 1976 av Martin Gardner att vara säkert i 40 *biljarder år*. År 1994, mindre än 20 år senare, visade det sig falskt när ett 129 siffrigt tal ur den kända RSA-129 utmaningen faktoriserades för första gången [8, s.1]. RSA hotas också av kvantalgoritmer. Idag finns det inga kvantdatorer som är kraftulla nog för att RSA ska anses osäkert, men detta tros vara verklighet inom en snar framtid [9]. År 1994 beskrev Peter Shor en algoritm som teoretiskt sett hittar icke-triviala faktorer till ett tal nästan exponentiellt mycket snabbare än någon känd algoritm som använder sig av klassiska datorer [10].

1.1 Syfte och metod

Det övergripande syftet med arbetet är att undersöka hur väl dagens asymmetriska kryptosystem fungerar, vilket vi reducerar ner till tre problem. De tre problemen vi undersöker är säkerheten i ElGamal, att hitta stora primtal samt säkerheten i RSA. Störst vikt läggs vid säkerheten i RSA genom att studera faktoriseringsalgoritmer. Vi studerar även hur dess säkerhet potentiellt kan hotas av en kvantalgoritm. Metoden består av en litteraturstudie, implementering av algoritmerna och test av deras hastighet med våra implementationer.

2 Bakgrund

I det här kapitlet introducerar vi grundläggande teori inom kryptografi. Vi börjar med att gå igenom Diffie-Hellman key exchange algoritm. Därefter förklarar vi ElGamal's kryptosystem för att sedan avsluta med kryptosystemet RSA. För att underlätta förståelsen för hur meddelanden sänds kommer vi att använda oss av karaktärerna Alice, Bob och Eve. Alice och Bob är de som vill kommunicera genom ett kryptosystem. Eve vill läsa deras meddelanden men har endast tillgång till chiffertexten och den publika nyckeln.

2.1 Diffie-Hellman key exchange

Som det nämndes i inledningen är symmetrisk kryptering beroende av en enda nyckel som måste bevaras hemlig från de som inte ska ha tillgång till meddelandena. Diffie-Hellman key exchange är en algoritm som löser dilemmat att Alice och Bob ska enas om en nyckel utan att träffas

i verkligheten och som förblir hemlig för Eve. Därmed är Diffie-Hellman inte ett kryptosystem eftersom det inte omvandlar ett meddelande till chiffertext.

Låt oss nu gå igenom Diffie-Hellman key exchange. Alice och Bob vill kommunicera med hemliga meddelanden. För att utföra det här utan att Eve ska kunna dekryptera de meddelandena som skickas mellan Alice och Bob måste de enas om en hemlig nyckel. Första steget i Diffie-Hellman key exchange är att Alice och Bob enas om ett stort primtal p och ett positivt heltal $g \pmod p$. Det är viktigt att g är en generator[1] till $\mathbb{F}_p^*$, annars är g^x inte längre en injektiv funktion och systemet blir mindre säkert då det finns fler sätt att skapa den privata nyckeln. Notera att de här talen är publik information, alltså synligt för Eve. Nästa steg är att Alice och Bob väljer varsitt hemligt heltal, a respektive b, som de håller hemliga för Eve och varandra. De räknar sedan ut

$$A \equiv g^a \pmod p \text{ respektive } B \equiv g^b \pmod p.$$

Därefter skickar de A och B till varandra. Alltså är A och B tillgängliga för Eve men inte a och b. Till sist räknar de ut

$$A' \equiv B^a \pmod p \text{ och } B' \equiv A^b \pmod p.$$

Vi har då att

$$A' \equiv B^a \equiv (g^b)^a \equiv g^{ab} \equiv (g^a)^b \equiv A^b \equiv B' \pmod p.$$

Alltså har de samma nyckel men Eve vet inte om den eftersom a och b förblir dolda.

2.2 ElGamals kryptosystem.

Förklaringen av ElGamals kryptosystem är snarlik den för Diffie-Hellman. Alice får meddelanden från Bob och Eve lyssnar på deras kommunikationen. Först väljer Alice ett primtal p och en generator till $\mathbb{F}_p^*$ kallad g. Hon väljer sedan ett hemligt tal a och beräknar

$$A \equiv g^a \pmod p.$$

Sedan gör hon p, g och A publika så att alla kan se dem. Nu kan vem som helst skicka meddelanden till henne. Notera att det bara är Alice som känner till värdet på a. Bob skriver sitt meddelande som ett tal m sådant att $1 \leq m \leq p-1$, se Appendix A.8, och väljer sedan ett slumpmässigt tal k inom intervallet $1 \leq k \leq p-1$. Bob räknar ut

$$C_1 \equiv g^k \pmod p, \; C_2 \equiv mA^k \pmod p$$

och skickar (C_1, C_2) till Alice. För att dekryptera detta beräknar Alice

$$m' \equiv ((C_1)^a)^{-1} C_2 \pmod p.$$

Nu är vi klara eftersom

$$m' \equiv ((C_1)^a)^{-1} C_2 \equiv \left(\left(g^k\right)^a\right)^{-1} \left(mA^k\right) \equiv g^{-ak} m \left(g^a\right)^k \equiv g^{-ak} g^{ak} m \equiv m \pmod p.$$

Notera. Om Bob väljer $1 \leq m \leq p-1$ så behöver Alice bara beräkna $h \equiv m' \pmod p$, $1 \leq h \leq p-1$, så får hon att $m = h$.

För att skicka sitt krypterade meddelande behöver Bob skicka C_1 och C_2 men oftast är storlekarna på m, C_1 och C_2 ungefär samma vilket ger att Bob då behöver skicka dubbelt så många bits som meddelandet innehåller.

En viktig fråga är om ElGamals kryptosystem är lika säkert som Diffie-Hellman eller om vi infört några säkerhetsbrister i detta system. Frågan besvaras av följande sats.

Sats 2.2.1. *Låt funktionerna $f : \left(\mathbb{F}_p^*\right)^4 \to \mathbb{F}_p^*$ och $h : \left(\mathbb{F}_p^*\right)^5 \to \mathbb{F}_p^*$ ta in de kända parametrarna i Diffie-Hellman resp ElGamal och returnera lösningarna, d.v.s. $f(A, B, g, p) = g^{ab}$, $h(C_1, C_2, A, g, p) = (C_1^a)^{-1} C_2$. Då kan man på ett beräkningseffektivt sett även lösa Diffie-Hellman med h och ElGamal med f.*

[1] Vi vet att g existerar enligt Sats 1.30 i [3]

Bevis. Vi börjar med att lösa Diffie-Hellman med hjälp av h. Vi beräknar $h\left(B,1,A,g,p\right)=\left(B^a\right)^{-1}$ och observerar att $\left(B^a\right)^{-1}\equiv\left(\left(g^b\right)^a\right)^{-1}\equiv g^{-ab}\pmod p$. Sedan beräknar vi $\left(g^{-ab}\right)^{-1}\equiv g^{ab}\pmod p$ enligt appendix A.9 och därav är vi klara.

Nu visar vi att man kan lösa ElGamal med f. Observerar att $f(A,C_1,g,p)=f(g^a,g^k,g,p)=g^{ak}$. Vi beräknar $g^{-ak}\pmod p$ från g^{ak} enligt appendix A.9. Slutligen beräknar vi $C_2g^{-ak}\equiv mA^kg^{-ak}\equiv m\left(g^a\right)^kg^{-ak}\equiv mg^{ak}g^{-ak}\equiv m\pmod p$, vilket slutför beviset. □

Vad Sats 2.2.1 säger oss är att om vi kan lösa det ena problemet så kan vi enkelt lösa det andra problemet, varvid problemen är lika svåra att lösa. Notera att vi i satsen ovan endast multiplicerar och upphöjer tal modulos p, vilket gör att vi ökar komplexiteten med $\mathcal{O}(\log(p))$, se Appendix A.3. Vi kommer se att de algoritmer som löser Diffiehellman och ElGamal har mycket större komplexitet vilket ger att den totala komplexiteten inte ökar.

2.3 RSA-kryptering

Idén med RSA-kryptering är att kryptera ett meddelande med en publik nyckel och dekryptera med en privat nyckel. Den publika nyckeln består av två tal e, n där n är produkten av två privata primtal p, q, båda större än 2, och e är ett positivt heltal sådant att

$$\text{sgd}(e,\ (p-1)(q-1))=1. \tag{2.3.1}$$

När Alice vill skicka ett meddelande m_1 till Bob frågar hon honom vilken publik nyckel han har. Bob ger Alice informationen $(e,\ n)$. Därefter beräknar Alice chiffret c

$$c\equiv m_1^e\pmod n.$$

Eftersom ekvationen (2.3.1) är uppfylld finns det enligt Sats A.4.2 ett d sådant att

$$ed\equiv 1\pmod{(p-1)(q-1)}$$

Bob använder d som sin privata nyckel.

Sats 2.3.1. *Antag att*

$$sgd(c,\ pq)=1\ och\ de\equiv 1\pmod{(p-1)(q-1)},\quad p,\ q\ udda\ primtal.$$

Då har ekvationen

$$c\equiv x^e\pmod{pq} \tag{2.3.2}$$

den unika lösningen $x\equiv c^d\pmod{pq}$.

Bevis. Sätt in $x=c^d$ i (2.3.2). Vi har:

$$x^e\equiv c^{de}\equiv c^{1+k(p-1)(q-1)}\equiv cc^{k(p-1)(q-1)}\equiv c\left(c^{(p-1)(q-1)}\right)^k\equiv c\pmod{pq} \tag{2.3.3}$$

ty enl Sats A.4.1 är $(c^{(p-1)(q-1)/2})^k\equiv 1\pmod{pq}\implies(c^{(p-1)(q-1)})^k\equiv 1^{2k}\pmod{pq}$.

Detta visar att c^d är en lösning.

Låt nu u vara en lösning till ekvationen (2.3.2). Vi använder åter igen sats A.4.1 och får:

$$u\equiv u^1\equiv u^{de-k(p-1)(q-1)}\equiv\left(u^e\right)^d\left(u^{(p-1)(q-1)}\right)^{-k}\equiv\left(u^e\right)^d\cdot 1^{-k}\equiv c^d\pmod{pq} \tag{2.3.4}$$

vilket visar att c^d är den unika lösningen till ekvation (2.3.2).

□

Bob tar emot chiffret c och beräknar

$$m_2\equiv c^d\pmod n,$$

och enligt Sats 2.3.1 kan Bob lita på att $m_1\equiv m_2\pmod n$.

Antag nu att Eve vill läsa m_1. Till sitt förfogande har hon c, e och n. Hon vet att för något heltal y gäller $m_1 \equiv c^y \pmod{n}$ och att $ey \equiv 1 \pmod{(i-1)(j-1)}$ där $ij = n$. Kan Eve faktorisera n och beräkna

$$(i-1)(j-1) = ij - (i+j) + 1 \tag{2.3.5}$$

är hon klar! Observera att det är tillräckligt för henne att känna till kvantiteten $i + j$ varför $(i-1)(j-1)$ hålls privat. Skulle $(i-1)(j-1)$ vara känd för Eve får hon $i+j$ ur ekvation (2.3.5) för att sedan lösa ekvationen

$$0 = (x-i)(x-j) = x^2 - (i+j)x + ij$$

som bekant har rötterna i, j. Bob väljer därför så stora primtal p,q att dagens datorkraft och faktoriseringsalgoritmer inte räcker för Eve att inom ramen för rimlig tid kunna faktorisera n.

2.3.1 Personen-i-mitten-attacken

Eve behöver inte nödvändigtvis utföra all aritmetik för att ta del av Alice och Bobs kommunikation. Antag som ovan att Alice vill skicka Bob ett krypterat meddelande. Antag vidare att Eve fungerar som ett slags hemlig mellanhand i kommunikationen och tar emot c från Alice. Eve begär bekräftelse från Bob att han är den som är ämnad att ta emot meddelandet genom att be honom kryptera meddelandet $ck^e \pmod{n}$ där k är heltal som Eve hittat på. Bob returnerar då $(ck^e)^d \pmod{n}$ var på Eve gör kalkylen (se ekvation (2.3.3))

$$(ck^e)^d \equiv (m^e k^e)^d \equiv m^{ed} k^{ed} \equiv mk \pmod{n}$$

Eftersom Eve känner till k löser hon lätt ut Alice meddelande m.

3 Diskreta logaritmproblemet

Diffie-Hellman och därmed även ElGamal-systemet baserar sin säkerhet på ett svårt matematiskt problem vid namn diskreta logaritmproblemet. För alla p som är primtal existerar kroppen $\mathbb{F}_p$ och det finns en generator g till $\mathbb{F}_p*$. Vi definierar funktionen,

$$\begin{aligned} g &: \mathbb{F}_p \to \mathbb{F}_p^* \\ g(x) &= g^x. \end{aligned}$$

Det är enkelt att räkna ut $g(x)$ när man vet g och x, med hjälp av t.ex. Fast Powering Algorithm (se Appendix A.3). Det man kallar diskreta logaritmproblemet är följande:

Givet ett g som är en generator till $\mathbb{F}_p^*$, och ett $h \in \mathbb{F}_p^*$, hitta $x \in \mathbb{F}_p$ sådant att $g^x = h$. (DLP)

Vi redogör nu för några olika algoritmer som löser DLP.

3.1 Direkt Metod för att lösa DLP

Det mest intuitiva sättet att lösa DLP är att börja med ett slumpmässigt valt tal y, beräkna g^y, och se om det överensstämmer med det givna värdet h. Om det inte stämmer testas ett nytt y. Nedan följer en algoritm.

Steg 1: Sätt $K = \mathbb{F}_p$.

Steg 2: Välj ett slumpmässigt element $y \in K$. Om detta är omöjligt för att $K = \emptyset$ avslutas algoritmen, det finns ingen lösning.

Steg 3: Beräkna g^y, om $g^y = h$ avslutas algoritmen, y är lösningen. Gå annars vidare till Steg 4.

Steg 4: Sätt $K = K \setminus \{y\}$ och gå tillbaka till Steg 2.

Komplexiten för att räkna ut g^y är $\mathcal{O}(\log(y))$ med användning av Fast Powering Algorithm[2]. Då algoritmen för den direkta metoden kräver max p steg ger detta i värsta fall en total komplexitet på $\mathcal{O}\big(p \times \log(p)\big)$. I praktiska tillämpningar när p är stort så är metoden olämplig. I vår implementering har vi ytterligare förenklat algoritmen, istället för att välja slumpmässiga värden på y börjar vi på 0 och går igenom hela $\mathbb{F}_p$ i ordning. Detta påverkar inte komplexiteten.

3.2 Kollisionsalgoritmen Shanks Babystep-Giantstep

En algoritm med en lägre tidskomplexitet är Shanks Babystep-Giantstep-algoritm. Vi börjar med att betrakta algoritmen. Låt $g \in \mathbb{F}_p$, $h \equiv g^x \pmod{p}$ och $0 \leq h < p$.

Steg 1: Beräkna $n = 1 + \lfloor \sqrt{p} \rfloor$.

Steg 2: Skapa Lista 1: g^k, $0 \leq k \leq n$.

Steg 3: Beräkna g^{-n}.

Steg 4: Skapa Lista 2: $h \cdot g^{-kn}$, $0 \leq k \leq n$.

Steg 5: Hitta i, j sådana att $g^i = hg^{-jn}$.

Steg 6: Beräkna $x = i + jn$.

Det finns nu några saker som är värda att notera och förklara med denna algoritm. Vi ser att $n > \sqrt{p}$. För att skapa Lista 1 har vi att $g^0 = e$ och $g^{k+1} = g \cdot g^k$, så vi behöver bara multiplicera med g för att hitta nästa element i listan. För att beräkna g^{-n} använder vi att sista elementet i Lista 1 är g^n och sedan samma metod som i Appendix A.9. För att beräkna elementen i Lista 2 har vi första element h och följande element ges av att multiplicera föregående element med g^{-n}. I Steg 5 är g^i det i:te elementet i Lista 1 och $h \cdot g^{-jn}$ är det j:te elementet i Lista 2. Att finna i, j som uppfyller att $g^i = hg^{-jn}$ diskuteras i Appendix A.13.

Sats 3.2.1. *Shanks Babystep-Giantstep-algoritm löser DLP med komplexitet* $\mathcal{O}\left(\sqrt{p}\log\left(p\right)\right)$.

Bevis. Först visar vi att Shanks Babystep-Giantstep algoritm löser DLP. Vi använder här samma beteckningar som i algoritmen. Antag att det finns ett tal x sådant att $0 \leq x < p$ löser $g^x \equiv h \pmod{p}$. Vi kan då skriva x på formen $x = qn + r$ där $0 \leq r < n$. Vi har även att $q = \frac{x-r}{n} \leq \frac{x}{n} < \frac{p}{n} < \frac{n^2}{n} = n$. Notera även att $0 \leq q$ eftersom $0 \leq x$. Nu har vi att

$$h \equiv g^x \equiv g^{qn+r} \equiv g^r \cdot g^{qn} \pmod{p},$$

vilket ger att

$$g^r \equiv h \cdot g^{-qn} \pmod{p}.$$

Vi vet att $g^r \pmod{p}$ finns i Lista 1 eftersom $0 \leq r < n$ och att $h \cdot g^{-qn} \pmod{p}$ finns i Lista 2 eftersom $0 \leq q < n$, och eftersom algoritmen testar att matcha alla element i listorna med varandra så måste algoritmen finna x.

Slutligen studerar vi komplexiteten av algoritmen. Vi kommer beräkna komplexiteten som antal beräkningar med multiplikation, addition, kvadratrot etc. och kommer inte ta hänsyn till hur svårt det är att t.ex. multiplicera stora tal. Vi ser att komplexiteten för varje steg är

Steg 1: $\mathcal{O}\left(1\right)$ eftersom vi endast gör beräkningar med grundläggande operationer, se Appendix A.2.1.

Steg 2: $\mathcal{O}\left(n\right)$ eftersom vi endast multiplicerar med g för varje element i listan.

Steg 3: $\mathcal{O}\left(\log\left(p\right)\right) = \mathcal{O}\left(\log\left(\sqrt{p}\right)\right) = \mathcal{O}\left(\log\left(n\right)\right)$, se Appendix A.9.

Steg 4: $\mathcal{O}\left(n\right)$ eftersom vi endast multiplicerar med g^{-n} för varje element i listan.

Steg 5: $\mathcal{O}\left(n\log\left(n\right)\right)$, se Appendix A.13.

[2]Se Appendix A.3

Steg 6: $\mathcal{O}(1)$ eftersom vi endast adderar och multiplicerar ett fixt antal gånger.

Den totala komplexiteten blir då summan av komplexiteterna i de olika stegen vilket ger komplexiteten $\mathcal{O}(n\log(n))$. Om vi nu sätter in att $n = 1 + \lfloor\sqrt{p}\rfloor$ och förenklar så får vi att den slutliga komplexiteten blir

$$\mathcal{O}(n\log(n)) = \mathcal{O}((1+\lfloor\sqrt{p}\rfloor)\log(1+\lfloor\sqrt{p}\rfloor)) = \mathcal{O}(\sqrt{p}\log(p)),$$

vilket var vad vi ville visa. □

Ett problem som hindrar Shanks Babystep-Giantstep algoritm från att vara användbar i praktiken är minnesanvändningen. För stora tal krävs mycket fysiskt minne för att spara Lista 1 och Lista 2. Om $2^{h+1} < p$ och $h \in \mathbb{N}$ så är $2^{h/2} < \sqrt{p} < n$. Då måste varje plats i listorna kunna spara tal större än 2^{h+1} vilket kräver mer än h bits i en vanlig dator. Eftersom listorna är n långa så får vi att det behövs mer än $2^{h/2}$ sådana tal. Vi får nu att det krävs mer än $h2^{h/2}$ bits för att spara en lista. Om vi låter varje lista ta upp en terabyte av minnet så får vi att

$$h2^{h/2} < 10^{12}$$

vilket ger att $h \leq 67$. Då de tal man använder i praktiken för kryptering ofta har flera hundra bits så är detta inte en så användbar algoritm i praktiken.

3.3 Resultat

Vi har testat algoritmerna för olika värden på p, g, och h på en gemensam testbädd, se Appendix C. Vår implementation av Shanks algoritm tog slut på datorns minne vid test av det sista talet.

p	g	$h = g^x$	Direkt metod	Shanks	x
502217	12653	64641	0,058s.	0,008 s.	190447
4129232533	126543	64641	8m. 44s.	0.037 s.	1418331086
4129232533	123134134	1325278813	0,005s.	0,036 s.	200
123456791	123134134	67277073	4,109s.	0,015 s.	12833413
Primtal $\approx 10^{231}$	$\approx 10^{226}$	$\approx 10^{227}$	Avbröts.	Avbröts.	Okänt

4 Primtalstest

Asymmetriska kryptosystem bygger på att det är relativt lätt att kryptera och dekryptera meddelandet med fullständig information, och svårt att dekryptera ett meddelande med endast den offentliga informationen. En del i krypteringen av meddelanden när man använder RSA är att hitta stora primtal. För att hitta stora primtal genereras först ett stort slumpmässigt heltal n, vilket ger upphov till *Primalitetsproblemet*: Att bestämma definitivt eller med hög sannolikhet om n är ett primtal. För att lösa primalitetsproblemet använder man så kallad *primtalstestning*. I det här Kapitlet går vi igenom tre algoritmer för att avgöra om ett givet tal n är ett sammansatt tal eller ett primtal.

4.1 Direkt Metod för primtalstest

Den mest grundläggande metoden för att lösa Primalitetsproblemet är att dela vårt tal n med $i = 2, 3, 4, \cdots, \lfloor\sqrt{n}\rfloor$. Ger en av dessa operationer ett heltal är vi klara, n är sammansatt. Om vi går igenom hela listan utan att få ett heltal är n definitivt ett primtal. Detta är den enda metoden vi redovisar som med säkerhet kan bestämma om n är ett primtal eller sammansatt. Ett sätt att snabba upp processen är att endast testa för udda värden på i.

Steg 1: Antag att n är ett positivt udda tal, välj $i = 3$.

Steg 2: Beräkna $\frac{n}{i}$. Om det är ett heltal avbryts algoritmen, n är ett sammansatt tal. Får vi inte ett heltal går vi vidare till Steg 3.

Steg 3: Om $i \geq \lfloor \sqrt{n} \rfloor$ avbryts algoritmen, n är ett primtal. Annars välj $i = i + 2$ och gå till Steg 2.

Anledningen till att vi med säkerhet kan bestämma om n är ett primtal är att vi testar alla möjliga primtalsfaktorer. n är udda och kan därför inte ha 2 eller något jämnt tal som faktor, och vi testar alla udda tal mindre än $\lceil \sqrt{n} \rceil$. Algoritmen tar i värsta fall $\sqrt{n}$ steg och utför lika många divisioner. Vi har därmed en komplexitet på $\mathcal{O}(\sqrt{n})$.

4.2 Solovay-Strassens test

För att snabbare bestämma om ett positivt heltal n är ett primtal kan man använda stokastiska test. Ett stokastiskt test visar aldrig definitivt att n är ett primtal, utan kan endast definitivt visa att n är sammansatt. Låt $\left(\frac{b}{n}\right)$ vara Jacobisymbolen. Eulers kriterium säger oss att (se [2, s.43])

Proposition 4.2.1. *Låt p vara ett primtal. Då gäller att,*

$$b^{(p-1)/2} \equiv \left(\frac{b}{p}\right) \pmod{p}, \quad \textit{för varje heltal } b. \tag{4.2.1}$$

Det finns inget som hindrar sambandet (4.2.1) att gälla för sammansatta n. Baserat på definitionen av ett *Euler-pseudoprimtal* (se [2, s.128]) bildar vi följande definition.

Definition 4.2.2. Låt n vara ett sammansatt tal, låt $b \in (0, n)$ vara ett heltal. Vi säger att b är ett *Euler-vittne* till n om $\text{sgd}(b, n) \neq 1$ eller om

$$b^{(n-1)/2} \not\equiv \left(\frac{b}{n}\right) \pmod{n}. \tag{4.2.2}$$

Anledningen till att vi kan konstruera ett primtalstest följer från nästa proposition.

Proposition 4.2.3. *Låt n vara ett sammansatt tal, då gäller att minst 50% av alla heltal b mellan 1 och $n-1$ är Euler-vittnen till n (se Solovays och Strassens bevis i [11]).*

Propositionen ger oss en möjlighet att uppskatta sannolikheten att n är ett primtal efter ett fullbordat test. Vi kan nu beskriva en algoritm för att utföra *Solovay-Strassens test.*

Steg 1: Antag att n är ett positivt udda tal, välj slumpmässigt k heltal b sådana att $0 < b < n$ för alla b. Vi har då en lista $[b_1, b_2, \ldots, b_k]$. Sätt $i = 1$.

Steg 2: Använd b_i och beräkna både vänster och högerled i (4.2.1).

Steg 3: Om ekvation (4.2.2) uppfylls vittnar b om att n är sammansatt och algoritmen avslutas. Annars sätter vi $i = i + 1$, och går vidare till Steg 4.

Steg 4: Om $i = k + 1$ är n ett primtal med sannolikhet minst $1 - 2^{-k}$ (enligt Proposition 4.2.3). Annars går vi tillbaka till Steg 2.

Komplexiteten för att räkna ut vänsterledet är densamma som för högerledet, $\mathcal{O}(\log^3(n))$ enligt Koblitz [2, s.129]. I värsta fall räknar vi ut ekvationen k gånger. Den totala komplexiteten för algoritmen blir $\mathcal{O}(k \times \log^3(n))$.

4.3 Miller-Rabins test

Låt oss införa lite teori.

Proposition 4.3.1 (s.126 i [3])**.** *Låt $p > 2$ vara ett primtal och skriv $p = 2^k q + 1$ med q udda. Låt a vara ett tal ej delbart med p. Då gäller ett av följande två fall*

$$(i)\ a^q \equiv 1 \pmod{p}$$
$$(ii)\ a^{2^i q} \equiv -1 \pmod{p}, \quad \textit{något } i \textit{ sådant att } 0 \leq i < k.$$

Likt tidigare använder vi propositionen för att skapa definitionen av ett vittne, och utvecklar *Miller-Rabintestet.*

Definition 4.3.2. Låt n vara ett udda tal med $n-1 = 2^k q$ med q udda. Vi säger att ett heltal a där $\text{sgd}(a,n) = 1$ är ett *Miller-Rabinvittne* till n ifall följande villkor är uppfyllda

$$(i)\ a^q \not\equiv 1 \pmod{n} \tag{4.3.1}$$

$$(ii)\ a^{2^i} \not\equiv -1 \pmod{n}, \quad \forall\, i \in \{0,1,...,k-1\}. \tag{4.3.2}$$

Till skillnad från Solovay-Strassens test använder vi inte Jacobisymbolen som i Definition 4.2.2 utan undersöker huruvida kongruenserna (4.3.1) och (4.3.2) gäller.

Nedan ger vi en proposition som vi kan använda för att avgöra sannolikheten för att ett tal n är ett primtal om Miller-Rabin ger troligtvis primtal.

Proposition 4.3.3 (sid. 127 [3])**.** *Låt n vara ett sammansatt tal. Då är minst 75% av talen a mellan 1 och $n-1$ Miller-Rabin vittnen till n.*

4.3.1 Algoritmen

Låt n vara ett udda heltal.

Steg 1: Välj $i > 0$ för att ange hur många varv algoritmen skall köras innan avslut.
Sätt $q = (n-1)/2^d$ med $d \in \mathbb{N}$ sådant att q är ett udda heltal. Vi har $n-1 = q2^d$.

Steg 2: Sätt $a < n$ till ett slumpmässigt valt positivt heltal större än 1, $b = a^{n-1} \pmod{n}$ och $k = 0$.
Gå till steg 3.
Om $b = 1$ är a per definition inget vittne. Gå till Steg 4.

Steg 3: Här är $a^q \not\equiv 1 \pmod{n}$ så villkor (i) i Proposition 4.3.1 är inte uppfyllt.
Så länge $b \not\equiv \pm 1 \pmod{n}$ och $k \leq d-2$, tilldela b värdet $b^2 \pmod{n}$ och öka k med ett.
Om $b \neq n-1$ det vill säga $b \not\equiv -1 \pmod{n}$ är inte heller det andra villkoret i Proposition 4.3.1 uppfyllt varför vi avslutar och returnerar sammansatt.

Steg 4: Minska i med 1. Om $i > 0$ gå till Steg 2. Annars avsluta och returnera troligtvis primtal.

Algoritmen har en komplexitet runt $\mathcal{O}(\log^3(n))$. Se [12, s.415].

4.4 Resultat

Vi har testat algoritmerna för olika heltal n, se Appendix C. Vi körde Solovay-Strassen 200 runor, och Miller-Rabin 100 rundor, vilket ger en sannolikhet för primtal på $1 - 2^{-200}$.

n	Direkt metod	Solovay-Strassen	Miller-Rabin (C#)
Primtal av storlek $\approx 10^{19}$	24,479s.	0,006s.	0,003s.
Primtal av storlek $\approx 10^{173}$	> 15m. algoritmen avbröts.	0,026s.	0,786s.
Sammansatt tal med faktor 5 av storlek $\approx 10^{174}$	0,005s.	0,005s.	0,003s.

5 Faktoriseringsalgoritmer

I det här kapitlet studerar, implementerar och testar vi några utvalda algoritmer för att faktorisera ett stort tal $N = pq$ där p och q är udda primtal. Med ett stort N är det svårare att dekryptera meddelandet utan information om faktoriseringen till N. Med den privata informationen p och q kan vi enkelt utföra dekrypteringen. Därifrån kommer intresset att studera faktoriseringsalgoritmer. För att jämföra de olika faktoriseringsalgoritmerna förklarar vi deras komplexitet samt tiden det tar att faktorisera olika storlekar av N med två valda primtal.

5.1 Direkt Metod för faktorisering

I kapitlet Primtalstest beskrevs en direkt metod (se avsnitt 4.1) som enkelt anpassas till en algoritm för faktorisering.

Steg 1: Antag att N är ett positivt udda tal, välj $i = 3$.

Steg 2: Beräkna $\frac{N}{i}$. Om resultatet är ett heltal avslutas algoritmen, N har två faktorer i och $\frac{N}{i}$. Får vi inte ett heltal går vi vidare till Steg 3.

Steg 3: Om $i = \lfloor\sqrt{N}\rfloor$ är N är ett primtal och kan inte faktoriseras, algoritmen avslutas. Annars välj $i = i + 2$ och gå tillbaka till Steg 2.

Algoritmen tar i värsta fall $\sqrt{N}$ steg och utför lika många divisioner. Vi har därmed en komplexitet på $\mathcal{O}(\sqrt{N})$.

5.2 Pollards rho-algoritm

En version av Pollards rho-algoritm är när vi använder två talföljder enligt algoritmen nedan. Denna version av Pollards rho-algoritm är baserad på teorin från [3, s.234-243].

Steg 1: Välj en funktion $f : \mathbb{Z}_N \to \mathbb{Z}_N$, ett startvärde $x_0 = y_0 \in \mathbb{Z}_N$ och sätt $i = 0$.

Steg 2: Beräkna $x_{i+1} = f(x_i)$ och $y_{i+1} = f(f(y_i))$.

Steg 3: Beräkna $\text{sgd}(x_{i+1} - y_{i+1}, N)$.

Om $\text{sgd}(x_{i+1} - y_{i+1}, N) = 1$ eller $\text{sgd}(x_{i+1} - y_{i+1}, N) = N$ öka i med 1 och fortsätt från Steg 2. Annars får vi att $\text{sgd}(x_{i+1} - y_{i+1}, N) = p$ är en faktor till N.

Vi börjar med att välja en funktion $f : \mathbb{Z}_N \to \mathbb{Z}_N$ som mixar, se [2, s.142], de N elementen i mängden $\mathbb{Z}_N$. Med mixar menar vi att $f(x)$ kan anses vara relativt slumpmässig för olika x men eftersom funktionen är deterministisk så antar $f(x)$ alltid samma värde för samma x. I vår implementation valde vi $f(x) = x^2 + 1$. I Steg 2 beräknar vi våra två följder. Eftersom vi applicerar funktionen f två gånger på den ena följden har vi att $y_i = x_{2i}$, därmed är följden y_i dubbelt så snabb som följden x_i. Efter ett antal iterationer T, kommer följden y_i att ta sig in i en loop av längd M. Vi fortsätter att uppdatera följderna x_i och y_i tills vi finner ett i sådant att $y_i = x_i$, alltså att $x_{2i} = x_i$. En fördel med den här algoritmen är att vi endast behöver spara det senaste värdet av följderna x_i och y_i vilket kan vara en stor fördel vid faktorisering av större tal. Nu kommer en sats som säger något om komplexiteten för Pollards rho-algoritm. För ett bevis till att $y_i = x_i$ för något $1 \leq i < T + M$, se Appendix A.11.

Sats 5.2.1. *Låt $f : \mathbb{Z}_N \to \mathbb{Z}_N$ och låt $x \in \mathbb{Z}_N$ vara ett startelement. Låt vidare T och M vara som i texten ovan. Om f mixar elementen i $\mathbb{Z}_N$ tillräckligt bra så är det förväntade antalet iterationer*

$$\mathbb{E}(T + M) \approx 1.2533 \cdot \sqrt{N} \tag{5.2.1}$$

Bevis. För det här beviset behöver vi resultatet att integralen

$$I = \int_0^\infty t^2 e^{-\frac{t^2}{2}}\, dt = \sqrt{\frac{\pi}{2}}. \tag{5.2.2}$$

För en fullständig lösning se Appendix A.10.

Vi vill räkna ut sannolikheten att alla värden $x_0, x_1, ..., x_{k-1}$ är distinkta. Vi antar att f genererar alla x_i slumpmässigt ur $\mathbb{Z}_N$ med återplacering, alltså att f kan generera samma x_i igen. Vi får för $k \approx \sqrt{N}$ att

$$\begin{aligned} P(x_0, x_1, ..., x_{k-1} \text{ är olika }) &= \prod_{i=1}^{k-1} P(x_i \neq x_j \text{ för } 0 \leq j < i | x_0, x_1, ..., x_{k-1} \text{ är olika}) \\ &= \prod_{i=1}^{k-1} \frac{N-i}{N} = \prod_{i=1}^{k-1} \left(1 - \frac{i}{N}\right) =: \bigstar. \end{aligned}$$

Vi kommer nu att använda det faktum att $1-t \approx e^{-t}$ för små värden på t. Vi tar $t = \frac{i}{N}$ i $\bigstar$ vilket kan rättfärdigas genom att N är ett stort tal att faktorisera så $\frac{i}{N}$ är litet för $1 \leq i < k$. Vi får att

$$\bigstar \approx \prod_{i=1}^{k-1} e^{-\frac{i}{N}} = e^{-(1+2+...+(k-1))/N} = e^{-(\frac{k^2-k}{2})\cdot\frac{1}{N}} \approx e^{-\frac{k^2}{2N}}$$

eftersom $\frac{k^2-k}{2} \approx \frac{k^2}{2}$ när k är stort. Nu har vi sannolikheten att de $k-1$ valda elementen bland N möjliga är distinkta. Vi vet att sannolikheten att x_k är en match, alltså lika med, till något av de första $k-1$ är $\frac{k}{N}$. Sannolikehten att x_k är den första matchningen är då

$$P(x_k \text{ är den första matchningen }) = P(x_k \text{ är en match och } x_0, x_1, ..., x_{k-1} \text{ är distinkta})$$
$$= P(x_k \text{ är en match } | x_0, x_1, ..., x_{k-1} \text{ är distinkta}) \cdot P(x_0, x_1, ..., x_{k-1} \text{ är distinkta}) \approx \frac{k}{N} \cdot e^{-\frac{k^2}{2N}}.$$

Väntevärdet för antalet iterationer innan vi hittar den första matchningen är då

$$\mathbb{E}(\text{första matchningen}) = \sum_{k\geq 1} k \cdot \frac{k}{N} \cdot e^{-\frac{k^2}{2N}} = \sum_{k\geq 1} \frac{k^2}{N} \cdot e^{-\frac{k^2}{2N}}. \tag{5.2.3}$$

Vi behöver nu ett lemma för att uppskatta (5.2.3).

Lemma 5.2.1.1 ([3], s.238)**.** *Låt $F(t)$ vara en reell funktion med kontinuerlig derivata och med egenskapen att $\int_0^\infty F(t)dt$ konvergerar. För stora heltal n har vi då att*

$$\sum_{k=1}^{\infty} F(\frac{k}{n}) \approx n \cdot \int_0^\infty F(t)dt. \tag{5.2.4}$$

Med (5.2.4) kan vi fortsätta beräkningen av (5.2.3). Vi låter $F(t) = t^2 e^{-\frac{t^2}{2}}$ och får att

$$\sum_{k\geq 1} \frac{k^2}{N} \cdot e^{-\frac{k^2}{2N}} = \sum_{k\geq 1} F(\frac{k}{\sqrt{N}}) \approx \sqrt{N} \cdot \int_0^\infty t^2 e^{-\frac{t^2}{2}} dt = \sqrt{\frac{\pi}{2}} \cdot \sqrt{N} \approx 1.2533 \cdot \sqrt{N}$$

vilket skulle visas. □

Det Sats 5.2.1 säger är att komplexiteten för den här versionen av Pollards rho-algoritm är approximativt $\mathcal{O}(\sqrt{N})$ om N är stort. I praktiken visar sig $\mathcal{O}(\sqrt{N})$ vara en övre gräns för komplexiteten och under antagandet att f fördelar x_i pseudoslumpmässigt så blir den förväntade komplexiteten $\mathcal{O}(\sqrt[4]{N})$ [13]. I Appendix A.12 finns en alternativ version av Pollards rho-algoritm. Där finns även en analys av komplexiteten för den alternativa versionen av Pollards rho-algoritm under antagandet att f fördelar x_i pseudoslumpmässigt, vilket slutligen ger komplexiteten $\mathcal{O}(\sqrt[4]{N}\log^3(N))$ med hög sannolikhet.

5.3 Fermats algoritm

Idén bakom Fermats algoritm visas av följande proposition.

Proposition 5.3.1. *Låt $a^2 - b^2 = kN$ och $N = pq$ där $a, b, k, p, q \in \mathbb{N}$ samt $c = sgd(a+b, N)$ och $d = sgd(a-b, N)$. Låt även p och q vara primtal samt $p \neq q$. Om nu c och d varken är 1 eller N så är $N = cd$.*

För bevis se Appendix A.7. Denna proposition ger ett effektivt sätt att hitta faktoriseringen av N givet att man vet $a, b \in \mathbb{N}$ sådana att $kN = a^2 - b^2$. Vi skriver om ekvationen och får att $a^2 \equiv b^2 \pmod{N}$. Både Fermats algoritm och Dixons algoritm, se avsnitt 5.4, bygger på att finna tal a och b som uppfyller denna proposition, men deras metoder för att finna dessa tal är olika. Fermats algoritm kan förklaras som:

Steg 1: Börja med $i = 1$.

Steg 2: $x_i = \lfloor \sqrt{N} \rfloor + i$

Steg 3: Om $\exists b : x_i^2 \equiv b^2 \pmod{N}$ och $0 \leq b < N$ fortsätt till Steg 4. Annars öka i med 1 och gå tillbaka till Steg 2.

Steg 4: Om $\text{sgd}(a+b, N) \neq 1$ och $\text{sgd}(a+b, N) \neq N$ så är $\text{sgd}(a+b, N)$ en faktor av N och vi är klara. Annars öka i med 1 och gå till Steg 2.

Komplexiteten för Fermats algoritm, med en mindre modifikation, är $\mathcal{O}(N^{\frac{1}{3}})$ [14]. En nackdel med denna algoritm är att vi testar varje x_i separat istället för att testa alla kombinationer av olika x_i, vilket Dixons algoritm gör.

5.4 Dixons algoritm

Både Dixons algoritm och Fermats algoritm bygger på Proposition 5.3.1. Skillnaden är att Dixons algoritm testar om kombinationer av tal kan multipliceras till en jämn kvadrat. Vi behöver följande definition.

Definition 5.4.1. Ett heltal kallas *B-glatt* om dess primtalsfaktorer är mindre än eller lika med B (se [3, s.146]).

Dixons algoritm för att faktorisera $N = pq$, där p och q är primtal, kan beskrivas med följande 4 steg.

Steg 1: Välj två tal $k, r \in \mathbb{N}$.

Steg 2: Bilda mängderna $A = \{a_1, a_2, \ldots, a_r\}$ och $C = \{c_1, c_2, \ldots, c_r\}$ sådana att $a_i, c_i \in \mathbb{N}$, $a_i^2 \equiv c_i \pmod{N}$, alla c_i är k-glatta, samt $c_i = \prod_j p_j^{\gamma_{i,j}}$ där $\gamma_{i,j} \in \mathbb{N}$.

Steg 3: Välj $(u_0, u_1, \ldots, u_r) \in \{0,1\}^r$ sådan att $\exists \lambda_i : c = \prod_i c_i^{u_i} = \prod_j p_j^{2\lambda_j}$, det vill säga en produkt av c_i vars primtalsfaktorisering har jämna exponenter.

Steg 4: Låt $a = \prod_i a_i^{u_i}$, det vill säga a är produkten av de a_i som motsvarar de c_i vilkas produkt utgör c. Om nu $a \not\equiv c \pmod{N}$ och $a \not\equiv -c \pmod{N}$ så är $\text{sgd}(a + \sqrt{c}, N)$ och $\text{sgd}(a - \sqrt{c}, N)$ de två faktorerna i N.

I Steg 2 låter vi $c_i \equiv a_i^2 \pmod{N}$, $0 \leq c_i < N$ för ett givet a_i. Vi väljer a_i slumpmässigt från an likformigfördelning mellan 1 och N. Om a_i ger ett k-glatt c_i lägger vi till a_i i A och c_i i C. Annars väljer vi ett nytt a_i ur den likformiga fördelningen. Vi fortsätter tills $|A| = r$.

I Steg 3 vill vi hitta en delmängd till C sådan att deras produkt är en jämn kvadrat. Låt $u_i \in \{0,1\}$ vara $u_i = 1$ om c_i ska vara med i produkten och $u_i = 0$ annars. Vi vill nu lösa $c = \prod_i c_i^{u_i} = \prod_j p_j^{2\lambda_j}$. Notera nu att $\gamma_{i,j}$ är exponenten hos det j:te primtalet i primtalsfaktoriseringen av c_i samt att p_j är samma tal för alla c_i. Nu kan vi skriva om ekvationen som

$$\prod_j p_j^{\sum_i \gamma_{i,j} u_i} = \prod_j p_j^{2\lambda_j}.$$

Vi kan nu lösa problemet igenom att finna u_i sådana att

$$\sum_i \gamma_{i,j} u_i \equiv 0 \pmod{2} \quad \forall j. \tag{5.4.1}$$

För att lösa ekvationerna skapar vi en matris M vars element på plats (j, i) är den ekvivalensklass i $\mathbb{F}_2$ som korresponderar med $\gamma_{i,j}$. Denna matris rader korresponderar mot exponenterna för ett specifikt primtal hos alla c_i och kolumnerna korresponderar med exponenterna för varje primtal hos ett specifikt c_i. Nu gauss-eliminerar vi matrisen och hittar nollrummet, vilket spänner upp alla lösningar till ekvationen. Vi minns från algebran att matrisräkningen fungerar eftersom $\mathbb{F}_2$ är en kropp. Vi får även att det existerar u_i som löser ekvation (5.4.1) pågrund av vårt val av r, (se förklaringen av Steg 1).

Vi använder ett val av u_i som löser ekvation (5.4.1) för att lösa Steg 4. Först beräknar vi $a = \prod_i a_i^{u_i}$. Om sedan $a \equiv c \pmod{N}$ eller $a \equiv -c \pmod{N}$ så har vi inte hittat en faktorisering

och vi väljer en annan uppsättning u_i som löser ekvation (5.4.1), annars är $\text{sgd}(a+\sqrt{c}, N)$ och $\text{sgd}(a-\sqrt{c}, N)$ faktorerna enligt Sats 5.3.1. Om vi testat alla lösningar vi hittade till ekvation (5.4.1) och inte funnit en faktorisering så väljer vi ett större r, slumpar fram fler a_i tills vi åter har att $|A| = r$ och fortsätter från Steg 3. Notera att vi kan beräkna $\sqrt{c}$ enligt $\sqrt{c} = \prod_j p_j^{\sum_i \gamma_{i,j} u_i/2}$.

I Steg 1 ska vi välja passande värden på r och k. Låt d vara antalet primtal mindre än k. Vi väljer $r = d+2$ och får en $d \times (d+2)$ matris till ekvation (5.4.1) som är lösbar enligt linjär algebra. Valet av k bestämmer nu storleken på matrisen, storleken på A men också hur stor sannolikheten är att ett givet a_i korresponderar med ett c-glatt tal. För stora k kan det krävas mycket fysiskt minne att spara matrisen eftersom storleken på matrisen växer med d^2 (även kallat minneskomplexitet $\mathcal{O}\left(d^2\right)$).

5.4.1 Komplexitet

Steg 2, 3 och 4 i Dixons algoritm har sin egen komplexitet. Men den totala komplexiteten blir $e^{C\sqrt{\log(N)\log(\log(N))}}$ [2, s.152]. För en enklare härledning till den här komplexiteten se Appendix A.14.

5.5 Kedjebråksmetoden

I början av 1930-talet introducerade de två matematikerna D.H. Lehmer och R.E. Powers kedjebråksmetoden som med den tidens beräkningskraft hade begränsat användningsområde. Anledningen till det var de mycket omfattande beräkningarna som behövdes göras för hand.

5.5.1 Enkla kedjebråk

Låt $C \in \mathbb{R}^+$ och a_0 vara heltalsdelen till C. Om $a_0 \neq C$ kan vi skriva

$$C = a_0 + b_0 = a_0 + 1/b_0^{-1},\ 0 < b_0 < 1. \tag{5.5.1}$$

Eftersom $b_0^{-1} > 1$, låter vi a_1 vara heltalsdelen till b_0^{-1} och om $a_1 \neq b_0^{-1}$ sätter vi

$$b_0^{-1} = a_1 + b_1 = a_1 + 1/b_1^{-1},\ 0 < b_1 < 1. \tag{5.5.2}$$

Vi har således

$$C = a_0 + \frac{1}{a_1 + \frac{1}{b_1^{-1}}},\ \text{där } b_1^{-1} > 1. \tag{5.5.3}$$

Vi kan nu upprepa ekvation (5.5.2) och ersätta b_1^{-1} med $a_2 + 1/b_2^{-1}$ och så vidare.

Om $C = a_0$ är C ett heltal och kedjebråket framställs endast med C. Om det för något heltal n gäller att $a_n = b_{n-1}^{-1}$ avbryts utvecklingen med a_n som den sista nämnaren. Vi kallar högerledet i ekvation (5.5.3) för ett enkelt kedjebråk och inför notationen

$$C = [a_0, a_1, b_1^{-1}] \text{ alternativt } C = [a_0, a_1, a_2, ...],$$

och för ändliga kedjebråk skriver vi

$$C = [a_0, a_1, ..., a_n]. \tag{5.5.4}$$

Definition 5.5.1. Låt $R = [a_0, a_1, a_2...]$ vara ett oändligt kedjebråk. Sätt $K_n = [a_0, a_1, a_2, ..., a_n]$. Vi säger att K_n är den $n : te$ *konvergenten* i kedjebråksutvecklingen av R.

Eftersom K_n i definitionen ovan har en ändlig kedjebråksutveckling är K_n ett rationellt tal och vi kan skriva

$$K_n = A_n/B_n, \quad A_n, B_n \in \mathbb{Z}.$$

A_n och B_n beräknas rekursivt genom

$$\frac{A_0}{B_0} = \frac{a_0}{1}, \quad \frac{A_1}{B_1} = \frac{a_0 a_1 + 1}{a_1}, \quad \frac{A_n}{B_n} = \frac{a_n A_{n-1} + A_{n-2}}{a_n B_{n-1} + B_{n-2}}, \quad n \geq 2 \tag{5.5.5}$$

där $\text{sgd}(A_n, B_n) = 1$ för alla n (se[2, s.155]).

5.5.2 Idé

Antag att vi vill faktorisera $N \in \mathbb{N}$. Låt A_n/B_n vara den n:te konvergenten i kedjebråksutvecklingen av $\sqrt{N}$. Faktum är att A_n/B_n ger snabbt en mycket god approximation till $\sqrt{N}$ varför vi kan skriva (se [13])

$$A_{n-1}/B_{n-1} \approx \sqrt{N} \implies A_{n-1}^2/B_{n-1}^2 \approx N \implies A_{n-1}^2 - NB_{n-1}^2 \approx 0.$$

Definiera

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

vilket medför att

$$(-1)^n Q_n \equiv A_{n-1}^2 \pmod{N}.$$

Att $Q_n \approx 0$ betyder här att $Q_n \ll N$. I praktiken gäller att $|Q_n| \leq 2\sqrt{N}$ för alla n (se s. 185 i [15]). Fördelen med att Q_n är förhållandevis små är att vi kan fakorisera dessa genom att testa delbarheten med element ur en förvald mängd av primtal, en så kallad primtalsbas, för att i sin tur hitta olika Q_n sådana att

$$Q := \prod_{n \in \mu} Q_n = X^2, \; X \in \mathbb{Z}, \text{ för något } \mu \subset \mathbb{N}. \tag{5.5.6}$$

Detta inträffar då alla primtalsfaktorer till Q har jämn exponent.

Om vi definierar

$$Y = \prod_n A_{n-1} \pmod{N}$$

ser vi att

$$X^2 \equiv Y^2 \pmod{N}. \tag{5.5.7}$$

Ekvationen (5.5.7) medför att för något heltal j har vi

$$X^2 - Y^2 = (X+Y)(X-Y) = jN$$

vilket ger oss goda chanser till att $X - Y$ och N har gemensamma äkta delare varför vi beräknar

$$D = \text{sgd}(X - Y, N).$$

Visar det sig att D är en äkta delare till N är vi klara ty i det här projektet intresserar vi oss endast av N sådana att dess faktorisering består av två olika primtal. Annars går vi tillbaka för att beräknar fler Q_n och fortsätter så.

5.5.3 Algoritmen

Låt $N > 2$ vara ett udda heltal.

Steg 1: I detta steget skapar vi en mängd $M = \{(Q_n, A_{n-1}) \mid 0 \leq n \leq S\}$ där S är ett förvalt heltal. S skall väljas så stort att det finns $F_j \subset M$ sådana att $\prod_{(Q_n, A_{n-1}) \in F_j} Q_n = X^2$ där $X \in \mathbb{Z}$. För att hitta paren (Q_n, A_{n-1}) använder vi följande algoritm [15] som utvecklar kedjebråket för $\sqrt{kN}$, beräknar alla Q_n och A_{n-1}. Multiplikatorn k använder vi om faktoriseringen misslyckas och sätter den i början till 1.

1: Sätt $A_{-2} = 0$, $A_{-1} = 1$, $Q_{-1} = kN$, $r_{-1} = g = \lfloor \sqrt{kN} \rfloor$, $P_0 = 0$, $Q_0 = 1$.

2: $q_n = \lfloor (g + P_n)/Q_n \rfloor$.

3: $r_n = g + P_n - q_n Q_n$.

4: $A_n = q_n A_{n-1} + A_{n-2} \pmod{N}$.

5: $P_{n+1} = g - r_n$.

6: $Q_{n+1} = Q_{n-1} + q_n(r_n - r_{n-1})$.

7: $n < s \implies$ Tilldela n värdet $n+1$ och gå till 2. Annars avsluta.

Nu har vi M.

Steg 2: Här skapar vi mängderna F_j som vi nämnde i Steg 1.
Undersök vilka element i M som har egenskapen (5.5.6) och bilda mängderna F_j.
I [15] beskrivs en mycket effektiv metod för att göra detta där man utnyttjar egenskapen att ett heltal C är en kvadrat *omm* varje primtalsfaktor till C har en jämn exponent. I denna framställning ger vi en, minst sagt, komprimerad beskrivning av tillvägagångsättet.

Faktorisera alla nya Q_n vi erhållit från Steg 1 fullständigt. Om Q_0 ingår bland dessa börja med att skapa en radvektor

$$P = (-1, p_1, p_2, ..., p_i) \tag{5.5.8}$$

där p_j, $1 \leq j \leq i$ är alla distinkta primtal som uppkommit vid faktoriseringen. För varje $Q_n : n \geq 1$ skapa en exponentvektor

$$E_{n,\rho} = \begin{cases} 1 & \text{om } (\rho = 0 \text{ och n udda}) \text{ eller } (\rho \geq 1 \text{ och } p_\rho | Q_n \text{ med udda exponent}), \\ 0 & \text{om } (\rho = 0 \text{ och n jämn}) \text{ eller } (\rho \geq 1 \text{ och } p_\rho | Q_n \text{ med jämn exponent}). \end{cases} \tag{5.5.9}$$

Låt E vara en matris vars (i, j):te element är $E_{i,j}$ och bilda en $i \times i$ identitetsmatris H. Låt E_m och H_m vara de m:te radvektorerna i E resp. H.

Radreducera E och H enligt följande schema[15]:

1: Sätt $\delta = i$

2: För kolumn δ, låt $\Delta(\delta) = \{n : E_{n,\delta} = 1,\ E_{n,\gamma} = 0,\ \gamma > \delta\}$.
Om $\Delta(\delta) = \emptyset$, hoppa till Steg 4. Annars sätt $d = \inf \Delta(\delta)$.

3: $\forall n \in \Delta(\delta) \backslash \{d\}$ tilldela E_n värdet $E_n + E_d \pmod 2$ och H_n värdet $H_n + H_d \pmod 2$

4: Minska δ med 1. Om $\delta \geq 0$ gå till 2. Annars avsluta.

Sätt

$$L = \{k : \sum_{\alpha=0}^{i} E_{k,\alpha} = 0\} \tag{5.5.10}$$

och låt

$$\forall k \in L,\ F_k = \{(Q_n, A_{n-1}) : H_{k,n} = 1\} \tag{5.5.11}$$

Steg 3: $\forall j \in L$ beräkna

$$X = \sqrt{\prod_{(Q_n, A_{n-1}) \in F_j} Q_n} \pmod N \quad \text{och} \quad Y = \prod_{(Q_n, A_{n-1}) \in F_j} A_{n-1} \pmod N . \tag{5.5.12}$$

Beräkna $D = \text{sgd}(X - Y, N)$. Om D inte är en äkta delare till N för något F_j, kan vi antingen utöka M genom lägga något heltal till S och gå till Steg 1.7, eller välja ett annat värde på k och köra hela algoritmen igen.

5.6 Kvadratiskt såll

Det kvadratiska sållet är en optimering av Dixons algoritm, först beskriven av Carl Pomerance 1982 [16, s.130]. Det kvadratiska sållet är fortfarande den snabbaste faktoriseringsalgoritmen för stora allmänna tal upp till ca $N = 2^{350}$, enligt Hoffstein m.fl. [3, s. 152]. Namnet kvadratiskt såll kommer från kombinationen av att sålla värden på en följd y_i för att hitta glatta tal (se Definition 5.4.1), och den valda formen på talföljden y_i, skapad med hjälp av ett kvadratiskt polynom.

5.6.1 Idé

Idén är som tidigare baserad på Fermats faktoriseringsalgoritm: hitta x och y sådana att $N = x^2 - y^2 = (x + y)(x - y)$. Detta ger en faktorisering av N, se Appendix A.7. Denna faktorisering kan vara trivial vilket diskuteras senare. Vi skapar först en talföljd $x_i = \lfloor \sqrt{N} \rfloor + i,\ i \in \mathbb{N}$. Med hjälp av polynomet $y = x^2 - N$ skapas nu en motsvarande talföljd av element $y_i = x_i^2 - N$. Målet är likt i avsnitt 5.4 och 5.5 att hitta en mängd av tal y_i sådana att $y = \prod_i y_i \pmod N$ är en jämn

kvadrat. Detta gör man genom att sålla sekvensen y_i med primtalen i den utvalda *primtalsbasen*. Anledningen till att y_i väljes med hjälp av ett kvadratiskt polynom är för att man kan minska på primtalsbasen. De primtal som är relevanta för sållet är alla p_j som delar polynomet $x^2 - N$, för något $x \in \mathbb{N}$, ekvivalent uttryckt om $\left(\frac{N}{p}\right) = 1$ (se [17, s. 265]).

5.6.2 Algoritm

Vi beskriver kortfattat en algoritm som implementerar en grundläggande version av det kvadratiska sållet, baserad på Pomerances beskrivning (se [17, s. 266]).

Steg 1: Skapa en primtalsbas P, det vill säga en mängd bestående av alla primtal p_j mindre än en övre gräns B, sådana att $\left(\frac{N}{p_j}\right) = 1$. Skapa talföljden, $x_i = \lfloor\sqrt{N}\rfloor + i,\ i \in \mathbb{N}$.

Steg 2: Sålla talföljden $y_i = x_i^2 - N$, för tal sådana att alla deras primtalsfaktorer är i P.

Steg 3: Beräkna primtalsfaktoriseringen av varje $y_i = \prod_{j=1}^{k} p_j^{\gamma_{ij}},\ \gamma_{ij} \in \mathbb{N}$. Ställ upp en matris $A = \{\gamma_{ij}\} \pmod 2$, där varje rad representerar ett y_i och varje kolonn ett primtal p_j i primtalsbasen P. Hitta med hjälp av linjär algebra en linjärkombination av rader som summerar till 0-raden, vilket ger en kombination av y_i sådana att $y = \prod_{i=1}^{n} y_i$ är en jämn kvadrat.

Steg 4: Välj ut alla värden y_i som i Steg 3 summerade till en 0-rad, och välj ut motsvarande x_i. Beräkna $X = \prod_{i=1}^{n} x_i \pmod N$, $Y = \sqrt{\prod_{i=1}^{n} y_i} \pmod N$. Slutligen, beräkna $D = \text{sgd}(X - Y, N)$ för att få en faktor till N.

I Steg 1 är valet av B viktigt, ett litet värde innebär att färre tal y_i måste sållas fram, men det blir svårare då de måste bestå av färre primtal. Pomerance [17, s. 265] ger det optimala värdet som:

$$B = e^{\frac{1}{2}\sqrt{\log N \log\log N}}. \tag{5.6.1}$$

En annan aspekt att beakta är det begränsade minnet på datorn som kör algoritmen. Att gå under det optimala värdet i ekvation (5.6.1) leder till längre beräkningstid men mindre minnesanvändning. I vår implementering har vi valt att begränsa oss till primtal under $\approx 4 \times 10^6$. Vi har även en minimigräns då det optimala värdet ovan inte fungerar bra för att faktorisera små tal.

I Steg 2 bildas en lång talföljd $y_i = x_i^2 - N$, vi kallar följden för F. Primtalen p_j i primtalsbasen P är valda just så att ekvationen

$$x^2 - N \equiv 0 \pmod{p_j} \iff x^2 \equiv N \pmod{p_j} \tag{5.6.2}$$

går att lösa. Ekvation (5.6.2) har då, för varje $p_i > 2$, två lösningar. Slutligen anmärker vi att

$$\begin{aligned} p_j | \big((x + kp_j)^2 - N\big) \ &\Leftrightarrow\ (x+kp)^2 - N = x^2 + 2kp + k^2p^2 \equiv \\ &\equiv x^2 - N \equiv 0 \pmod{p_j} \ \Leftrightarrow\ p_j | (x^2 - N),\ \ \forall k \in \mathbb{Z}. \end{aligned}$$

Detta tillsammans med ekvation (5.6.2) låter oss konstruera ett såll. För varje primtal $p_j \in P$, hitta de två första $y_{j0}, y_{j1} \in F$ som är delbara med p_j och dela bort alla deras faktorer p_j. Efter det hoppar vi p_j steg i följden F och delar bort alla faktorer p_j ur följande två tal y_{j0+p_j}, y_{j1+p_j}. Upprepa tills vi har gått igenom listan F. De ursprungliga tal i listan F som slutligen blir 1 är de som passar in i vår faktorbas.

Det är här den stora tidsbesparingen uppstår jämfört med Fermats & Dixons faktoriseringsalgoritmer. Används en direkt metod (se 5.1) för att hitta tal som passar in i primtalsbasen P skulle det kräva k steg, där k är antalet primtal i P, och ha en komplexitet $\mathcal{O}(k)$, per tal. Med ett såll som beskrivet ovan går komplexiteten ner till $\mathcal{O}(\log\log B)$ per tal [17, s. 264].

I Steg 3 utför vi Gauss-elimination på en matris reducerad modulo 2, detta beskrivs detaljerat i Kapitel 5.4. Att ta kvadratroten ur Y i Steg 4 beskrivs även detta i Kapitel 5.4.

Det finns fyra möjliga värden på D varav endast två är intressanta. Om vi i Steg 2 väljer att hitta $k + 1$ värden y_i är det $\approx 50\%$ risk att vi endast hittar en trivial faktor. Det är vanligt att man väljer ut fler värden y_i i Steg 2 till en kostnad av mer beräkningstid. Varje ytterliggare värde skapar en till nollrad i Steg 3 och väljer vi då $k + n$ styck y_i har vi n möjliga kombinationer som ger en jämn kvadrat. Sannolikheten att alla ger triviala faktorer är då $\left(\frac{1}{2}\right)^n$ [17, s. 258].

Komplexiteten för det kvadratiska sållet är föreslagen som $\approx \mathcal{O}\left(e^{\sqrt{\log N \log\log N}}\right)$ enligt Pomerance ([8, s. 1478][17, s.265]). Den föreslagna komplexiteten är *subexponentiell* och därmed lägre än för alla faktoriseringsalgoritmer som hittills har studerats i detta arbete.

5.7 Resultat

Vi har testat algoritmerna för olika värden N på en gemensam testbädd, se Appendix C.

$N = pq$	Direkt metod	Pollard-Rho	Fermat
Storlek $\approx 10^6$	0,004s.	0,009s.	0,006s.
Storlek $\approx 10^{11}$	0,012s.	0,006s.	0,006s.
Storlek $\approx 10^{24}$	13m. 53s.	0,167s.	$>$ 15m. Avbröts.
Storlek $\approx 10^{30}$	$>$ 15m. Avbröts.	6s.	$>$ 15m. Avbröts.
Storlek $\approx 10^{18}$, med liten faktor.	0,006s.	0,006s.	$>$ 15m. Avbröts
Storlek $\approx 10^{578}$.	$>$ 15m. Avbröts.	$>$ 15m. Avbröts.	$>$ 15m. Avbröts

$N = pq$	Dixon	Kedjebråk (C#)	Kvadratiskt Såll
Storlek $\approx 10^6$	0,015s.	0,298s.	0,151s.
Storlek $\approx 10^{11}$	0,189s.	0,316s.	0,193s.
Storlek $\approx 10^{24}$	11m. 35s.	1,444s.	15,207s.
Storlek $\approx 10^{30}$	$>$ 15m. Avbröts.	$>$ 15m. Avbröts	1m. 47s.
Storlek $\approx 10^{18}$ med liten faktor.	7,740s.	0,171s.	2,926s.
Storlek $\approx 10^{578}$.	$>$ 15m. Avbröts.	$>$ 15m. Avbröts.	$>$ 15m. Avbröts

6 En kvantalgoritm

Hittills har vi fokuserat på så kallade klassiska algoritmer, vilket innebär att de är genomförbara med operationer som en klassisk dator tillåter. I detta kapitlet kommer vi diskutera en naturlig breddning av dessa, nämligen kvantalgoritmer. Detta är alltså motsvarigheten som endast kan utföras på en kvantdator. Mer specifikt ska vi fokusera på Shors algoritm, en kvantalgoritm som visar sig ha lägre komplexitet än någon känd klassisk algoritm.

Det som skiljer kvantdatorer från klassiska datorer är att de baseras på kvantfysik, vilket tillåter fenomen som *superpositioner* av olika tillstånd och *sammanflätningar* av dessa tillstånd med mera. Målet med detta kapitlet är att ge en kort introduktion till kvantfysik för att beskriva vissa verktyg inom området. Vi kommer sedan använda verktygen för att slutligen redovisa för hur Shors algoritm fungerar, en faktoriseringslgoritm som visar sig vara mycket snabbare än alla hittills kända klassiska algoritmer. I just detta kapitlet kommer vi huvudsakligen använda oss av Dirac-notation, se Appendix B.1.

Avsnitten 6.1-6.3 är huvudsakligen en sammanställning av De Wolfes föreläsningsanteckningar [18] med fokus mot just Shors algoritm, med vissa personliga justeringar. I avsnitt 6.4 går vi in djupare på bevisen om framgångssannolikhet genom att följa Hirvensalos resonemang i [19].

6.1 En kort introduktion till kvantfysik

Kvantfysik är en fysisk teori om hur ljus och materia beter sig på framförallt atomisk skala [20]. Vi kommer dock fokusera på några få användbara fenomen inom kvantfysiken. En egenskap som vi utnyttjar är att ett system kan befinna sig i en superposition av flera klassiska tillstånd samtidigt. Tillstånd kan representera olika egenskaper hos olika system, till exempel positionsuppgifter hos ett objekt eller spinn på en partikel. Vi gör en mer matematisk definition av fenomenet.

Definition 6.1.1. Låt $|1\rangle, |2\rangle, \ldots, |N\rangle$ vara N olika ömsesidigt uteslutande klassiska tillstånd. En *superposition* $|\psi\rangle$ är en linjärkombination av dessa på följande vis:

$$|\psi\rangle = \sum_{k=1}^{N} a_k |k\rangle$$

där $a_i \in \mathbb{C}$, $i = 1, 2, \ldots, N$ är konstanter som kallas för tillståndens amplituder. Dessa konstanter begränsas av $1 = \sum_{k=1}^{N} |a_k|^2$.

Anledningen till ovanstående begränsning är att amplitudernas kvadrat tolkas som sannolikheten att en superposition kollapsar till motsvarande tillstånd. Man kan nämligen inte observera en ren superposition, utan en så kallad mätning ger alltid klassiska tillstånd. Märk också att en mätning inte nödvändigtvis är en en mänsklig observation, utan sker när systemet interagerar med sin omgivning.

Det finns sätt man kan manipulera superpositioner förutom mätningar, bland annat genom att applicera unitära operatorer på dem. För att förstå hur dessa verkar är det nyttigt att tänka på hur tillstånden $|1\rangle, |2\rangle, \ldots, |N\rangle$ spänner ett N-dimensionellt rum. En superposition kan då ses som en vektor i detta rum. Det är enkelt att visa att just unitära operatorer krävs, och att dessa kan representeras med $N \times N$-matriser. I nästa avsnitt kommer vi gå igenom två sådana nyttiga transformer som används i Shors algoritm.

6.2 En kort introduktion till kvantberäkningar

Framöver kommer vi bara intressera oss för två olika tillstånd, $|0\rangle$ och $|1\rangle$. En kvantdators minsta beståndsdel, en så kallad *qubit*, är en kvantbit som kan befinna sig i en superposition av dessa. Vi kommer även representera tillstånden med vektorer, så att $|0\rangle = (1, 0)^T$ och $|1\rangle = (0, 1)^T$, för att enklare visualisera kommande ideér.

Man kan tänka sig att en qubit lever i $\mathbb{C}^2$ från definitionerna ovan, och system av flera qubits lever i tensorprodukten mellan dess komponenter, eftersom att vardera komponent spänner ett delrum till systemet. Till exempel har system med två qubits fyra basvektorer, $|0\rangle \otimes |0\rangle$, $|0\rangle \otimes |1\rangle$, $|1\rangle \otimes |0\rangle$ och $|1\rangle \otimes |1\rangle$. Vi kommer ofta förkorta denna notation på följande vis: $|a\rangle \otimes |b\rangle = |ab\rangle$.

Med två eller fler qubits kan vi stöta på fenomenet sammanflätning. Om vi tittar på till exempel två qubits i superpositionen $|\psi\rangle = \frac{1}{\sqrt{2}} |10\rangle + \frac{1}{\sqrt{2}} |01\rangle$ så ser vi att värdet på den andra qubiten bestäms om vi mäter den första och tvärtom. Man säger då att de är sammanflätade. Generellt innebär det att att ett objekts karaktär är beroende av ett annat objekts karaktär.

Ett annat verktyg vi kommer använda framöver är *grindar*. Vi kommer representera dem med unitära matriser (se Appendix B.2), som verkar på en eller flera qubits. Grindar är alltså unitära operatorer. Vi kommer fokusera på två olika typer av grindar i denna uppsats, *fasskiftsgrindar* och *Hadamard-transformen*.

Fasskiftsgrindar är grindar som inte påverkar amplituden om invärdet är $|0\rangle$, men som roterar amplituden hos $|1\rangle$ med en vinkel θ. Vi betecknar dessa med R_θ och ser att de kan representeras i matrisform med:

$$R_\theta = \begin{pmatrix} 1 & 0 \\ 0 & e^{\theta i} \end{pmatrix}.$$

Hadamardtransformen H är en reell matris som fördelar amplituderna likformigt över alla tillstånd, med en teckenväxling på amplituden om invärdet var $|1\rangle$ på följande vis:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

och kan representeras i matrisform som $H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$.

Hadamardtransformer kan även appliceras på system med flera qubits. Det generella uttrycket för transformen av en godtycklig sträng med n bits $j \in \{0, 1\}^n$ är:

$$H^{\otimes n} |j\rangle = \frac{1}{\sqrt{2^n}} \sum_{k \in \{0,1\}^n} (-1)^{k \cdot j} |k\rangle$$

där $k \cdot j$ är skalärprodukten mellan k och j.

Nästa verktyg vi behöver för Shors algoritm är *kvantfouriertransformen*, en kvantfysisk tolkning av den klassiska fouriertransformen.

6.3 Kvantfouriertransformen

I detta avsnitt gör vi en snabb genomgång av klassisk diskret fouriertransform och hur vi kan generera den effektivt genom snabb fouriertransform. Vi ska också undersöka hur detta kan appliceras på en superposition för att få den så kallade kvantfourieranalysen.

6.3.1 Klassisk diskret fouriertransform

Den diskreta fouriertransformen kan appliceras på data för att analysera periodiciter och starka frekvenser hos indatan [21]. Motiveringen är att de flesta tänkbara signalvågerna går att brytas ner som kombinationer av vågor med olika frekvenser. Det finns många fall då detta förenklar olika frågeställningar i områden som till exempel differentialekvationer, men i denna uppsats kommer vi fokusera på dess definition och några åtråvärda egenskaper. Vi kommer även att se på den diskreta fouriertransformen som en $N \times N$-matris för tydlighets skull när vi visar att transformen är en unitär operator.

Definition 6.3.1 (Diskret fouriertransform)**.** Låt $w_N = e^{\frac{2\pi i}{N}}$ vara den N:te enhetsroten, det vill säga den icke-triviala N:te roten till 1. Fouriertransformen F_N representeras då av matrisen med elementen $\frac{1}{\sqrt{N}} w_N^{jk}$ för varje index (j,k), $j,k = 0,\ldots,N-1$. Vektorn $\hat{v} = F_N v$ kallas för fouriertransformen av v, och dess koordinater kan skrivas som

$$\hat{v}_j = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} w_N^{jk} v_k.$$

I Appendix B.4 motiverar vi varför F_N är en unitär matris, och visar hur man kan tillämpa transformen i $\mathcal{O}(N \log N)$ steg med hjälp av snabb fouriertransform.

6.3.2 Kvantfouriertransform

Som tidigare nämnts kan F_N beskrivas med en unitär matris, vilket är tillåtet inom kvantfysiken. Transformen kallas då kvantfouriertransformen (kort QFT för *quantum Fourier transform*) och ser ut på följande vis:

$$F_N : |k\rangle \to \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} w_N^{jk} |j\rangle .$$

Märk att j och k är n-bitsträngar och tolkas som binära tal. Märk också att $F_N |k\rangle$ inte beter sig som en klassisk fouriertransform, utan fortfarande är en superposition som vi inte kan observera direkt. I Appendix B.5 visar vi hur transformen kan brytas ner till operationer på enskilda qubits, för att lättare visualisera hur detta skulle tillämpas som en serie grindar i en så kallad kvantkrets.

6.4 Shors faktoriseringsalgoritm

Som vi tidigare sett i uppsatsen så löser vissa faktoriseringsalgoritmer det enklare problemet att hitta ett elements ordning (hädanefter kommer vi referera till ordningen som period) inom en modulo-grupp, med avseende på multiplikation. Liknande koncept tillämpas nämligen i Pollards rho-algoritm i Avsnitt 5.2. Detta gör även Shors algoritm, och härnäst ska vi motivera varför.

Säg att vi har ett $x \in \{2,\ldots,N-1\}$ som är relativt primt till N (ty om x och N inte är relativt prima har vi redan en icketrivial faktor till N, nämligen $\text{sgd}(x,N)$). Nästa steg är att inse att det finns ett minsta tal $r \in \{2,...,N-1\}$ så att $x^r \equiv 1 \pmod N$, det vill säga elementet x: s period. Anta nu att $x^{r/2} \not\equiv -1 \pmod N$ och att r är jämnt. Då ser vi att

$$\begin{aligned} x^r &\equiv 1 \pmod N \Rightarrow \\ x^r - 1 &= (x^{r/2}-1)(x^{r/2}+1) \equiv 0 \pmod N \Rightarrow \\ (x^{r/2}-1)(x^{r/2}+1) &= kN, \text{ för något } k \in \mathbb{Z}. \end{aligned}$$

Eftersom att $x^{r/2} \not\equiv \pm 1 \pmod N$ enligt periodens egenskaper och antagande måste alltså $\text{sgd}(x^{r/2} \pm 1, N)$ vara två icketriviala faktorer till N.

Det visar sig att hitta ett element som uppfyller villkoren ovan inte är särskilt svårt, sannolikheten att ett slumpvalt x gör det är minst $\frac{3}{8}$, se Appendix B.6.

6.4.1 Shors periodsökaralgoritm

Det Shors algoritm gör är att hitta perioden i fråga. Låt först $N^2 \leq Q = 2^n < 2N^2$ för något $n \in \mathbb{Z}$. Q behöver vara stort eftersom att vi vill se flera perioder i de kommande stegen, och förhållandet till N kommer senare förenkla räkningarna när det kommer till kedjebråkskonvergens. Vi börjar algoritmen med strängen som består av $2n$ nollor. Anledningen till att vi väljer just $2n$ är för att strängen ska kunna representera två olika värden i två olika register, ett index a och ett $f(a) = x^a \pmod N$. Hädanefter kommer vi för enkelhet referera till de olika registren efter vilket binärt tal de representerar.

Första steget är att applicera Hadamardtransformen på första registret. Vi får:

$$H^{\otimes n} |0\rangle |0\rangle = \frac{1}{\sqrt{Q}} \sum_{a=0}^{Q-1} |a\rangle |0\rangle .$$

Därefter kallar vi på varje funktionsvärde $f(a)$ i andra registret (det finns snabba klassiska algoritmer för att beräkna x^a [22]), vilket ger:

$$\frac{1}{\sqrt{Q}} \sum_{a=0}^{Q-1} |a\rangle |f(a)\rangle .$$

Nästa steg är att göra en mätning på andra registret och för att generera någon superposition:

$$\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |jr + s\rangle |f(s)\rangle .$$

där $0 \leq s < r$, och m är antalet element så att $jr + s < Q$. Vi kan nu ignorera andra registret och applicera en kvantfouriertransform på första registret:

$$F_Q \left(\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |jr + s\rangle \right) = \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} \frac{1}{\sqrt{Q}} \sum_{b=0}^{Q-1} w_Q^{(jr+s)b} |b\rangle = \frac{1}{\sqrt{mQ}} \sum_{b=0}^{Q-1} w_Q^{sb} \left(\sum_{j=0}^{m-1} w_Q^{bjr} \right) |b\rangle . \tag{6.4.1}$$

Om vi gör en mätning på denna superpositionen så kommer det visa sig att vissa speciella b kommer framträda med god sannolikhet. Dessa b kan vi använda för att hitta perioden. Vi visar först detta i ett enkelt fall då $r|Q$.

6.4.2 Enkel framgångssannolikhet då $r|Q$

Om $r|Q$ har vi att $m = \frac{Q}{r}$. Sannolikheten $P(b)$ att bevittna något b är

$$P(b) = \left| \frac{w_Q^{sb}}{\sqrt{mQ}} \sum_{j=0}^{m-1} w_Q^{bjr} \right|^2 = \frac{1}{mQ} \left| \sum_{j=0}^{m-1} e^{\frac{2\pi i b j r}{Q}} \right|^2 = \begin{cases} \frac{m^2}{mQ} = \frac{1}{r} \text{ om } b = c\frac{Q}{r}, & c \in \mathbb{Z}_r \\ 0 \text{ annars.} \end{cases}$$

Vi kan konstatera att alla $b \in \{0, \frac{Q}{r}, \dots (r-1)\frac{Q}{r}\}$ sammanlagt har sannolikhet 1 att observeras i en mätning. Därefter kan vi lösa ut r genom att applicera kedjebråk på $\frac{b}{Q}$ för att få ut $\frac{c}{r}$ och identifiera r i nämnaren (se avsnitt 5.5 för att läsa mer om kedjebråk). Detta fungerar dock bara om $\text{sgd}(c, r) = 1$, men det visar sig vara tämligen sannolikt, se Appendix B.7. Vad som däremot inte är särskilt sannolikt är $r|Q$, eftersom att Q är 2-potens, men vi ska visa att b vi kan utnyttja dyker upp även i det generella fallet.

6.4.3 Generell framgångssannolikhet då $r \nmid Q$

Även om b inte är en multipel av $\frac{Q}{r}$ kan $\frac{c}{r}$ vara en så kallad konvergent (se 5.5) till kedjebråksutvecklingen av $\frac{b}{Q}$. Detta är fallet om

$$|br - cQ| \leq \frac{r}{2}$$

(se [19, sid. 57-58, sid. 169-175]). Det finns r sådana b för varje bråk $\frac{c}{r}$, ty $c \in \mathbb{Z}_r$. Härnäst har vi ett lemma som säger att vi sannolikt kan välja ett sådant b med $\mathcal{O}(1)$ steg.

Lemma. *Låt $N \geq 100$. Då är sannolikheten att observera ett b i (6.4.1) sådant att $|br - cQ| \leq \frac{r}{2}$ inte mindre än $\frac{1}{5}$.*

Bevis. Se Appendix B.8. □

Slutsatsen är alltså att oavsett om r delar Q eller inte så kan vi sannolikt välja ett b som vi kan hitta perioden med i $\mathcal{O}(1)$ steg. Den slutgiltiga komplexiteten för att Shors algoritm sannolikt hittar icke-trivial faktor till N blir $\mathcal{O}(\log^3(N) \log\log(N))$, se Appendix B.10.

7 Slutsats

Bakgrunden till arbetet är de två kryptosystemen ElGamal och RSA som beskrevs i Kapitel 2. Där visades att RSA är ett snabbt och effektivt kryptosystem givet att man har två väl valda stora primtal. I Kapitel 4 visades att det är väldigt enkelt att hitta primtal med hjälp av stokastiska test som Solovay-Strassens och Miller-Rabins. Även om dessa stokastiska test inte definitivt bestämmer om ett givet tal är ett primtal så kan de ge en extremt hög sannolikhet vilket är tillräckligt för tänkbara tillämpningar. Därmed har det visats att med RSA är det enkelt och snabbt att utföra själva krypteringen och dekrypteringen, om man har den privata nyckeln.

I Kapitel 3 jämfördes två metoder för att lösa DLP och vi såg en stor skillnaden i beräkningstid beroende på vilken algoritm som valdes. Vi såg även att tiden för att lösa DLP ökade snabbt med storleken på primtalet p. Vi drar slutsatsen att de nycklar som används idag är säkra för våra algoritmer som angriper det diskreta logaritmproblemet direkt. Men det finns anledning till viss oro, Diffie-Hellman kryptering som baserar sin säkerhet på 512-bit nycklar har redan visat sig möjlig att lösa i många fall, t.ex. i den uppmärksammade logjam-attacken. Adrian med flera [6] uppskattar att en aktör med tillräckliga resurser, t.ex. NSA, skulle kunna lösa system med 1024-bit nycklar som används flitigt idag.

När vi testade våra faktoriseringsalgoritmer i Kapitel 5 fick vi flera intressanta resultat. Den direkta metoden samt Fermats algoritm var väldigt långsamma som väntat. Mer intressant var att Dixons algoritm bara var aningen snabbare än den direkta metoden i ett fall, och att det kvadratiska sållet aldrig var snabbast. Det kvadratiska sållet har lägst komplexitet av samtliga implementarade faktoriseringsalgoritmer och flera författare så som Koblitz [2, s. 160], Hoffstein med flera [3, s. 152], och Pomerance [8, s. 1] beskriver det Kvadratiska Sållet som den snabbaste algoritmen av de vi har testat. Detta visar på att komplexitet aldrig berättar hela historien, kanske är det så att vi testat för små tal. Pollards rho-algoritm tog en oväntad första plats i samtliga test men visar tecken på att inte skala lika bra med stora värden på N som Kedjebråksmetoden och det Kvadratiska Sållet. Det finns ingen känd algoritm till en klassisk dator för att lösa faktoriseringsproblemet i polynomiell komplexitet (se t.ex. [17, s.278]). Detta innebär att problemet är mycket känsligt för storleken på N. Våra algoritmer saktade in snabbt och vi kan dra slutsatsen att ingen av dem skulle kunna användas till att lösa faktoriseringsproblemet för primtal som används till RSA idag.

I Kapitel 6 beskrivs kvantalgoritmen Shors algoritm. Det intressanta med den är att dess komplexitet är mycket mindre än alla hittills kända klassiska algoritmer. Som tidigare nämnt är det ett öppet problem om klassiska algoritmer kan faktorisera tal med polynomiell komplexitet. Detta lyckas dock Shors algoritm med. Ledande forskare ser kvantdatorer som det största hotet mot klassiska kryptosystem och flera aktörer arbetar med att ta fram så kallade kvantsäkra kryptosystem (se [23, s. 6]).

Slutsatsen om dagens kryptosystem är säkra blir Ja, om man väljer sina parametrar korrekt. I framtiden finns det möjlighet både för att kvantdatorer slår igenom (se [23, s. 8]), och att klassisk beräkningskraft ökar stort (se [24, s. 3]). Detta kan utgöra en risk.

Referenser

[1] John R. Durbin. *Modern Algebra: An Introduction, 6th Edition*. Wiley, 2008. ISBN: 978-0-470-38443-5.

[2] Neal Koblitz. *A Course in Number Theory and Cryptography*. Springer-Verlag New York, 1994. ISBN: 978-1-4419-8592-7.

[3] Jeffrey Hoffstein, Jill Pipher och J.H. Silverman. *An Introduction to Mathematical Cryptography*. Springer-Verlag New York, 2008. ISBN: 978-1-4939-1710-5.

[4] W. Diffie och M. Hellman. "New Directions in Cryptography". I: *IEEE Trans. Inf. Theor.* 22.6 (nov. 1976), s. 644–654. URL: http://dx.doi.org/10.1109/TIT.1976.1055638.

[5] T. M. Zaw, M. Thant och S. V. Bezzateev. "User Authentication in SSL Handshake Protocol with Zero-Knowledge Proof". I: *2018 Wave Electronics and its Application in Information and Telecommunication Systems (WECONF)*. Nov. 2018. URL: http://dx.doi.org/10.1109/WECONF.2018.8604392.

[6] David Adrian m. fl. "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice". I: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, 2015, s. 5–17. ISBN: 978-1-4503-3832-5. URL: http://doi.acm.org/10.1145/2810103.2813707.

[7] Taher ElGamal. "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". I: *IEEE Transactions on Information Theory* IT-31.4 (1985), s. 469–472.

[8] Carl Pomerance. "A tale of two sieves". I: *Notices Amer. Math. Soc* 43 (1996), s. 1473–1485. URL: https://www.ams.org/notices/199612/pomerance.pdf.

[9] John Loeffler. "How Peter Shor's Algorithm Dooms RSA Encryption to Failure". I: (2019). Interesting Engineering. URL: https://interestingengineering.com/how-peter-shors-algorithm-dooms-rsa-encryption-to-failure.

[10] Peter W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring". I: (1994). Last accessed 15 May 2019. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=365700.

[11] R. Solovay och V. Strassen. "A Fast Monte-Carlo Test for Primality". I: *SIAM Journal on Computing* 6.1 (1977), s. 84–85. URL: https://doi.org/10.1137/0206006.

[12] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag New York, 1993. ISBN: 3-540-55640-0.

[13] Steven D Galbraith. "Towards a rigorous analysis of Pollard rho". I: *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012. Kap. 14.2.5, s. 272–273. ISBN: 978-1-107-01392-6.

[14] R. Sherman Lehman. "Factoring Large Integers". I: *MATHEMATICS OF COMPUTATION* 28.126 (1974), s. 637–646. URL: http://www.ams.org/journals/mcom/1974-28-126/S0025-5718-1974-0340163-2/S0025-5718-1974-0340163-2.pdf.

[15] Michael A. Morrison och John Brillhart. "A Method of Factoring and the Factorization of F_7". I: *Mathematics of Computation* 29.129 (1975), s. 183–205. URL: https://doi.org/10.1090/S0025-5718-1975-0371800-5.

[16] Carl Pomerance. "Analysis and Comparison of some Common Integer Factoring Algorithms". I: *Computational Methods in Number Theory*. Mathematisch Centrum, 1982, s. 89–139.

[17] Richard Crandall och Carl Pomerance. *Prime Numbers, A Computational Perspective*. Springer-Verlag New York, 2005. ISBN: 978-0-387-28979-3.

[18] R. De Wolfe. "Quantum Computing: Lecture Notes". I: (2019). Last accessed 15 May 2019, s. 1–38. URL: https://homepages.cwi.nl/~rdewolf/qcnotes.pdf.

[19] Mika Hirvensalo. *Quantum Computing*. Springer-Verlag Berlin Heidelberg, 2004, s. 56–61. ISBN: 978-3-540-40704-1.

[20] Michael A. Gottlieb och Rudolf Pfeiffer. “Quantum Behavior”. I: (2013). Last accessed 15 May 2019, s. 1–38. URL: http://www.feynmanlectures.caltech.edu/III_01.html.

[21] Eric W. Weisstein. “Discrete Fourier Transform”. I: (2019). Last accessed 15 May 2019. URL: http://mathworld.wolfram.com/DiscreteFourierTransform.html.

[22] Martin Fürer. “Faster Integer Multiplication”. I: (2007). Last accessed 15 May 2019. URL: https://ivv5hpp.uni-muenster.de/u/cl/WS2007-8/mult.pdf.

[23] Vasileios Mavroeidis m. fl. “The Impact of Quantum Computing on Present Cryptography”. I: *International Journal of Advanced Computer Science and Applications* 9.3 (2018). URL: http://arxiv.org/abs/1804.00200.

[24] Lawrence M. Krauss och Glenn D Starkman. *Universal Limits on Computation*. 2004. URL: https://arxiv.org/abs/astro-ph/0404510.

[25] http://www.asciitable.com/. *ASCII Table and Description*. Last accessed 20 April 2019. 2019. URL: http://www.asciitable.com/.

[26] Inc. Unicode. *About the Unicode Character Database*. Last accessed 20 April 2019. 2019. URL: http://www.unicode.org/ucd/.

[27] Stanford Sir Charles Antony Richard Hoare. *Quicksort*. Last accessed 20 April 2019. 2008. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2008-09/tony-hoare/quicksort.html.

[28] Robert Sedgewick och Kevin Wayne. *Algorithms, 4th Edition, Kapitel 2.2*. Last accessed 20 April 2019. 2018. URL: https://algs4.cs.princeton.edu/22mergesort/.

[29] ChitraNayal och Mayank Khanna 2. *Merge Sort*. Last accessed 3 May 2019. URL: https://www.geeksforgeeks.org/merge-sort/.

[30] J. Barkley och Lowell Schoenfeld. *Approximate formulas for some functions of prime numbers*. Illinois Journal of Matematics, 1962, s. 64–94.

A Fördjupning

$\mathcal{O}(N^{\frac{1}{4}})$

A.1 Kinesiska restklassatsen

Vi har tidigare stött på en hel del moduloräkning, och den kinesiska restklassatsen säger något om när vi kan hitta unika lösningar till system av moduloekvationer.

Sats A.1.1 (Kinesiska restklassatsen)**.** *Låt $m_1, m_2, ..., m_k$ vara k relativt prima heltal, och $a_1, a_2, ..., a_k$ vara godtyckliga heltal. Då har systemet*

$$x \equiv a_1 \pmod{m_1}, \quad x \equiv a_2 \pmod{m_2}, \quad \dots, \quad x \equiv a_k \pmod{m_k}$$

en lösning, $x = c$. Om $x = c'$ också är en lösning har vi att $c \equiv c' \pmod{m_1 m_2 ... m_k}$.

Beviset är ett induktionsbevis och visar inte bara existens hos lösningen utan även konstruktion. För att tydliggöra börjar vi med ett exempel.

Exempel. *Låt*

$$\begin{cases} x \equiv 1 \pmod{3} \\ x \equiv 6 \pmod{7}. \end{cases}$$

Från första ekvationen ser vi att $x = 1 + 3y$ för godtyckligt heltal y. Om vi sätter in detta i vår andra ekvation får vi

$$1 + 3y \equiv 6 \pmod{7} \quad \Rightarrow \quad 3y \equiv 5 \pmod{7}.$$

För att lösa ut y här multipliceras höger- och vänsterledet med den multiplikativa inversen av 3 modulo 7. Eftersom $sgd(3, 7) = 1$ vet vi att inversen existerar (enligt Sats A.4.2), och i detta fall är det lätt att hitta den med trial and error. Vi har att $3^{-1} \equiv 5 \pmod{7}$ och därmed fås

$$\begin{aligned} &y \equiv 25 \equiv 4 \pmod{7} \quad \Rightarrow \quad y = 4 + 7z, \quad z \in \mathbb{Z} \\ &x = 1 + 3(4 + 7z) = 13 + 21z. \end{aligned}$$

Alltså är $x \equiv 13 \pmod{21}$ alla lösningar till systemet. Nu när vi sett tillvägagångssättet kan vi generalisera processen och bevisa tidigare sats.

Bevis. Säg att vi har löst första ekvationen, så att $x \equiv a_1 \pmod{m_1}$. Vi vill visa att vi kan lösa hela systemet induktivt. Anta nu att vi har löst de första $0 \leq i < k$ ekvationerna så att

$$x \equiv a_1 \pmod{m_1}, \quad x \equiv a_2 \pmod{m_2}, \quad \dots, \quad x \equiv a_i \pmod{m_i}$$

och låt $x = c_i + m_1 m_2 \cdots m_i y$. För att lösa ännu en ekvation behöver vi välja y så att

$$c_i + m_1 m_2 \cdots m_i y \equiv a_{i+1} \pmod{m_{i+1}}.$$

Eftersom att $m_1, m_2, ..., m_k$ är relativt prima kan vi alltid finna ett sånt y (enligt sats A.4.2). För att visa att lösningen är unik modulo $m_1 m_2 \cdots m_k$ antar vi att $x = c$ och $x = c'$ är lösningar. Eftersom att

$$m_1 | c - c', \quad m_2 | c - c', \quad \dots \quad m_k | c - c'$$

och eftersom att alla m är relativt prima har vi att

$$m_1 m_2 \cdots m_k | c - c' \quad \Rightarrow \quad c \equiv c' \pmod{m_1 m_2 \cdots m_k}.$$

□

Satsen har fått sitt namn av att problemet formulerades för första gången runt sent 300-tal till tidigt 400-tal av kinesiska matematiker[3]. Den visar sig var väldigt användbar i många sammanhang, bland annat när man löser diskreta logaritmproblem. Pohlig-Hellman algoritmen beskriver detta, ty lösningen x lever som tidigare nämnt i $\mathbb{Z}/(p-1)\mathbb{Z}$, så genom att faktorisera $p-1$ kan vi dela upp det ursprungliga problemet i flera moduloproblem, och sedan berättar kinesiska restklassatsen hur vi flätar ihop det till den slutgiltiga lösningen.

A.2 Komplexitet

För att kunna uppskatta hur snabb en algoritm är kommer vi behöva begreppet komplexitet, vår definition är baserad på Hoffsteins m.fl. beskrivning (se [3, s.152]).

Definition A.2.1. Låt $g : \mathbb{R} \to \mathbb{N}$ och $f : \mathbb{R} \to \mathbb{R}$. En algoritm som tar $g(n)$ steg att lösa för indatan n har komplexitet $\mathcal{O}(f(n))$ om $\exists N, \in \mathbb{R},\ C \in \mathbb{R}$ sådant att

$$g(n) \leq C \cdot f(n) \quad \forall n \geq N.$$

En algoritm som tar $g(n)$ steg att lösa för indatan n har komplexitet $\Omega(f(n))$ om $\exists N, \in \mathbb{N},\ C \in \mathbb{R}$ sådant att

$$g(n) \geq C \cdot f(n) \quad \forall n \geq N.$$

En algoritm som tar $g(n)$ steg att lösa för indatan n har komplexitet $\Theta(f(n))$ om $\exists N, \in \mathbb{N},\ C \in \mathbb{R}$ sådant att

$$\frac{1}{C} \cdot f(n) \leq g(n) \leq C \cdot f(n) \quad \forall n \geq N.$$

Notera: Med att algoritmen tar ett steg kan avses att man gör en addition, en multiplikation eller en antagningsvis grundläggande beräkning.

För att kunna förenkla analysen av komplexiteterna i arbetet använder vi följande räkneregler:

- $\mathcal{O}(g(x)) + \mathcal{O}(f(x)) = \mathcal{O}(g(x) + f(x))$.

  Bevis. $\mathcal{O}(g(x)) + \mathcal{O}(f(x))$ ger $\mathcal{O}(g(x) + f(x))$:
  Låt $C_3 = \max(C_1, C_2)$. Vi får att

  $$C_1 g(x) + C_2 f(x) \leq C_3 g(x) + C_3 f(x) = C_3(g(x) + f(x)).$$

  $\mathcal{O}(g(x) + f(x))$ ger $\mathcal{O}(g(x)) + \mathcal{O}(f(x))$:

  $$C_3(g(x) + f(x)) = C_3 g(x) + C_3 f(x).$$

  Alltså är $\mathcal{O}(g(x)) + \mathcal{O}(f(x)) = \mathcal{O}(g(x) + f(x))$. □

- Om $\exists N \in \mathbb{R} : g(x) \leq f(x)\ \forall x \geq N$ så är $\mathcal{O}(g(x)) + \mathcal{O}(f(x)) = \mathcal{O}(f(x))$.

  Bevis. $\mathcal{O}(g(x)) + \mathcal{O}(f(x))$ ger $\mathcal{O}(f(x))$:
  Vi har att

  $$C_1 g(x) + C_2 f(x) \leq C_1 f(x) + C_2 f(x) = (C_1 + C_2) f(x), \quad \forall x \geq N.$$

  $\mathcal{O}(f(x))$ ger $\mathcal{O}(g(x)) + \mathcal{O}(f(x))$:
  Enligt Definition A.2.1 finns det $C_5 \in \mathbb{R} : 0 \leq C_5 g(x)\ \forall x \geq N$ eftersom $C_5 g(x)$ är större än en funktion i $\mathbb{N}$.

  $$C_4 f(x) \leq C_4 f(x) + C_5 g(x), \quad \forall x \geq N.$$

  Därav är $\mathcal{O}(g(x)) + \mathcal{O}(f(x)) = \mathcal{O}(f(x))$. □

- $\mathcal{O}(g(x))\mathcal{O}(f(x)) = \mathcal{O}(g(x)f(x))$.

Bevis. Vi har att

$$C_1 g(x) C_2 f(x) = (C_1 C_2) g(x) f(x).$$

Därav är $\mathcal{O}(g(x))\mathcal{O}(f(x)) = \mathcal{O}(g(x)f(x))$. □

- Antag att det finns ett N i $\mathbb{R}$ sådant att $1 \leq g(x)$ och $1 \leq f(x)$, för alla x större än N. Då har vi att komplexiteten $\mathcal{O}(g(x)h(x) + h(x)f(x))$ ger $\mathcal{O}(g(x)h(x)f(x))$.

  Bevis. Vi har att

$$\begin{aligned} C_1(g(x)h(x) + h(x)f(x)) &\leq C_1(g(x)h(x)f(x) + g(x)h(x)f(x)) = \\ &= 2C_1(g(x)h(x)f(x)), \quad \forall x \geq N. \end{aligned}$$

  Därav stämmer påståendet. □

A.3 Fast Powering Algorithm

Fast powering algorithm används för att beräkna det minsta positiva heltal h som löser ekvationen

$$h \equiv g^a \pmod{p}$$

för givna heltal g, a och p. Vi börjar med att skriva a på binär form, det vill säga $a = \sum_{k=0}^{n} c_k 2^k$ där $c_i \in \{0, 1\},\ \forall i \in \mathbb{N} : 0 \leq i \leq n$. Vi har nu att

$$g^a \equiv g^{\sum_{k=0}^{n} c_k 2^k} \equiv \prod_{k=0}^{n} g^{c_k 2^k} \pmod{p}.$$

För de k som $c_k = 0$ så får vi att $g^{c_k 2^k} = g^0 = 1$ och vi ser att vi inte behöver multiplicera med dem i produkten. Nu behöver vi endast beräkna g^{2^k} och sedan multiplicera de faktorer vars $c_k \neq 0$. Vi beräknar g^{2^k} faktorerna igenom upprepad kvadrering, det vill säga

$$\begin{aligned} v_0 &\equiv g \pmod{p}, \\ v_{k+1} &\equiv (v_k)^2 \pmod{p}, \quad \forall k \in \mathbb{N} : 1 \leq k \leq n. \end{aligned}$$

Slutligen multiplicerar vi de v_k som korresponderar med $c_k \neq 0$, tar modulo p i varje multiplikation och då erhåller vi h.

Sats A.3.1. *Komplexiteten för Fast Powering Algorithm är* $\mathcal{O}(\log(a))$.

Notera: Vi antar här att multiplikation och att hämta tal ur en dators minne är grundläggande operationer.

Bevis. I en dator är tal skrivna i binärt, så att finna a:s binära utvecklingen anses ha komplexitet $\mathcal{O}(1)$. För att beräkna alla v_k behöver vi kvadrera n gånger vilket ger komplexiteten $\mathcal{O}(n) = \mathcal{O}(\log(a))$. Denna omskrivning fungerar eftersom $n = \lfloor \log_2(a) \rfloor \leq \frac{\log(a)}{\log(2)}$. Slutligen multiplicerar vi ihop talen och eftersom det finns n stycken så blir komplexiteten $\mathcal{O}(n) = \mathcal{O}(\log(a))$. Den totala komplexiteten blir då

$$\mathcal{O}(1) + \mathcal{O}(\log(a)) + \mathcal{O}(\log(a)) = \mathcal{O}(\log(a))$$

och vi är klara. □

A.4 Sats A.4.1 och A.4.2 med bevis

Här visar vi två satser som är viktiga inom RSA.

Sats A.4.1. *Låt p och q vara två olika primtal. Låt även*

$$g = sgd(p-1, q-1).$$

Då är

$$a^{(p-1)(q-1)/g} \equiv 1 \pmod{pq} \quad \forall a \text{ s.a. } sgd(a, pq) = 1.$$

Särskilt om p och q är udda primtal, då är

$$a^{(p-1)(q-1)/2} \equiv 1 \pmod{pq} \quad \forall a \text{ s.a. } sgd(a, pq) = 1$$

Bevis. Eftersom $p \nmid a$ och $g|q-1$ kan vi beräkna $a^{(p-1)(q-1)/g} = \left(a^{p-1}\right)^{(q-1)/g} \equiv$ {enligt fermats lilla} $\equiv 1^{(q-1)/g} \equiv 1 \pmod{p}$. Analoga beräkningar för q ger att $a^{(p-1)(q-1)/g} \equiv 1 \pmod{q}$. Det här visar att $a^{(p-1)(q-1)/g} - 1$ är delbart med p och q, som är relativt prima, vilket ger att $pq|a^{(p-1)(q-1)/g} - 1$. □

Sats A.4.2. *Låt $a, m \in \mathbb{N}$. Då gäller att*

$$\exists\, b \in \mathbb{N} : ab \equiv 1 \pmod{m} \iff sgd(a, m) = 1. \tag{A.4.1}$$

Bevis. Antag först att $\text{sgd}(a, m) = 1$. Man kan då visa att det finns heltal u, v sådana att $au + mv = 1$. Av det följer att m delar $au - 1$ så

$$au \equiv 1 \pmod{m} \tag{A.4.2}$$

vilket ger oss ett b.

Antag nu att a har en invers b modulo m, det vill säga att

$$ab \equiv 1 \pmod{m}. \tag{A.4.3}$$

Då finns det ett heltal c sådant att $ab - 1 = cm$. Men då gäller $\text{sgd}(ab, cm) = 1 \Rightarrow \text{sgd}(a, m) = 1$. □

A.5 RSA-algoritmen

Här ges algoritmen som Alice och Bob följer [sid. 123[3]].
Bob tar fram den publika nyckeln.

1: Välj två hemliga primtal p och q.

2: Välj en krypteringsexponent e s.a $\text{sgd}(e, (p-1)(q-1)) = 1$.

3: Beräkna $N = pq$.

4: Publicera N och e.

Alice krypterar sitt meddelande.

5: Välj meddelande m_1.

6: Använd den publika nyckeln (N, e) för att beräkna chiffret $c \equiv m_1^e \pmod{N}$.

7: Skicka c till Bob.

Bob avkrypterar chiffret c.

8: Beräkna d s.a $ed \equiv 1 \pmod{(p-1)(q-1)}$.

9: Beräkna $m_2 \equiv c^d \pmod{N}$.

Nu är $m_1 = m_2$.

A.6 Försållning

Miller-Rabin primtalstest kräver flertalet moduloberäkningar. Detta gäller även då n är delbart med små primtal. Emellertid krävs endast en moduloräkning för att avgöra att ett tal är delbart med ett annat fixt tal. Antag att vi vill slumpmässigt erhålla ett N siffror stort primtal med hjälp av Miller-Rabin. Det man då kan göra är att låta en slumpgenerator ge ett udda tal n för att sedan primtalstesta det. Visar det sig att detta tal är sammansatt undersöker man $n+2$ och så vidare.

Sats A.6.1 (Primtalssatsen)**.** *Om $\pi(n)$ är antalet olika primtal mindre än n så gäller*

$$\pi(n) \longrightarrow \frac{n}{\ln(n)} \quad \text{då } n \longrightarrow \infty. \tag{A.6.1}$$

Av primtalssatsen ovan följer det vi kan förvänta oss 1 primtal på 1381 sammansatta i en omgivning runt 10^{600}. Däremot är exakt vart tredje tal delbart med tre, vart femte delbart med fem osv. Därför undersöker vi delbarheten med de lägre primtalen för att slippa testa dessa med Miller-Rabin-algoritmen.

Med Erastotenes såll får man mycket snabbt en lista med primtal som man kan använda för detta ändamål. Hur stora primtal denna lista ska innehålla bestäms av storleken på det önskade primtalet.

A.7 Bevis av proposition 5.3.1

Proposition. *Låt $a^2 - b^2 = kN$, $N = pq$ där $a, b, k, p, q \in \mathbb{N}$ samt $c = sgd(a+b, N)$ och $d = sgd(a-b, N)$. Låt även p och q vara primtal samt $p \neq q$. Om nu c och d varken är 1 eller N så är $N = cd$.*

Bevis. Vi ser att eftersom $c = \text{sgd}(a+b, N)$ och $N = pq$ så måste $c \in \{1, p, q, N\}$ på grund av att p och q är primtal. Satsen sa att $c \neq 1$ och $c \neq N$ vilket ger att $c \in \{p, q\}$. Samma argument för d ger att $d \in \{p, q\}$. Vi ser nu att de enda sätten som ger att $N \neq cd$ är då $c = d = p$ eller $c = d = q$. Antag att $c = d = p$. Vi observerar nu att

$$kN = a^2 - b^2 = (a+b)(a-b).$$

Vi har att c är den största gemensamma delaren mellan $(a+b)$ och N vilket ger att $\exists\, m_1 \in \mathbb{N}$ sådant att $m_1 c = (a+b)$ och $q \nmid m_1$. På samma sätt gäller att $\exists\, m_2 \in \mathbb{N}$ sådant att $m_2 d = (a-b)$ och $q \nmid m_2$. Vi använder detta och får

$$(a+b)(a-b) = m_1 c\, m_2 d = m_1 m_2 p^2$$

vilket ger att

$$m_1 m_2 p^2 = kN = kpq.$$

Vi ser nu att q delar högerledet men inte vänsterledet eftersom $q \nmid m_1$ och $q \nmid m_2$. Notera även att $q \nmid p$ eftersom p är ett primtal och $p \neq q$. Samma argument visar att $c = d = q$ ger en motsägelse. Vi kan nu konstatera att $c = p$, $d = q$ eller $c = q$, $d = p$ vilket ger att $N = cd$. □

A.8 Översätta meddelanden till tal

Att göra om ett meddelande till ett tal i en dator är ganska lätt. Man kan ta varje tecken i meddelandet och översätta det till sitt ascii-värde [25]. Sedan sätter man samman värdena eftervarandra varvid man får ett tal. Om detta talet är för stort kan man dela upp meddelandet i mindre bitar och skicka dem mindre meddelandena vart för sig. Om man vill använda mer tecken än vad so är definierat i den vanliga Ascii kodningen så kan man istället använda Unicode [26] men eftersom Unicode tecknens korresponderande värden är större så kommer man kanske behöva dela upp meddelandet i mindre bitar än om man använder Ascii.

A.9 Att finna inverser modulo primtal.

Att finna inversen till ett tal modulo ett primtal p går att göra i $\mathcal{O}(\log(p))$ tid med hjälp av Fast Powering Algorithm , se A.3, och Fermats lilla sats. Om vi har ett primtal p och ett heltal a så ger fermats lilla sats att

$$a^p \equiv a \pmod{p}.$$

Om vi förkortar bort ett a får vi att $a^{p-1} \equiv 1 \pmod{p}$. Vi kan nu bryta ut ett a och se att $a^{p-1} = a^{p-2} \cdot a \equiv 1 \pmod{p}$, vilket ger att $a^{p-2} \equiv a^{-1} \pmod{p}$ eftersom $1 = a^{-1} \cdot a$ i $\mathbb{F}_p$.

Nu återstår det endast att beräkna $a^{p-2} \pmod{p}$ för att finna $a^{-1} \pmod{p}$, vilket går att göra i $\mathcal{O}(\log(p))$ tid med Fast Powering Algorithm.

A.10 Lösning till (5.2.2)

$$\begin{aligned}
I^2 &= \int_0^\infty \int_0^\infty x^2 e^{-\frac{x^2}{2}} y^2 e^{-\frac{y^2}{2}} \\
&= \text{Polära koordinater} \begin{cases} x = r \cdot \cos\theta \\ y = r \cdot \sin\theta \end{cases} \\
&= \int_0^\infty \int_0^{\frac{\pi}{2}} (r\cos\theta)^2 e^{-\frac{(r\cdot\cos\theta)^2}{2}} (r\cdot\sin\theta)^2 e^{-\frac{(r\cdot\sin\theta)^2}{2}} r dr d\theta \\
&= \Big(\int_0^\infty r^5 e^{-\frac{r^2}{2}}\Big)\Big(\int_0^{\frac{\pi}{2}} (\cos^2\theta)(\sin^2\theta) d\theta\Big).
\end{aligned}$$

I sista likheten använde vi den trigonometriska ettan $\cos^2\theta + \sin^2\theta = 1$. Vi har nu gjort om I^2 till en produkt av två integraler. Vi börjar med att beräkna den första integralen med hjälp av variabelbytet $[r^2 = z,\ \frac{dr}{dz} = \frac{1}{\sqrt{z}}\frac{1}{2}]$ och att använda partiell integration två gånger.

$$\begin{aligned}
&\int_0^\infty r^5 e^{-\frac{r^2}{2}} dr = \int_0^\infty z^2 e^{-\frac{z}{2}} \frac{1}{2} dz = \Big[-z^2 e^{-\frac{z}{2}}\Big]_0^\infty + 2\int_0^\infty z e^{-\frac{z}{2}} dz = 2\int_0^\infty z e^{-\frac{z}{2}} dz \\
&= \Big[-4z e^{-\frac{z}{2}}\Big]_0^\infty + 4\int_0^\infty e^{-\frac{z}{2}} dz = 4\int_0^\infty e^{-\frac{z}{2}} dz = \Big[-8e^{-\frac{z}{2}}\Big]_0^\infty = 8.
\end{aligned}$$

För att beräkna den andra integralen använder vi att $\cos^2\theta\sin^2\theta = (\cos\theta\sin\theta)^2 = \big(\frac{1}{2}\sin(2\theta)\big)^2 = \frac{1}{4}\frac{1-\cos(4\theta)}{2}$. Vi får att

$$\int_0^{\frac{\pi}{2}} (\cos^2\theta)(\sin^2\theta) d\theta = \int_0^{\frac{\pi}{2}} \frac{1}{4}\frac{1-\cos(4\theta)}{2} d\theta = \Big[\frac{\theta}{8} - \frac{\sin(4\theta)}{32}\Big]_0^{\frac{\pi}{2}} = \frac{\pi}{16}.$$

Vi kan nu räkna ut att $I^2 = 8 \cdot \frac{\pi}{16} = \frac{\pi}{2}$ och därmed att $I = \sqrt{\frac{\pi}{2}}$.

A.11 Kollision i Pollards rho-algoritm

Sats A.11.1. *Låt $f : \mathbb{Z}_N \to \mathbb{Z}_N$ och låt $x_0 \in \mathbb{Z}_N$ vara ett startelement. Antag att banan $O_f^+(x) = x_0, x_1, x_2, \ldots$ är av längd T och att loopen är av längd M. Då är*

$$x_{2i} = x_i \text{ för något } 1 \le i < T + M$$

Bevis. Låt $j > i$ då har vi att $x_j = x_i$ om och endast om $i \ge T$ och $j \equiv i \bmod M$, eftersom vi får att $x_j = x_i$ när i har tagit sig in i loopen och x_j har tagit sig igenom loopen och förbi x_i ett antal gånger. Vi har alltså att $j - i$ är en multipel till M. Därför är $x_{2i} = x_i$ om och endast om $i \ge T$ och $2i \equiv i \bmod M$. Vi har alltså att $M|i$ så vi får att $x_{2i} = x_i$ när i är första multipeln till M som är större än T. Men vi vet också att något av talen $T, T+1, ..., T+M-1$ är delbart med M, det visar att $x_{2i} = x_i$ för något $1 \le i < T + M$. □

A.12 Alternativ version av Pollards rho-algoritm

I den alternativa versionen av Pollards rho-algoritm har vi ett heltal N som är produkten av två primtal, $N = pr$, och vi vill hitta en av dess faktorer, kallad r. Denna versionen av Pollards rho-algoritm baseras på teorin från s.138-143 i [2].

Vi börjar med att visa algoritmen och idéerna bakom algoritmen innan vi förklarar varför denna metod fungerar. Denna version av Pollards rho-algoritm är följande

Steg 1: Välj en funktion $f : \mathbb{Z}_N \to \mathbb{Z}_N$, ett startvärde $x_0 \in \mathbb{Z}_N$ och sätt k=0.

Steg 2: Beräkna $x_{k+1} = f(x_k)$.

Steg 3: Finn det h som uppfyller att $2^h \leq k < 2^{h+1}$ och beräkna $j = 2^h - 1$.

Steg 4: Beräkna $\text{sgd}(x_k - x_j, N)$.

Om $\text{sgd}(x_k - x_j, N) = 1$ eller $\text{sgd}(x_k - x_j, N) = N$ öka k med 1 och gå till Steg 2, annars får vi att $\text{sgd}(x_k - x_j, N) = r$, det vill säga en faktor av N.

Valet att endast beräkna $\text{sgd}(x_k - x_j, N)$ då $2^h \leq k < 2^{h+1}$ och $j = 2^h - 1$ görs för att få bättre komplexitet. Vi kommer nedan välja f som ett polynom för att förenkla komplexitetsanalysen. Vi kommer även att anta att polynom modulo N beter sig approximativt som pseudoslumptal vilket kommer ge en bättre komplexitet men om antagandet är giltigt och om den komplexitet vi får reflekterar den förväntade komplexiteten är ett öppet problem [13].

Komplexiteten kommer diskuteras nedan men vi kommer då behöva en uppskattning på hur många fler x_k vi kommer behöva beräkna med denna metod, det vill säga innan vi hittar k, j sådant att $x_k \equiv x_j \pmod r$ om vi endast testar $j = 2^h - 1$ för varje $k : 2^h \leq k < 2^{h+1}$ jämfört med att testa $\forall j < k$. Vi kommer visa att om k är det minsta tal som löser problemet då man testar med $j = 2^h - 1$ och k_0 är det minsta tal som löser problemet då man testar med all $j < k$ så är $k < 4k_0$.

Detta betyder att k är det minsta talet sådant att $x_k \equiv x_j \pmod r$ där $2^h \leq k < 2^{h+1}$, $j = 2^h - 1$ och k_0 är det minsta tal sådant att $\exists j_0 : x_{k_0} \equiv x_{j_0} \pmod r$. Innan vi gör detta behöver vi ett kort lemma.

Lemma A.12.0.2. *Låt $f^k(x)$ vara notation för $\underbrace{f \circ ... \circ f}_{k \text{ times}}(x)$.*

Om $f^k(x) = f^j(x)$ så är $f^{k+a}(x) = f^{j+a}(x)$ $\forall a \in \mathbb{N}$.

Bevis. Vi gör ett enkelt induktions bevis. För $a = 1$ får vi

$$f^{k+1}(x) = f(f^k(x)) = f(f^j(x)) = f^{j+1}(x).$$

Om vi antar att $f^{k+a}(x) = f^{j+a}(x)$ så får vi att

$$f^{k+(a+1)}(x) = f^{k+a+1}(x) = f(f^{k+a}(x)) = f(f^{j+a}(x)) = f^{j+a+1}(x) = f^{j+(a+1)}(x).$$

Per induktion får vi då att $f^{k+a}(x) = f^{j+a}(x)$ $\forall a \in \mathbb{N}$. □

Enligt s.140-141 i [2] gäller det även att $f^k(x) \equiv f^j(x) \pmod r$ och f är ett polynom ger $f^{k+a}(x) \equiv f^{j+a}(x) \pmod r$ $\forall a \in \mathbb{N}$. Vi formulerar uppskattningen nedan.

Proposition A.12.1. *Låt $k = \min_{k \in \mathbb{N}} k : \exists j, h \in \mathbb{N} : 2^h \leq k < 2^{h+1}, j = 2^h - 1, x_k \equiv x_j \pmod r$. Låt sedan $k_0 = \min_{k \in \mathbb{N}} k : \exists j \in \mathbb{N} : x_k \equiv x_j \pmod r, j < k$. Slutligen om vi har att $x_{k+1} \equiv f(x_k) \pmod N$ med samma notation som i Lemma A.12.0.2, $f : \mathbb{Z} \to \mathbb{Z}$ är ett polynom, $x_0 \in \mathbb{Z}$, $N = pr$ och $r, N, p \in \mathbb{N}$.*

Då gäller att $k < 4k_0$.

Bevis. Låt j_0 vara det $j_0 \in \mathbb{N}$ sådant att $x_{k_0} \equiv x_{j_0} \pmod{r}$. Låt även $h \in \mathbb{N} : 2^h \leq k_0 < 2^{h+1}$. Betrakta $k' = j + k_0 - j_0$, med $j = 2^{h+1} - 1$. Vi har nu att

$$k' = j + k_0 - j_0 = 2^{h+1} - 1 + k_0 - j_0.$$

Vi använder att $k_0 > j_0$ och $k_0, j_0 \in \mathbb{N}$ ger $k_0 - j_0 \geq 1$ och får

$$k' = 2^{h+1} - 1 + k_0 - j_0 \geq 2^{h+1} - 1 + 1 = 2^{h+1}.$$

Vi har även att

$$k' = 2^{h+1} - 1 + k_0 - j_0 < 2^{h+1} + k_0 < 2^{h+1} + 2^{h+1} = 2 \cdot 2^{h+1} = 2^{h+2}$$

Nu ser vi att $2^{h+1} \leq k' < 2^{h+2}$ och $j = 2^{h+1} - 1$.

Vi vill nu visa att $x_{k'} \equiv x_j \pmod{r}$. Vi ser att

$$f^{k'}(x_0) = f^{j+k_0-j_0}(x_0) = f^{(j-j_0)+k_0}(x_0).$$

Vi använder nu att $f^k(x) \equiv f^j(x) \pmod{r}$ och f är ett polynom ger $f^{k+a}(x) \equiv f^{j+a}(x) \pmod{r}$ $\forall a \in \mathbb{N}$ (se kommentaren efter Lemma A.12.0.2 eller s.140-141 i [2]). Detta ger att

$$\begin{aligned} &f^{(j-j_0)+k_0}(x_0) \equiv f^{(j-j_0)+j_0}(x_0) \pmod{r}, \\ &f^{(j-j_0)+j_0}(x_0) = f^j(x_0) \end{aligned}$$

så $f^{k'}(x_0) = f^j(x_0)$ vilket ger att $x_{k'} \equiv x_j \pmod{r}$ vilket var det vi ville.

Eftersom k' uppfylller alla krav för k och eftersom k är det minsta tal som uppfyller de kraven så har vi att $k \leq k'$. Slutligen får vi att $k \leq k' < 2^{h+2} = 4 \cdot 2^h < 4k_0$. □

Vi kommer även behöva en proposition.

Proposition A.12.2. *Låt S vara en mängd med $|S| = r$ samt $f : S \to S$, $x_0 \in S$ och $x_{i+1} = f(x_i)$. Låt även $l \in \mathbb{N}, l = 1 + \lfloor\sqrt{2\lambda r}\rfloor$. Då gäller att sannolikheten att $(f, x_0), (f, x_1), ..., (f, x_l)$ alla är distinkta är mindre än $e^{-\lambda}$.*

Bevis. Om vi undersöker på hur många sätt man kan välja ett x_i, om vi inte kräver att de ska vara distinkt, så ser vi att det är r sätt eftersom $|S| = r$. Vi kan välja f på r^r sätt eftersom vi för varje x sådant att $f(x) = y$ så finns det r stycken y eftersom $|S| = r$. Vi får nu att de totala antalet kombinationer av par på formen (f, x_i) är r^{r+1}.

Om vi istället undersöker på hur många sätt vi kan välja distinkta x_i så får vi att x_0 kan väljas fritt, r sätt, x_1 kan väljas på $r - 1$ sätt eftersom vi inte kan välja x_0, om vi fortsätter detta resonemang så ser vi att x_k kan väljas på $r - k$ sätt. Detta bestämmer hur f måste mappa sina första l tal men de övriga kan mappas på godtyckligt sätt, vilket ger att f här kan väljas på r^{r-l} sätt. Detta ger att

$$\text{P(Alla par är distinkta)} = \frac{r^{r-l} \prod_{k=0}^{l} r - k}{r^{r+1}} = r^{-l-1} \prod_{k=0}^{l} r - k = \prod_{k=1}^{l} 1 - \frac{k}{r}.$$

Vi använder ett resultat från envariabelanalysen och minns att $\log(1 - x) < -x$ $\forall x \in (0, 1)$ samt regler för summor. Vi använder detta till att

$$\log(\text{P(Alla par är distinkta)}) = \log\left(\prod_{k=1}^{l} 1 - \frac{k}{r}\right) = \sum_{k=1}^{l} \log\left(1 - \frac{k}{r}\right) < \sum_{k=1}^{l} -\frac{k}{r}$$

En omskrivning från envariabel analysen ger att summan kan skrivas på formen

$$\sum_{k=1}^{l} -\frac{k}{r} = \frac{-l\,(l+1)}{2r} = \frac{-l^2 - l}{2r} < \frac{-l^2}{2r}$$

Vi lägger in formeln för l och erhåller

$$\frac{-l^2}{2r} = \frac{-\left(1+\lfloor\sqrt{2\lambda r}\rfloor\right)^2}{2r} < \frac{-\left(\sqrt{2\lambda r}\right)^2}{2r} = \frac{-2\lambda r}{2r} = -\lambda$$

och eftersom e^x är en växande funktion ger $\log(\text{P(Alla par är distinkta)}) < -\lambda$ att
P(Alla par är distinkta) $< e^{-\lambda}$ □

Vi kan nu skriva satsen som knyter ihop allting.

Sats A.12.3. *Låt $N = pr$ där p, r är primtal och $r < \sqrt{N}$. Låt även $f : \mathbb{Z}_N \to \mathbb{Z}_N$ vara ett polynom samt $x_0 \in \mathbb{Z}_N$, $x_{i+1} = f(x_i)$ då kommer Pollards rho-algoritm sannolikt hitta r med komplexitet $\mathcal{O}\left(\sqrt[4]{N}\log^3(N)\right)$, detta är med en sannolikhet större än $1 - e^{-\lambda}$ för ett λ som vi förenkelhetens skull lämnar till bevisdelen.*

Bevis. Varje steg kommer vi göra en beräkning av en sgd och en beräkning av $f(x)$. Vi vet från teorin att dessa har komplexitet $\mathcal{O}\left(\log^3(N)\right)$ respektive $\mathcal{O}\left(\log^2(N)\right)$ vilket ger att varje steg tar $C_1\log^3(N)+C_2\log^2(N)$ för några $C_1, C2$. Om vi antar att algoritmen tar k steg och att k_0 är första gången vi hade hittat r om vi testade att ta sgd mellan N och differensen av alla kombinationer av x_i, x_j, så ger oss Proposition A.12.1 att $k < 4k_0$. Detta gör att den totala tiden blir

$$k\left(C_1\log^3(N) + C_2\log^2(N)\right) < 4k_0\left(C_1\log^3(N) + C_2\log^2(N)\right).$$

Vi väljer ett fixt tal l som en uppskattning av vårt k_0 om vi antar att (f, x_0) valdes slumpmässigt och definierar nu λ sådant att $l = 1+\lfloor\sqrt{2\lambda r}\rfloor$. Per definition av k_0 så är alla x_i distinkta för $i < k_0$. Med hjälp av Proposition A.12.2 får vi då att sannolikheten att $k_0 > l$ är mindre än $e^{-\lambda}$. Om vi därför betraktar de fall då $k_0 \leq l = 1 + \lfloor\sqrt{2\lambda r}\rfloor$ så får vi att

$$4k_0\left(C_1\log^3(N) + C_2\log^2(N)\right) \leq 4\left(1+\lfloor\sqrt{2\lambda r}\rfloor\right)\left(C_1\log^3(N) + C_2\log^2(N)\right)$$

Vi vet att $\lfloor\sqrt{2\lambda r}\rfloor < \sqrt{2}\sqrt{\lambda}\sqrt{r}$ och om vi sätter in det får vi

$$4\left(1+\lfloor\sqrt{2\lambda r}\rfloor\right)\left(C_1\log^3(N) + C_2\log^2(N)\right) < 4\left(1+\sqrt{2}\sqrt{\lambda}\sqrt{r}\right)\left(C_1\log^3(N) + C_2\log^2(N)\right)$$

Vi använder nu att $r < \sqrt{N}$ och får

$$4\left(1+\sqrt{2}\sqrt{\lambda}\sqrt{r}\right)\left(C_1\log^3(N) + C_2\log^2(N)\right) < 4\left(1+\sqrt{2}\sqrt{\lambda}\sqrt[4]{N}\right)\left(C_1\log^3(N) + C_2\log^2(N)\right)$$

Vi använder slutligen att $\left(1+\sqrt{2}\sqrt{\lambda}\sqrt[4]{N}\right) < 2\sqrt{2}\sqrt{\lambda}\sqrt[4]{N}$ samt att $\log^2(N) < \log^3(N)$ för att få ut rätt konstant term. Vi har nu att

$$4\left(1+\sqrt{2}\sqrt{\lambda}\sqrt[4]{N}\right)\left(C_1\log^3(N) + C_2\log^2(N)\right) < 8\sqrt{2}\sqrt{\lambda}\left(C_1+C_2\right)\sqrt[4]{N}\log^3(N).$$

Det är nu möjligt att välja ett C så att $8\sqrt{2}\sqrt{\lambda}\left(C_1+C_2\right) < C$ vilket ger komplexiteten $\mathcal{O}\left(\sqrt[4]{N}\log^3(N)\right)$ med en sannolikhet större än $1 - e^{-\lambda}$, vilket var vad vi ville visa.

□

A.13 Komplexitet för att finna ett matchande element mellan två listor

Antag att vi har två listor innehållande heltal, x respektive y. Låt x_i vara det i:te elementet i x och y_j vara j:te elementet i y. Låt även n vara antalet element i x och y. Vi vill hitta de index i och j för vilka $x_i = y_j$. Detta kan göras med att man jämföra alla element i x med alla element i y men detta kommer ge komplexiteten $\mathcal{O}\left(n^2\right)$. Vi skriver här en kort algoritm som löser detta med komplexitet $\mathcal{O}\left(n\log\left(n\right)\right)$.

Steg 1: Sortera x och y.

Steg 2: Sätt $i = 0$ och $j = 0$.

Steg 3: Om $i \geq n$ eller $j \geq n$ så finns inga element i x och y som är lika och vi är klara. Annars gå till Steg 4.

Steg 4: Om $x_i = y_j$ så är vi klara. Annars gå till Steg 5.

Steg 5: Om $x_i < y_j$ öka i med 1 gå till Steg 3. Om $x_i > y_j$ öka j med 1 gå till Steg 3.

För att få en bättre komplexitet sorterar vi först båda listorna, vilket kan göras med komplexitet $\mathcal{O}(n \log(n))$, se Quicksort [27] eller Mergesort [28]. För att nu undersöka om några element är lika så börjar vi med var sitt index på de minsta elementen i listorna och sedan ökar vi med ett på det indexet som pekar på de minsta av de värden som indexen pekar på. Detta upprepas tills de värden indexen pekar på är lika eller tills det ena indexet blir större än storleken på listan. Om det finns element som är lika så kommer vi hitta dem eftersom algoritmen kommer nå en av dem och då kan det indexet inte passera innan algoritmen har hittat indexet i den andra listan. Här kommer vi passera varje element i båda listorna max en gång och får då $2n$ som det största antal steg som krävs. Den slutliga komplexiteten för att finna de matchande elementen blir $\mathcal{O}(n \log(n)) + \mathcal{O}(n) = \mathcal{O}(n \log(n))$.

Notera att implementeringen av Mergesort till stor del baseras på [29].

A.14 Komplexitet för Dixons algoritm

Följande uträkningar och resonemang är förkortade och baseras på teorin i Koblitz bok [2, s.148-153]. Vi delar in komplexiteten för Dixons algoritm i tre steg. Det första steget är att finna slumpmässiga $a_i \in [1, N]$ sådana att a_i^2 mod N kan skrivas som en produkt av primtal mindre än y tills vi har r stycken sådana. Det andra steget är att finna linjärt beroende rader i en matris av storleken $d \times d + 1$ där $r = d + 1$. Matrisen består av nollor och ettor för att få en kongruens på formen $a_i^2 \equiv c_i \bmod N$. Det tredje är att upprepa steg 1 och steg 2 tills vi funnit ett a_i sådant att $\text{sgd}(a + \sqrt{c}, N)$ och $\text{sgd}(a - \sqrt{c}, N)$ ger två icketriviala lösningar.

Vi börjar med komplexiteten för första steget. Låt $u = \frac{\log(N)}{\log(y)}$ där N är talet vi vill faktorisera och y är de högst tillåtna primtalen. Om N är ett $r-$bit tal och y ett $s-$bit tal så är $u \approx \frac{r}{s}$. Det tar en fix tid att slumpa ett tal b_i mellan 1 och N, alltså $\mathcal{O}(r)$. Därefter ska vi beräkna b_i^2 mod N som har en komplexitet på $\mathcal{O}(r^2)$. Nu måste vi dividera b_i^2 mod N med alla primtal $\leq y$. Att dividera ett $\leq r-$bit heltal med ett $\leq s-$bit heltal ger komplexiteten $\mathcal{O}(rs)$. Varje test av ett slumpmässigt b_i har alltså komplexiteten $\mathcal{O}(rsy)$. Vi kommer att testa $u^u(\pi(y) + 1)$, där $\pi(y)$ är antalet primtal $< y$, värden på b_i innan vi hittar $\pi(y) + 1$ stycken b_i^2 som är en produkt av primtal $\leq y$. Enligt Sats A.6.1 vet vi att $\pi(y) \approx \frac{y}{\log(y)} = \mathcal{O}(\frac{y}{s})$. Hela komplexiteten för steg ett blir därmed $\mathcal{O}(u^u \frac{y}{s} \cdot rsy) = \mathcal{O}(u^u y^2 r)$.

I steg två säger vi inte så mycket mer än att de operationer som görs är polynomiella för y och r eftersom operationerna som görs är radreducering samt att finna $a_i^2 \equiv c_i \bmod n$. Det ger $\mathcal{O}(y^j r^h)$ för lämpliga heltal j och h.

Steg tre är att upprepa de två första stegen tills vi finner en icketrivial faktor. Vi behöver därför uppskatta komplexiteten

$$\mathcal{O}(\alpha(u^u r y^2 + y^j r^h)) = \mathcal{O}(u^u y^j r^h) = \mathcal{O}((\frac{r}{s})^{\frac{r}{s}} e^{ks} r^h) \tag{A.14.1}$$

för några heltal h, k och α. Vi har nu att r är antalet bit i N som är fixt, vårt problem blir därmed att hitta s så att vi minimerar (A.14.1). Eftersom r är fixt ska vi minimera $(\frac{r}{s})^{\frac{r}{s}} e^{ks}$ eller ekvivalent att minimera dess logaritm. Vi får att

$$\frac{d}{ds}(\frac{r}{s} \log \frac{r}{s} + ks) = -\frac{r}{s^2}(\log \frac{r}{s} + 1) + k \approx -\frac{r}{s^2} \log \frac{r}{s} + k = 0. \tag{A.14.2}$$

Vi vill alltså hitta s så att $ks \approx \frac{r}{s} \log \frac{r}{s} + ks$. Eller om vi går tillbaka till vårt ordinarie problem så att $(\frac{r}{s})^{\frac{r}{s}}$ och e^{ks} är ungefär lika stora. Eftersom k är konstant så är s^2 enligt (A.14.2) av samma storlek

som $r \cdot \log\left(\frac{r}{s}\right) = r(\log(r) - \log(s))$. Detta betyder att s är av storlek mellan $\sqrt{r}$ och $\sqrt{r \cdot \log\left(\frac{r}{s}\right)}$. Vi kan nu använda approximationen $\log(s) \approx \frac{1}{2}\log(r)$ och med (A.14.2) får vi att

$$-\frac{r}{2s^2}\log(r) + k \approx 0. \tag{A.14.3}$$

Ekvationen (A.14.3) ger att $s \approx \sqrt{\frac{r}{2k}\log(r)}$. Eftersom $(\frac{r}{s})^{\frac{r}{s}}$ och e^{ks} är ungefär lika stora så uppskattas (A.14.1) till $\mathcal{O}(e^{2ks}) = e^{\sqrt{2k}\cdot\sqrt{r\cdot\log(r)}}$. Vi sätter $\sqrt{2k} = C$ och vi har att $r = \mathcal{O}(\log(N))$ alltså blir den slutgiltiga komplexiteten $e^{C\cdot\sqrt{\log(N)\cdot\log(\log(N))}}$.

A.15 Förbättringar Kedjebråksmetoden

I vår implementation av Kedjebråksmetoden har vi valt att förbättra algoritmen med redan kända delalgoritmer.

A.15.1 Legendresymbolen

Definition A.15.1. Låt N vara ett positivt heltal relativt prim med ett primtal p. Definiera Legendresymbolen $(\frac{N}{p})$ enligt:

$$(\tfrac{N}{p}) = \begin{cases} 1 & \text{om } \exists\ x \text{ s.a } x^2 \equiv N \pmod{p}, \\ 0 & \text{om } p|N, \\ -1 & \text{om } x^2 \not\equiv N \pmod{p},\ \forall x. \end{cases}$$

I primtalsbasen används endast primtal p sådana att $(\frac{N}{p}) = 1$.[[12]sid. 472]

A.15.2 Övre gräns till primtalsbas

Upptäcker vi i faktoriseringen av Q_n att den innehåller primtalsfaktorer större än en vald övre gräns bortser vi från Q_n. I resultatdelen återges de tiderna som, enligt våra experiment, motsvaras av de bästa värdena på den övre gränsen.

B Kvantteori

B.1 Dirac-notation

Dirac-notation används huvudsakligen i fysik. Vi noterar en godtycklig vektor v som $|v\rangle$ och dess konjugat v^* som $\langle v|$.

B.2 Unitära matriser

En matris kallas *unitär* om $U^{-1} = U^*$. I valfri linjär algebra-bok kan läsaren konstatera att följande egenskaper är ekvivalenta:

1. U är unitär,
2. U bevarar normen, så $\|U|v\rangle\| = \||v\rangle\|$, samt att
3. kolumnerna i U skapar en ortonormal bas.

Ovanstående egenskaper används löpande genom texten.

B.3 Tensorprodukt

Låt $A = (A_{i,j})$ vara en $m_1 \times n_1$-matris och B en $m_2 \times n_2$-matris. Deras *tensorprodukt* är definierad som $m_1 m_2 \times n_1 n_2$-matrisen

$$A \otimes B = \begin{pmatrix} A_{11}B & \cdots & A_{1n_1}B \\ \vdots & \ddots & \vdots \\ A_{m_1 1}B & \cdots & A_{m_1 n_1}B \end{pmatrix}.$$

Följande räkneregler är enkla att bekräfta:

1. $c(A \otimes B) = (cA) \otimes B = A \otimes (cB)$ för godtycklig konstant c,
2. $(A \otimes B)^* = A^* \otimes B^*$,
3. $A \otimes (B + C) = (A \otimes B) + (A \otimes C)$,
4. $A \otimes (B \otimes C) = (A \otimes B) \otimes C$,
5. $(A \otimes B)(C \otimes D) = (AC) \otimes (CD)$.

B.4 Bevis för fouriertransform

B.4.1 F_N är en unitär matris

Det är lätt att se att F_N är en unitär matris ifrån vår Definition i 6.3.1, ty normen hos varje kolumn är 1, och alla kolumner är ortogonala ty

$$\sum_{j=0}^{N-1} \frac{1}{\sqrt{N}} w_N^{jk_1} \frac{1}{\sqrt{N}} w_N^{-jk_2} = \frac{1}{N} \sum_{j=0}^{N-1} w_N^{j(k_1-k_2)} = \begin{cases} 1 \text{ if } k_1 = k_2 \\ 0 \text{ if } k_1 \neq k_2 \end{cases}.$$

B.4.2 Snabb fouriertransform

Att räkna ut fouriertransformen med matrismultiplikationen i 6.3.1 är tidskrävande, för varje element i $\hat{v}$ krävs $\mathcal{O}(N)$ steg, och med sammanlagt N element har vi $\mathcal{O}(N)$ i komplexitet. Därför ska vi beskriva ett mycket snabbare tillvägagångssätt, den snabba fouriertransformen, med komplexitet $\mathcal{O}(N \log N)$. Vi antar först och hädanefter att $N = 2^n$ för något $n \in \mathbb{N}$, vilket alltid är möjligt eftersom att vi kan fylla ut skillnaden med nollor i en vektor annars. Metoden börjar med en insikt från följande system av ekvationer:

$$\begin{aligned} \hat{v}_j &= \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} w_N^{jk} v_k \\ &= \frac{1}{\sqrt{N}} \left(\sum_{\text{jämna } k} w_N^{jk} v_k + w_N^j \sum_{\text{udda } k} w_N^{j(k-1)} v_k \right) \\ &= \frac{1}{\sqrt{2}} \left(\frac{1}{\sqrt{2N}} \sum_{\text{jämna } k} w_N^{jk} v_k + w_N^j \frac{1}{\sqrt{2N}} \sum_{\text{udda } k} w_N^{j(k-1)} v_k \right) \\ &= \frac{1}{\sqrt{2}} \left(\frac{1}{\sqrt{2N}} \sum_{\text{jämna } k} w_{N/2}^{jk/2} v_k + w_N^j \frac{1}{\sqrt{2N}} \sum_{\text{udda } k} w_{N/2}^{j(k-1)/2} v_k \right) \end{aligned}$$

Vi ser här att med en mild justering så kan vi beräkna transformen rekursivt. Vi börjar med att räkna ut de $N/2$-dimensionella transformerna av vektorn som bildas av de jämna indexen i v ($\widehat{v_{\text{jämna}}}$) samt av vektorn som bildas av de udda indexen ($\widehat{v_{\text{udda}}}$). Sedan ser vi att

$$\hat{v}_j = \frac{1}{\sqrt{2}} ((\widehat{v_{\text{jämna}}})_j + w_N^j (\widehat{v_{\text{udda}}})_j).$$

Den lilla justeringen vi gör nu för att göra denna ekvation väldefinierad är att $(\widehat{v_{\text{jämna}}})_{j+N/2} = (\widehat{v_{\text{jämna}}})_j$ och motsvarande för $\widehat{v_{\text{udda}}}$. Eftersom att vi för varje rekursivt steg beräknar $N/2, \ldots, N/2^n$ och $N = 2^n$ måste vi göra $\mathcal{O}(n)$ räkningar för varje element $\hat{v}_j$. N element i $\hat{v}$ ger oss en komplexitet $\mathcal{O}(Nn) = \mathcal{O}(N \log N)$ för snabb fouriertransform.

B.5 Kvantfouriertransform

Transformen kommer konstrueras av Hadamard- och fasskiftsgrindar för att ge oss denna transform. Vi kan få en insikt om hur detta skulle gå till genom att undersöka följande

$$\begin{aligned}
F_N |k\rangle &= \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi i jk/2^n} |j\rangle \\
&= \{\text{Märk att } j/2^n = \sum_{l=1}^{n} j_l 2^{-l}, \text{ där } j_l \text{ representerar siffrorna i } j\} \\
&= \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi i k(\sum_{l=1}^{n} j_l/2^l)} |j_1 \ldots j_n\rangle \\
&= \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \prod_{l=1}^{n} e^{2\pi i k j_l/2^l} |j_1 \ldots j_n\rangle \\
&= \frac{1}{\sqrt{N}} \bigotimes_{l=1}^{n} \left(|0\rangle + e^{2\pi i k/2^l} |1\rangle\right) \\
&= \bigotimes_{l=1}^{n} \frac{1}{\sqrt{2}} \left(|0\rangle + e^{2\pi i k/2^l} |1\rangle\right)
\end{aligned}$$

För att förstå varför det blir en tensorprodukt i slutändan kan man titta närmre på summan av produkter i tredje steget och vilka amplituder som genereras. Minns också att $|j_1 \ldots j_n\rangle = |j_1\rangle \otimes \cdots \otimes |j_n\rangle$. För att få konkreta exempel på hur en krets för ovanstående utvärden skulle kunna se ut, se [19, sid. 47-49].

B.6 Sannolikhet för lyckat val av x i Shors algoritm

Sats B.6.1. *Anta att sgd(x, N)= 1. Sannolikheten att ett sådant slumpvalt x har en jämn period r modulo N och uppfyller $x^{r/2} \not\equiv \pm 1 \pmod{N}$ är minst $\frac{3}{8}$.*

Bevis. Vi börjar med att visa sannolikheten för att x har en jämn period. För att göra detta påminner vi oss om Kinesiska restklassatsen (se Appendix A.1), som antyder att välja ett slumpvalt x modulo N är ekvivalent med att välja ett slumpvalt x_1 modulo p och ett slumpvalt x_2 modulo q. De kommer ha perioder r_1 respektive r_2, och det är tydligt att $r_1, r_2 | r$. Eftersom att grupper med multiplikation modulo primtal är cykliska vet vi att det finns generatorer g_1 och g_2 till grupperna. Därav är $x_1 \equiv g_1^{k_1} \pmod{p}$ för något k_1 och $x_2 \equiv g_2^{k_2} \pmod{q}$ för något k_2. Från Fermats lilla sats vet vi också att $a^{p-1} \equiv 1 \pmod{p}$ och motsvarande för q ty de är primtal. Vi har att

$$x_1^{p-1} \equiv x_1^{r_1} \equiv g_1^{k_1 r_1} \equiv 1 \pmod{p}$$

och liknande för x_2. I ekvationen ovan ser vi att $p - 1 | k_1 r_1$ och $q - 1 | k_2 r_2$. Vi kan anta p och q udda, annars är N redan väldigt lätt att faktorisa. Då har vi att r_1 är jämn om k_1 udda, och r_1 är jämn om k_2 udda. Eftersom att x_1 och x_2 är slumpvalda är också k_1 och k_2 slumpvalda, så sannolikheten att någon av perioderna är jämn är $\frac{3}{4}$. Därför är sannolikheten att x har en jämn period $\frac{3}{4}$.

Anta nu r jämn. Vi vill visa att $P(x^{r/2} \equiv \pm 1 \pmod{N}) \leq \frac{1}{2}$. Detta är fallet eftersom att vi har totalt två rötter till $x^r \equiv 1 \pmod{p}$ och två rötter till $x^r \equiv 1 \pmod{q}$. Dessa är ± 1 modulo p och ± 1 modulo q. Enligt Kinesiska restklassatsen (se Appendix A.1) har vi därför fyra rötter till 1 modulo N, varav två av dem uppfyller $x^{r/2} \not\equiv \pm 1 \pmod{N}$. Därför är den totala sannolikheten för ett lyckat val av x minst $\frac{3}{8}$. □

B.7 Sannolikhet för sgd$(c, r)= 1$

Låt $\varphi(r)$ vara Eulers fi-funktion. Det vill säga $\varphi(r)$ är antalet heltal mindre än r som är relativt prima med r.

Sats B.7.1. *Om $r \geq 3$ har vi att:*

$$\frac{r}{\varphi(r)} < e^{\gamma} \log\log(r) + \frac{2.50637}{\log\log(r)} \tag{B.7.1}$$

där $\gamma = 0.5772156649\ldots$ är Eulers konstant.

Bevis. Se [30] □

Lemma. *Om $r \geq 19$ är sannoliketen att r och slumpvalt c relativt prima minst $\frac{1}{4\log\log(N)}$.*

Bevis. Beviset följer direkt från B.7.1 för $r \geq 19$:

$$\frac{r}{\varphi(r)} < 4\log\log(r). \tag{B.7.2}$$

Därför har vi

$$\frac{\varphi(r)}{r} > \frac{1}{4\log\log(r)} > \frac{1}{4\log\log(N)} \tag{B.7.3}$$

vilket vi ville bevisa. □

B.8 Bevis av lemma

Lemma. *Låt $N \geq 100$. Då är sannolikheten att observera ett b i (6.4.1) sådant att $|br - cQ| \leq \frac{r}{2}$ inte mindre än $\frac{1}{5}$.*

Bevis.

$$\begin{aligned} P(b) &= \frac{1}{mQ}\left|\sum_{j=0}^{m-1} e^{\frac{2\pi i b j r}{Q}}\right|^2 = \frac{1}{mQ}\left|\frac{1-e^{\frac{2\pi i b m r}{Q}}}{1-e^{\frac{2\pi i b r}{Q}}}\right|^2 = \frac{1}{mQ}\left|\frac{\sin^2\left(\frac{\pi b m r}{Q}\right)}{\sin^2\left(\frac{\pi b r}{Q}\right)}\right| \\ &= \frac{1}{mQ}\left|\frac{\sin^2\left(\frac{\pi m(br-cQ)}{Q}\right)}{\sin^2\left(\frac{\pi(br-cQ)}{Q}\right)}\right| \end{aligned}$$

på grund av periodiciteten hos $\sin^2$. Då vi antar att $br - cQ \in [-\frac{r}{2}, \frac{r}{2}]$ kan vi analysera funktionen

$$f(x) = \frac{\sin^2\left(\frac{\pi m x}{Q}\right)}{\sin^2\left(\frac{\pi x}{Q}\right)}$$

på intervallet. Vi kan konstatera att f är jämn och att den har minimipunkter i ändpunkterna. Detta motiveras i Appendix B.9. Vi har också på grund av definitionen av m följande olikhet

$$\frac{\pi}{2}\left(1 - \frac{r}{Q}\right) < \frac{\pi m r}{2Q} < \frac{\pi}{2}\left(1 + \frac{r}{Q}\right).$$

Detta tillsammans med minimipunkterna hos f och det faktum att $\sin(x)$ är symmetrisk runt $x = \frac{\pi}{2}$ ger oss

$$f(x) \geq \frac{\sin^2\left(\frac{\pi}{2}\left(1 - \frac{r}{Q}\right)\right)}{\sin^2\left(\frac{\pi r}{2Q}\right)}.$$

Eftersom att Q är stort relativt till r (ty $r < N$, och $Q \geq N^2$) är $\frac{r}{Q}$ försumbart, och vi kan använda oss av olikheterna $\sin(x) \leq x$ samt att $\sin^2(\frac{\pi}{2}(1+x)) \geq 1 - (\frac{\pi}{2}x)^2$ (tydligt i Taylor-utvecklingen av $\sin^2(x)$). Detta ger oss att

$$f(x) \geq \frac{4Q^2}{\pi^2 r^2}\left(1 - \frac{\pi^2 r^2}{4Q^2}\right).$$

Termen $\left(1 - \frac{\pi^2 r^2}{4Q^2}\right)$ går mot 1 då Q växer, och är större än 0.9999 då $N \geq 100$. Vi har att

$$P(b) \geq \frac{1}{mQ}\frac{4Q^2}{\pi^2 r^2} > \frac{4Q}{\pi^2 m r^2} > \frac{4(r-1)}{\pi^2 r^2} = \frac{4}{\pi^2}\left(1 - \frac{1}{r}\right) \geq \frac{2}{\pi^2 r} > \frac{1}{5r}, \tag{B.8.1}$$

då $r \geq 2$. Men som tidigare nämnt finns det precis r stycken b som uppfyller $|br - cQ| \leq \frac{r}{2}$. Den totala sannolikheten är alltså större än $\frac{1}{5}$. Märk också att gränsvärdet av (B.8.1) är $\frac{2}{5}$ då r växer, samt att vi tidigare i Appendix B.7 antagit att $r \geq 19$. □

B.9 Analys av $f(x) = \frac{\sin^2\left(\frac{\pi m x}{Q}\right)}{\sin^2\left(\frac{\pi x}{Q}\right)}$

Det är tydligt att $f(x)$ är en jämn funktion. Därav är det tillräckligt att motivera att $f'(x)$ är strikt negativ på intervallet $x \in (0, \frac{r}{2}]$ för att konstatera att $f(x)$ har ett maximum i $x = 0$ och lokala minimipunkter i $x = \pm\frac{r}{2}$. Vi har att

$$f'(x) = -\frac{2\pi \sin\left(\frac{\pi m x}{Q}\right)\left(\cos\left(\frac{\pi x}{Q}\right)\sin\left(\frac{\pi m x}{Q}\right) - m\sin\left(\frac{\pi x}{Q}\right)\cos\left(\frac{\pi m x}{Q}\right)\right)}{Q\sin^3\left(\frac{\pi x}{Q}\right)}.$$

Termen $Q\sin^3\left(\frac{\pi x}{Q}\right)$ är positiv på vårt givna intervall ty $0 < \frac{\pi x}{Q} \leq \frac{\pi r}{2Q} < \frac{\pi}{2}$. Liknande gäller för termen $2\pi\sin\left(\frac{\pi m x}{Q}\right)$ eftersom $0 < \frac{\pi m x}{Q} \leq \frac{\pi m r}{2Q} < \frac{\pi(Q+r)}{2Q} < \pi$. Låt

$$g(x) = \cos\left(\frac{\pi x}{Q}\right)\sin\left(\frac{\pi m x}{Q}\right) - m\sin\left(\frac{\pi x}{Q}\right)\cos\left(\frac{\pi m x}{Q}\right).$$

Kvar att visa är $g(x) > 0$ då $x \in (0, \frac{r}{2}]$. Det är lätt att se att $g(0) = 0$, och om vi studerar $g'(x)$ ser vi att

$$g'(x) = \frac{\pi\left(m^2 - 1\right)\sin\left(\frac{\pi x}{Q}\right)\sin\left(\frac{\pi m x}{Q}\right)}{Q} > 0, \quad x \in (0, \tfrac{r}{2}],$$

enligt samma argument som för de tidigare termerna. Därmed måste $g(x) > 0$, och därav vet vi att $f'(x) < 0$. Vi kan nu konstatera att $f(x)$ har lokala minimipunkter i ändpunkterna $x = \pm\frac{r}{2}$, samt att $f(x)$ har ett maximum då $x = 0$. Detta maximum är gränsvärdet

$$\lim_{x\to 0}\frac{\sin^2\left(\frac{\pi m x}{Q}\right)}{\sin^2\left(\frac{\pi x}{Q}\right)} = \lim_{x\to 0}\left(\frac{\frac{\sin\left(\frac{\pi m x}{Q}\right)}{\frac{\pi m x}{Q}}\cdot\frac{\pi m x}{Q}}{\frac{\sin\left(\frac{\pi x}{Q}\right)}{\frac{\pi x}{Q}}\cdot\frac{\pi x}{Q}}\right)^2 = m^2.$$

B.10 Komplexitet hos Shors algoritm

Följande resonemang gör Hirvensalo i [19, sid. 61]. Att beräkna $\mathrm{sgd}(x, N)$ med Euklides algoritm kräver $\mathcal{O}(\log^3(N))$ steg. Det är sedan möjligt att testa om x har en period mindre än 19 i $\mathcal{O}(\log^3(N))$ med klassiska algoritmer ($r \geq 19$ krävs för beviset för Lemma B.7). Vi kan sedan applicera Hadamard-transformen i $\mathcal{O}(\log(N))$ steg. Nästa steg är att beräkna $x^a \pmod N$, vilket vi kan göra i $\mathcal{O}(\log^3(N))$ steg. Därefter genomför vi en kvantfouriertransform med komplexitet $\mathcal{O}(\log^2(N))$. Sedan beräknar vi konvergenter i $\mathcal{O}(\log^3(N))$ steg och tillslut använder vi än en gång Euklides algoritm för att beräkna $\mathrm{sgd}(x^{r/2} \pm 1)$ med samma komplexitet som tidigare.

Den totala komplexiteten är därför $\mathcal{O}(\log^3(N))$, men sannolikheten för en lyckad algoritm är bara $\Omega(\frac{1}{\log\log(N)})$ (se Appendix B.7). Därav krävs $\mathcal{O}(\log^3(N)\log\log(N))$ steg för att vi ska hitta en icke-trivial faktor med god sannolikhet.

C Information kring test av algoritmer

C.1 Tal

Följande tal användes vid de olika testerna:

Diskreta logaritmproblemet, kap 3.

1: $p = 502217, g = 12653, g^x = 64641, x = 190447$
2: $p = 4129232533, g = 126543, g^x = 64641, x = 14183331086$
3: $p = 4129232533, g = 123134134, g^x = 1325278813, x = 200$
4: $p = 123456791, g = 123134134, g^x = 67277073, x = 12833413$
5 : $p = 357111317192329313741434753596167717379838997101103107109113127131137139149151157163167173179181191193197199211223227223211199197193191181179173167163157151149139137131127113109107103101978983797371676159534743413731292319171311753$
$g = 4864681684765461413798798128894981897351324987415648974513789465187456168941539874987984891897618741876541897894864897468465432121321235417865148987745618974655418974175987109487130946149087112089347190328471084712314135135678$
$g^x = 8741718528929393285841717417816578974564171714714178184178178298282858282858555885899489798465461421632318971774717817818718787178828289593631874514123156456789795841612335486749654178919851235498718528975679874894897654113245$6

Primtalstest, kap 4.

1: 12345678910987654321
2: 313119843606266222723931946804611259287100533754507329247768328563715123287679948437264517293404303260795541448480275059086587161357892405477908284669463639709544036228659629
3: 3131198436062662227239319468046112592871005337545073292477683285637151232876799484372645172934043032607955414484802750590865871613578924054779082846694636397095440362286596295

Faktoriseringsproblemet, kap 5.

1: $1613 * 2143 = 3456659$
2: $1609161918110111 * 1510553619999637 = 2430725360566787603875546029707$
3: $10000000000283 * 111161191111 = 1111611911141458617084413$
4: $502217 * 524287 = 263305844279$
5: $1109 * 1609161918110111 = 1784560567184113099$
6: 27533333333333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333
∗1340185579782030309029142285845485748073406778702270938755484147318382420338087834406
82895571418700565464025703849579654515540228005598707625170455799463758972671270988931
20428018580440395901554076504716679079958882921239092780465639984417258813167026084549
532849694731411468851408226830492748537014 91
= 3689977629666523450860238427027904093028779997360252651373433018949946263997535170733469058066394888902442841045991759820994540944420817749946359883011902163714215661228572491181115814589004894555730965325640015345764314496878888206209042885259891987848612638044615949381957757043058934669602420223800477138483808397553107376590968708150528422720585988730405519772368142860433144845324765383473448494819923998133764124943181400178747009109576337022931906604731865346994819744984277736400137056929203024065114533385275803956109913051501557167684228628437161972577231690838209950 3

C.2 Testbädd

All kod, förutom CFRAC och MR, testades på en gemensam dator:
CPU: Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz, endast en kärna användes av våra algoritmer.
RAM: 16GB
OS: GNU/Linux kernel 3.10
För koden: GCC version 4.8.5, GMP version 6.1.2.

CFRAC och MR:
CPU: Intel(R) Core(TM) i5-7300 CPU @ 2.5000GHz, endast en kärna.
RAM: 8GB
OS:Windows 10 64-bitars
.Net Framework 4.8 System.Numerics BigInteger
System.Security.Cryptography RNGCryptoServiceProvider

D Kod

D.1 Direkt metod för DLP

```
#include <stdio.h>
#include <gmp.h>


int main() {

    mpz_t gOfX, g, p, test, i;
    // Valj h, p, och g
    mpz_init_set_str(gOfX, "67277073", 10)
    mpz_init_set_str(p, "123456791", 10);
    mpz_init_set_str(g, "123134134", 10);
    mpz_init(test);
    mpz_init_set_ui(i,0);
    int done = 0;

    while (mpz_cmp(i,p)) {
        mpz_powm(test, g, i, p);
        if (!mpz_cmp(gOfX, test)) {
            printf("x=");
            mpz_out_str(stdout, 10, i);
            printf(".\n");

            mpz_clears(gOfX, g, p, test, i, NULL);
            return 0;
        }

        mpz_add_ui(i, i, 1);
    }

    mpz_clears(gOfX, g, p, test, i, NULL);
    printf("Nagot gick fel,\n");
    return -1;

}
```

D.2 Shanks Babystep-Giantstep

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gmp.h>
#include <stdint.h>
#include <inttypes.h>

//Needed for sorting.
struct Pair {
        mpz_t number;
        mpz_t index;
};

//Merge sort:
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(struct Pair* arr, uint64_t l, uint64_t m, uint64_t r)
{
    uint64_t i, j, k;
    uint64_t n1 = m - l + 1;
    uint64_t n2 =  r - m;

    /* create temp arrays */
    struct Pair* L = (struct Pair *) malloc(n1*sizeof(struct Pair));
    struct Pair* R = (struct Pair *) malloc(n2*sizeof(struct Pair));

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++){
        mpz_init(L[i].number);
        mpz_init(L[i].index);
        mpz_set(L[i].number, arr[l + i].number);
        mpz_set(L[i].index, arr[l + i].index);
    }
    for (j = 0; j < n2; j++) {
        mpz_init(R[j].number);
        mpz_init(R[j].index);
        mpz_set(R[j].number, arr[m + 1+ j].number);
        mpz_set(R[j].index, arr[m + 1+ j].index);
    }

    /* Merge the temp arrays back uint64_to arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (mpz_cmp(L[i].number, R[j].number) <= 0)
        {
            mpz_set(arr[k].number, L[i].number);
            mpz_set(arr[k].index, L[i].index);
            i++;
        }
        else
```

```
        {
            mpz_set(arr[k].number, R[j].number);
            mpz_set(arr[k].index, R[j].index);
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1)
    {
        mpz_set(arr[k].number, L[i].number);
        mpz_set(arr[k].index, L[i].index);
        i++;
        k++;
    }
    free(L);

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2)
    {
        mpz_set(arr[k].number, R[j].number);
        mpz_set(arr[k].index, R[j].index);
        j++;
        k++;
    }
        free(R);
}

/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(struct Pair* arr, uint64_t l, uint64_t r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        uint64_t m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}


//Algorithms that solve DLP.
//g^a=h (mod p)
//Note % gives the remainder under divition (think mod).

//The algorithm might give another than Alice secret key in ...
//... Diffie-Hellman if g is not a generator.
```

```
void Shanks_Babystep_Gigantstep_Algorithm(mpz_t ans, mpz_t g, ...
...mpz_t h, mpz_t p){
        //step 1:
        //n = 1 + sqrtl(p);
        mpz_t n;
        mpz_init(n);
        mpz_sqrt(n, p);
        mpz_add_ui(n, n, 1);

        mpz_t temp;
        mpz_init(temp);

        //step 2:
        struct Pair* List1 = (struct Pair *) ...
        ...malloc((mpz_get_ui(n)+1)*sizeof(struct Pair));
        mpz_init(List1[0].number);
        mpz_set_ui(List1[0].number,1);
        mpz_init(List1[0].index);
        mpz_set_ui(List1[0].index,0);

        mpz_t i;
        mpz_init_set_ui(i, 1);

        for(; mpz_cmp(i, n)<=0; mpz_add_ui(i, i, 1)){
                mpz_init(List1[mpz_get_ui(i)].number);
                mpz_set(temp, List1[mpz_get_ui(i)-1].number);
                mpz_mul(temp, temp, g);
                mpz_mod(temp, temp, p);
                mpz_set(List1[mpz_get_ui(i)].number, temp);
                mpz_set(List1[mpz_get_ui(i)].index, i);
        }

        //Steg 3:
        mpz_t gninv;
        mpz_init(gninv);
        mpz_sub_ui(temp, p, 2);
        mpz_powm(gninv, List1[mpz_get_ui(n)].number, temp, p);
        //Fran Fermats lilla sats.

        //Steg 4:
        struct Pair* List2 = (struct Pair *) ...
        ...malloc((mpz_get_ui(n)+1)*sizeof(struct Pair));
        mpz_init(List2[0].number);
        mpz_set(List2[0].number, h);
        mpz_init(List2[0].index);
        mpz_set_ui(List2[0].index,0);

        mpz_set_ui(i, 1);
        for(; mpz_cmp(i, n)<=0; mpz_add_ui(i, i, 1)){

                mpz_init(List2[mpz_get_ui(i)].number);
                mpz_set(temp, List2[mpz_get_ui(i)-1].number);
                mpz_mul(temp, temp, gninv);
                mpz_mod(temp, temp, p);
                mpz_set(List2[mpz_get_ui(i)].number, temp);
                mpz_set(List2[mpz_get_ui(i)].index, i);
```

```
        }

        //Step 5:
        mergeSort(List1, 0, mpz_get_ui(n)-1);
        mergeSort(List2, 0, mpz_get_ui(n)-1);

/*      printf("List 1:");
        mpz_set_ui(i, 0);
        for(; mpz_cmp(i, n)<=0; mpz_add_ui(i, i, 1)){
                printf("(");
                mpz_out_str(0, 10, List1[mpz_get_ui(i)].number);
                printf(",");
                mpz_out_str(0, 10, List1[mpz_get_ui(i)].index);
                printf("), ");
        }
        printf("\n\n");

        printf("List 2:");
        mpz_set_ui(i, 0);
        for(; mpz_cmp(i, n)<=0; mpz_add_ui(i, i, 1)){
                printf("(");
                mpz_out_str(0, 10, List2[mpz_get_ui(i)].number);
                printf(",");
                mpz_out_str(0, 10, List2[mpz_get_ui(i)].index);
                printf("), ");
        }
        printf("\n\n");*/

        mpz_set_ui(i, 0);
        mpz_t j;
        mpz_init_set_ui(j, 0);

        //Om ingen matchning.
        mpz_set_si(ans, -1);

        int foundAns=0;

        while(mpz_cmp(i, n)<0 && mpz_cmp(j, n)<0 && !foundAns){
                if(mpz_cmp(List1[mpz_get_ui(i)].number,...
                ...List2[mpz_get_ui(j)].number)==0){
                        //Steg 6:
                        //ans=List1[i].index+List2[j].index*n;
                        mpz_mul(temp, List2[mpz_get_ui(j)].index, n);
                        mpz_add(temp, temp, List1[mpz_get_ui(i)].index);
                        mpz_set(ans, temp);
                        foundAns=1;
                }
                else if(mpz_cmp(List1[mpz_get_ui(i)].number,...
                ...List2[mpz_get_ui(j)].number)<0){
                        mpz_add_ui(i, i, 1);
                }
                else{
                        mpz_add_ui(j, j, 1);
                }
        }
        free(List1);
```

```
        free(List2);
}

int main(){
        //A = g^x mod p = g^ans mod p (om ans =/= -1).
        mpz_t x;
        mpz_init_set_str(x, "373373123", 10);
        mpz_t g;
        mpz_init_set_str(g, "1234567", 10);
        mpz_t p;
        mpz_init_set_str(p, "123456789012345", 10);
        mpz_nextprime(p, p);
        printf("p: ");
        mpz_out_str(0, 10, p);
        printf("\n");

        mpz_t A;
        mpz_init(A);
        mpz_powm(A, g, x, p);
        mpz_t ans;
        mpz_init(ans);
        Shanks_Babystep_Gigantstep_Algorithm(ans, g, A, p);
        printf("ans: ");
        mpz_out_str(0, 10, ans);
        printf("\n");

return 0;
}
```

D.3 Direkt metod för primtalstest & faktorisering

```
#include <stdio.h>
#include <gmp.h>


int main() {

    mpz_t s, N, i;
    mpz_init(s);
    mpz_init_set_str(N, "17845605671841130 99", 10);      // Valj tal
    mpz_init_set_ui(i,3);                                 // satt i = 3
    mpz_sqrt(s,N);

    if (mpz_divisible_ui_p(N, 2)) {      // Kontrollera att n ar udda
        mpz_out_str(stdout, 10, N);
        printf(" ar jamnt, ej ett primtal.\n");
        return 0;
    }

    // Fortsatt tills vi nar kvadroten av N
    while (mpz_cmp(s,i) > 0) {

        if (mpz_divisible_p(N, i)) {
            mpz_out_str(stdout, 10, N);
            printf(" ar ej ett primtal, har faktor ");
            mpz_out_str(stdout, 10, i);
            printf(".\n");

            mpz_clears(s, N, i, NULL);
            return 0;
        }

        mpz_add_ui(i, i, 2);
    }

    mpz_out_str(stdout, 10, N);
    printf(" ar ett primtal.\n");

    mpz_clears(s, N, i, NULL);
    return 0;
}
```

D.4 Fermats faktoriseringsalgoritm

```
#include <stdio.h>
#include <gmp.h>
#include <time.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <mpfr.h>

void Fermat_Factorization(mpz_t ans, mpz_t n){
        mpz_t b;
        mpz_init(b);
        mpz_sqrt(b, n);

        //Test if b2=b^2.
        mpz_t b2;
        mpz_init(b2);
        mpz_mul(b2, b, b);

        //Testa om n=b^2.
        if(mpz_cmp(b2, n)==0){
                mpz_set(ans, b);
        }
        else{
                //b_i=sqrt(n)+i.
                mpz_add_ui(b, b, 1);

                mpz_t temp;
                mpz_init_set_ui(temp, 0);

                mpz_t a;
                mpz_init(a);

                mpz_t a2;
                mpz_init(a2);

                mpz_t diff;
                mpz_init(diff);

                while(1){

                        mpz_mul(temp, b, b);
                        mpz_mod(temp, temp, n);
                        mpz_sqrt(a, temp);
                        mpz_mul(a2, a, a);

                        //Testa om a^2=b^2 mod n.
                        if(mpz_cmp(a2, temp)!=0){
                                mpz_add_ui(b, b, 1);
                                continue;
                        }

                        //ans=gcd(b-a, n).
                        mpz_sub(diff, b, a);
                        mpz_gcd(ans, n, diff);
```

```
                        //If ans =/= 1 and ans =/= n then ans...
                        ...is a factor of n.
                        if(!(mpz_cmp_ui(ans, 1)==0 ...
                        ...|| mpz_cmp(ans, n)==0)){
                                break;
                        }
                        else{
                                //b_{i+1}=b_i+1.
                                mpz_add_ui(b, b, 1);
                        }
                }
        }
}

void General_Fermat_Factorization(mpz_t ans, mpz_t n, mpz_t k){

        mpz_t tmp;
        mpz_init(tmp);
        mpz_add_ui(tmp, k, 1);
        long kplus1 = mpz_get_ui(tmp);

        mpz_t tmp2;
        mpz_init(tmp2);
        if(mpz_root(tmp2, n, kplus1)){
                mpz_set(ans, tmp2);
        }
        else{
                mpz_t b;
                mpz_init(b);
                mpz_mul(b, k, n);
                mpz_sqrt(b, b);

                mpz_add_ui(b, b, 1);

                mpz_t temp;
                mpz_init_set_ui(temp, 0);

                mpz_t a;
                mpz_init(a);

                mpz_t a2;
                mpz_init(a2);

                mpz_t diff;
                mpz_init(diff);

                mpz_t kn;
                mpz_init(kn);
                mpz_mul(kn, n, k);

                while(1){
                        mpz_mul(temp, b, b);
                        mpz_mod(temp, temp, n);
                        mpz_sqrt(a, temp);
                        mpz_mul(a2, a, a);
```

```
                        if(mpz_cmp(a2, temp)!=0){
                                mpz_add_ui(b, b, 1);
                                continue;
                        }
                        mpz_sub(diff, b, a);
                        mpz_gcd(ans, n, diff);
                        if(!(mpz_cmp_ui(ans, 1)==0 || ...
                        ...mpz_cmp(ans, n)==0)){
                                break;
                        }
                        else{
                                mpz_add_ui(b, b, 1);
                        }
                }
        }
}

int main(){
        mpz_t ans;
        mpz_init(ans);

        mpz_t p;
        mpz_init(p);
        mpz_set_str(p, "2", 10);
        mpz_nextprime(p, p);
        mpz_out_str(0, 10, p);
        printf("\n");

        mpz_t d;
        mpz_init(d);
        mpz_set_str(d, "100000000", 10);
        mpz_nextprime(d, d);
        mpz_out_str(0, 10, d);
        printf("\n");

        mpz_t k;
        mpz_init(k);
        mpz_set_str(k, "2", 10);
        mpz_out_str(0, 10, k);
        printf("\n");

        //n=p*d
        mpz_t n;
        mpz_init(n);
        mpz_mul(n, p, d);
        mpz_out_str(0, 10, n);
        printf("\n");

        clock_t start, end;
        double cpu_time_used;
        start = clock();

        Fermat_Factorization(ans, n);

        end = clock();
```

```
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

        mpz_out_str(0, 10, ans);
        printf("\n");

        //If we found a solution.
        if(mpz_cmp_ui(ans, 0)>0){
                mpz_t q;
                mpz_init(q);
                mpz_tdiv_q(q, n, ans);
                mpz_out_str(0, 10, q);
                printf("\n");
        }

        printf("Time: %f\n", cpu_time_used);

}
```

D.5 Dixons faktoriseringsalgoritm

```
#include <stdio.h>
#include <gmp.h>
#include <time.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <mpfr.h>

//Like the normal factorization but with mod to -n/2 to n/2.
void Improved_dixon(mpz_t ans, mpz_t n,...
            ...uint32_t maxPrime, uint32_t nrOfRetries){
        //  Berakna Erastothenes sall
        mpz_t numbersVector[maxPrime];
        mpz_init_set_si(numbersVector[0], -1);

        for (uint32_t i=1; i<maxPrime; i++) {
                mpz_init_set_ui(numbersVector[i],i);
        }
        for (uint32_t i=2; i<sqrt(maxPrime); i++) {
                if (mpz_cmp_ui(numbersVector[i],0)!=0) {
                        for (uint32_t j=i*2; j<maxPrime; j=j+i) {
                                mpz_set_ui(numbersVector[j],0);
                        }
                }
        }

        // Calculate the number of primes.
        uint32_t nrOfPrimes=0;
        for (uint32_t i=0; i<maxPrime; i++) {
                if (mpz_cmp_ui(numbersVector[i],0)!=0 &&...
                        ...mpz_cmp_ui(numbersVector[i],1)!=0) {
                                nrOfPrimes++;
                }
        }

    //Our final primevector.
    mpz_t primesVector[nrOfPrimes];
        uint32_t r=0;
        for (uint32_t i=0; i<maxPrime; i++) {
                if (mpz_cmp_ui(numbersVector[i],0)!=0 &&...
                    ...mpz_cmp_ui(numbersVector[i],1)!=0){
                        mpz_init_set(primesVector[r],numbersVector[i]);
                        r++;
                }
        }

        //Number of bi to choose randomly.
        uint32_t nrOfBi=nrOfPrimes+2;

    //List of the bi:s.
        mpz_t bis[nrOfBi+nrOfRetries];
        for (uint32_t i=0; i<nrOfBi+nrOfRetries; i++) {
                mpz_init(bis[i]);
        }
```

```
State for the random variables.
    gmp_randstate_t state;
    gmp_randinit_mt(state);

    uint32_t j=0;
    mpz_t bi;
    mpz_init(bi);
    mpz_t ci;
    mpz_init(ci);
    mpz_t nDividedBy2;
    mpz_init(nDividedBy2);
    mpz_tdiv_q_ui(nDividedBy2, n, 2);

//matrixOfPrimeVectors is in F_2 and numberedMatrixOfPrimeVectors...
...is the exponents in Z.
    short matrixOfPrimeVectors[nrOfPrimes][nrOfBi+nrOfRetries];
    short numberedMatrixOfPrimeVectors[nrOfPrimes][nrOfBi+nrOfRetries];

    uint32_t vector[nrOfPrimes+nrOfRetries];

    mpz_t ci_copy;
    mpz_init(ci_copy);

    //vectorMod2 is in F_2 and nrOfPrimesVector is the exponents in Z.
    short vectorMod2[nrOfPrimes];
    short nrOfPrimesVector[nrOfPrimes];

//Choose random bi until the number of bi is equal to nrOfBi.
    while(j<nrOfBi){
            mpz_urandomm(bi, state, n);
            mpz_mul(ci, bi, bi);
            mpz_mod(ci, ci, n);

            //Here we calculate mod from -n/2 to n/2.
            if(mpz_cmp(ci, nDividedBy2)>0){
                    mpz_sub(ci, ci, n);
            }

            // We ar not interested in b_i where b_i^2 = 0 mod n.
            if(mpz_cmp_ui(ci, 0)==0){
                    continue;
            }

            //Finding the vectors of the primes
            mpz_set(ci_copy, ci);
            for(uint32_t i=0; i<nrOfPrimes; i++){
                    vectorMod2[i]=0;
                    nrOfPrimesVector[i]=0;

                    while((i==0 && (mpz_cmp_ui(ci_copy, 0)<0)) ||...
                    ...(i!=0 && mpz_divisible_p(ci_copy, primesVector[i]))){
                            vectorMod2[i] ^= 1;
                            nrOfPrimesVector[i]++;
                            mpz_divexact(ci_copy, ci_copy, primesVector[i]);
                    }
```

```
            }
            //If c_i is maxPrime-smooth it is save, otherwise it is not.
            if(mpz_cmp_ui(ci_copy, 1)==0){
                    mpz_set(bis[j], bi);

                    for(uint32_t i=0; i<nrOfPrimes; i++){
                            matrixOfPrimeVectors[i][j]=vectorMod2[i];
                            numberedMatrixOfPrimeVectors[i][j]=...
                                ...nrOfPrimesVector[i];
                    }

                    j++;
            }
    }

    mpz_set_ui(ans, 0);

    //Variables for the loop:

    mpz_t upperlimit;
    mpz_init(upperlimit);

    mpz_t indexVector;
    mpz_init(indexVector);

    mpz_t testb;
    mpz_t testTemp;
    mpz_t diff;
    mpz_t indexAns;
    mpz_init(testb);
    mpz_init(testTemp);
    mpz_init(diff);
    mpz_init(indexAns);

    mpz_t testArray;
    mpz_init(testArray);

    mpz_t testc;
    mpz_t tempExp;
    mpz_init(testc);
    mpz_init(tempExp);

    //This loop will only run ones if we can find a factorization
    //with the bi we have gotten. Otherwise we will add more bi at
    // the end of the loop and run it until we find the answer.
    while(mpz_cmp_ui(ans, 0)==0 && nrOfBi < nrOfPrimes+2+nrOfRetries){

            uint32_t searchingRow=0;
            uint32_t searchingRowStart=0;
            short tempVect[nrOfBi];
            int foundAOne;

            //Gauss elimination of the matrix.
            for(uint32_t col=0; col<nrOfBi; col++){
                    searchingRow=searchingRowStart;
```

```
                foundAOne=0;

                //Find a row with a 1.
                while(searchingRow<nrOfPrimes){
                        if(matrixOfPrimeVectors[searchingRow][col]==1 &...
                            ...searchingRow==searchingRowStart){
                                foundAOne=1;
                                searchingRowStart++;
                                break;
                        }
                        else if(matrixOfPrimeVectors[searchingRow][col]==1){

                                //Move it to the first row
                                for(uint32_t colTemp=0; colTemp<nrOfBi;...
                                                                ...colTemp++){
                                        tempVect[colTemp]=...
...matrixOfPrimeVectors[searchingRowStart][colTemp];
                                        matrixOfPrimeVectors...
...[searchingRowStart][colTemp]=matrixOfPrimeVectors[searchingRow][col
                                        matrixOfPrimeVectors...
...[searchingRow][colTemp]=tempVect[colTemp];
                                }

                                foundAOne=1;
                                searchingRowStart++;

                                break;
                        }
                        else{
                                searchingRow++;
                        }
                }

                if(foundAOne){
                        //Add it to any other row with a one in that place.
                        for(uint32_t row=0; row<nrOfPrimes; row++){
                                if(row!=searchingRowStart-1 &...
                                ...matrixOfPrimeVectors[row][col]==1){
                                        for(uint32_t colTemp=0; colTemp<...
                                                        ...nrOfBi; colTemp++){
                                                matrixOfPrimeVectors[row]...
...[colTemp]=(matrixOfPrimeVectors[row][colTemp]+...
...matrixOfPrimeVectors[searchingRowStart-1][colTemp])%2;
                                        }
                                }
                        }
                }

        }

        //Solving Ax=0
        //Find free variables
        uint32_t freeVariables[nrOfBi];
        uint32_t row=0;
        uint32_t nrOfFreeVariabels=0;
        int foundFirstPivot=0;
```

```
for(uint32_t col=0; col<nrOfBi; col++){
        if(matrixOfPrimeVectors[row][col]==1){
                row++;
                freeVariables[col]=0;
        }
        else{
                nrOfFreeVariabels++;
                freeVariables[col]=1;
        }
}

//Finding spanning vectors
uint32_t spanningVectors[nrOfBi][nrOfFreeVariabels];
uint32_t variable=0;
uint32_t rowsWithoutPivot;

for(uint32_t col=0; col<nrOfBi; col++){
        rowsWithoutPivot=0;
        if(freeVariables[col]==1){
                for(uint32_t row=0; row<nrOfBi; row++){

                        if(col<row){
                                spanningVectors[row][variable]=0;
                        }
                        else if(col==row){
                                spanningVectors[row][variable]=1;
                        }
                        else if(freeVariables[row]==1){
                                spanningVectors[row][variable]=0;
                                rowsWithoutPivot++;
                        }
                        else{
                                if(rowsWithoutPivot>row){
                                        spanningVectors[row]...
                                                ...[variable]=0;
                                }
                                else{
                                        spanningVectors[row]...
                                                ...[variable]=
        ...matrixOfPrimeVectors[row-rowsWithoutPivot][col];
                                }
                        }

                }
                variable++;
        }
}

//Try all combinations of the spanning vectors.
/*
  Here we will use an integer as an array of bits
  where a bit being 1 means that we should include it and 0
  means we should not. The i:th bit can be recieved by
  taking ((The int) & (2 to the power of i))!=0.
```

```
  Note: we start at testArray=1 because
  we are not interested in the zero vector.
*/
mpz_t upperlimit;
mpz_init_set_ui(upperlimit, 1);
for(uint32_t i=0; i<nrOfFreeVariabels; i++){
        mpz_mul_ui(upperlimit, upperlimit, 2);
}

int boolean;
short cVector[nrOfPrimes];

mpz_set_ui(testArray, 1);

for(; mpz_cmp(testArray, upperlimit)<0; mpz_add_ui...
...(testArray, testArray, 1)){

        mpz_set_ui(testb, 1);
        for(uint32_t i=0; i<nrOfPrimes; i++){
                cVector[i]=0;
        }

        //Create the testb and cVector
        mpz_set_ui(indexVector, 1);
        for(uint32_t index=0; index<nrOfFreeVariabels; index++){
                mpz_and(indexAns, testArray, indexVector);
                if(mpz_cmp_ui(indexAns, 0)!=0){
                        for(uint32_t row=0; row<nrOfBi; row++){
                                if(spanningVectors[row][index]==1){
                        mpz_mul(testb, testb, bis[row]);
                        mpz_mod(testb, testb, n);

                        for(uint32_t i=0; i<nrOfPrimes; i++){
                                cVector[i]=cVector[i]+...
                                ...numberedMatrixOfPrimeVectors[i][row
                        }
                                }
                        }
                }
                mpz_mul_ui(indexVector, indexVector, 2);
        }

        mpz_set_ui(testc, 1);
        for(uint32_t i=0; i<nrOfPrimes; i++){
                if(cVector[i]!=0){
        mpz_powm_ui(tempExp, primesVector[i], cVector[i]/2, n);
                        mpz_mul(testc, testc, tempExp);
                        mpz_mod(testc, testc, n);
                }
        }

        //Try the testb and testc

        boolean = mpz_cmp(testb, testc)!=0;
        mpz_mul_si(testTemp, testc, -1);
        mpz_add(testTemp, testTemp, n);
```

```
                boolean = boolean && mpz_cmp(testb, testTemp)!=0;

                if(boolean){
                        mpz_sub(diff, testb, testc);
                        mpz_mod(diff, diff, n);
                        mpz_gcd(ans, diff, n);
                        break;
                }
        }

//If we do not find a solution.
        if(mpz_cmp_ui(ans, 0)==0){
                j=nrOfBi;
                nrOfBi++;

    //Choose random bi until the number of bi is equal to nrOfBi.
                while(j<nrOfBi){
                        mpz_urandomm(bi, state, n);
                        mpz_mul(ci, bi, bi);
                        mpz_mod(ci, ci, n);

                        //Here we calculate mod from -n/2 to n/2.
                        if(mpz_cmp(ci, nDividedBy2)>0){
                                mpz_sub(ci, ci, n);
                        }
                        // We ar not interested in b_i where b_i^2 = 0 mod n.
                        if(mpz_cmp_ui(ci, 0)==0){
                                continue;
                        }

                        //Finding the vectors of the primes
                        mpz_set(ci_copy, ci);
                        for(uint32_t i=0; i<nrOfPrimes; i++){
                                vectorMod2[i]=0;
                                nrOfPrimesVector[i]=0;

                                while((i==0 && (mpz_cmp_ui(ci_copy, 0)<0))...
                        ...|| (i!=0 && mpz_divisible_p(ci_copy, primesVector[i
                                        vectorMod2[i] ^= 1;
                                        nrOfPrimesVector[i]++;
                                        mpz_divexact(ci_copy, ci_copy,...
                                        ...primesVector[i]);
                                }
                        }

                        //If c_i is maxPrime-smooth it is save, otherwise it
                        if(mpz_cmp_ui(ci_copy, 1)==0){
                                mpz_set(bis[j], bi);

                                for(uint32_t i=0; i<nrOfPrimes; i++){
                                        matrixOfPrimeVectors[i][j]=...
                                        ...vectorMod2[i];
                                        numberedMatrixOfPrimeVectors[i][j]=...
                                        ...nrOfPrimesVector[i];
                                }
```

```
                                        j++;
                                }
                        }

                }
        }

        //If we have used all retries but have not found a factor.
        if(nrOfBi==nrOfPrimes+2+nrOfRetries){
                printf("\nDid not find solution\n\n");
        }
}

int main(){
        mpz_t ans;
        mpz_init(ans);

        mpz_t p;
        mpz_init(p);
        mpz_set_str(p, "1609161918110111", 10);
        //mpz_nextprime(p, p);
        mpz_out_str(0, 10, p);
        printf("\n");

        mpz_t d;
        mpz_init(d);
        mpz_set_str(d, "1109", 10);
        //mpz_nextprime(d, d);
        mpz_out_str(0, 10, d);
        printf("\n");

    //n=p*d
        mpz_t n;
        mpz_init(n);
        mpz_mul(n, p, d);
        mpz_out_str(0, 10, n);
        printf("\n");

    //Maxprime=J
        uint32_t J=5500;
        uint32_t nrOfRetries=10;

        clock_t start, end;
        double cpu_time_used;
        start = clock();

        Dixon(ans, n, J, nrOfRetries);

        end = clock();
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

        mpz_out_str(0, 10, ans);
        printf("\n");

        if(mpz_cmp_ui(ans, 0)>0){
```

```
                mpz_t q;
                mpz_init(q);
                mpz_tdiv_q(q, n, ans);
                mpz_out_str(0, 10, q);
                printf("\n");
        }

        printf("Time: %f\n", cpu_time_used);

}
```

D.6 Pollard-Rho

```
#include <stdio.h>
#include <gmp.h>
#include <time.h>
#include <stdlib.h>

//Don't care about sign.
void gcd(mpz_t ans, mpz_t a, mpz_t b){
        mpz_t r;
        mpz_init(r);

        mpz_t temp;
        mpz_init(temp);

        mpz_t a_new;
        mpz_init(a_new);

        mpz_t b_new;
        mpz_init(b_new);

        mpz_abs(a_new, a);
        mpz_abs(b_new, b);

        int go_On=1;
        while(go_On){
            //If a_new < b_new they switch places.
                if(mpz_cmp(a_new, b_new)<0){
                        mpz_set(temp, b_new);
                        mpz_set(b_new, a_new);
                        mpz_set(a_new, temp);
                }
                else{
                    //r is the reminder under divition if you divide...
                    ...a_new by b_new.
                        mpz_tdiv_r(r, a_new, b_new);

                //If the reminder was 0 we are done.
                        if(mpz_cmp_ui(r, 0)==0){
                                mpz_set(ans, b_new);
                                go_On=0;
                        }
                        else{
                            //Otherwise we run it again but with b_new and r.
                                mpz_set(a_new, b_new);
                                mpz_set(b_new, r);
                        }
                }
        }
}

void Rho_func(mpz_t value, mpz_t n){
        //f(x)=x^2+1 mod n.
        mpz_mul(value, value, value);
        mpz_add_ui(value, value, 1);
        mpz_mod(value, value, n);
```

```
}

void pollard_Rho_algorithm_two_steps(mpz_t ans,  mpz_t X, mpz_t n){
        mpz_t Y;
        mpz_init_set(Y, X);

        mpz_t diff;
        mpz_init(diff);

        int go_On=1;
        while(go_On){
                //x_{i+1}=f(x_i)
                //y_{i+1}=f(f(y_i))
                Rho_func(X, n);
                Rho_func(Y, n);
                Rho_func(Y, n);

                //ans = gcd(x_i - y_i, n)
                mpz_sub(diff, X, Y);
                mpz_gcd(ans, diff, n);

                //If ans =/= 1 and ans =/= n then ans is a factor of n.
                if(!(mpz_cmp_ui(ans, 1)==0 || mpz_cmp(ans, n)==0)){
                        go_On=0;
                }
        }
}

void pollard_Rho_algorithm_two_steps_own_gcd(mpz_t ans,  mpz_t X, mpz_t n){
        mpz_t Y;
        mpz_init_set(Y, X);

        mpz_t diff;
        mpz_init(diff);

        int go_On=1;
        while(go_On){
                //x_{i+1}=f(x_i)
                //y_{i+1}=f(f(y_i))
                Rho_func(X, n);
                Rho_func(Y, n);
                Rho_func(Y, n);

                //ans = gcd(x_i - y_i, n)
                mpz_sub(diff, X, Y);
                gcd(ans, diff, n);

        //If ans =/= 1 and ans =/= n then ans is a factor of n.
                if(!(mpz_cmp_ui(ans, 1)==0 || mpz_cmp(ans, n)==0)){
                        go_On=0;
                }
        }
}

int check_if_binary(mpz_t k){
        mpz_t q;
```

```
        mpz_init_set(q, k);

        mpz_t r;
        mpz_init(r);

        //Divide k by 2 until it is not divisible by 2 (not binary)  ...
        ...or you reach 1 (binary).
        while(mpz_cmp_ui(q, 1)>0){
                mpz_tdiv_qr_ui(q, r, q, 2);
                if(mpz_cmp_ui(r, 0)!=0){
                        return 0;
                }
        }
        return 1;
}

void pollard_Rho_algorithm_binary_slow(mpz_t ans, mpz_t X, mpz_t n){
        mpz_t k_plus_1;
        mpz_init_set_ui(k_plus_1, 1);

        mpz_t Xk;
        mpz_init_set(Xk, X);

        mpz_t Xj;
        mpz_init_set(Xj, X);

        mpz_t diff;
        mpz_init(diff);

        int go_On=1;
        while(go_On){
                Rho_func(Xk, n);
                mpz_add_ui(k_plus_1, k_plus_1, 1);

        //ans=gcd(Xk-Xj, n)
                mpz_sub(diff, Xk, Xj);
                mpz_gcd(ans, diff, n);

        //If ans =/= 1 and ans =/= n then ans is a factor of n.
                if(!(mpz_cmp_ui(ans, 1)==0 || mpz_cmp(ans, n)==0)){
                        go_On=0;
                }

        //If k_plus_1 is binary then k=2^h-1 for some h.
                if(check_if_binary(k_plus_1)){
                        mpz_set(Xj, Xk);
                }
        }
}

void pollard_Rho_algorithm_binary(mpz_t ans, mpz_t X, mpz_t n){
        mpz_t j_plus_1;
        mpz_init_set_ui(j_plus_1, 1);

        // j_plus_1 = 2^h, steps_until_change_j = 2^(h+1) - k
        mpz_t steps_until_change_j;
```

```
        mpz_init_set(steps_until_change_j, j_plus_1);

        mpz_t Xk;
        mpz_init_set(Xk, X);

        mpz_t Xj;
        mpz_init_set(Xj, X);

        mpz_t diff;
        mpz_init(diff);

        int go_On=1;
        while(go_On){
                Rho_func(Xk, n);
                mpz_sub_ui(steps_until_change_j, steps_until_change_j, 1);

        //ans=gcd(Xk-Xj, n)
                mpz_sub(diff, Xk, Xj);
                mpz_gcd(ans, diff, n);

        //If ans =/= 1 and ans =/= n then ans is a factor of n.
                if(!(mpz_cmp_ui(ans, 1)==0 || mpz_cmp(ans, n)==0)){
                        go_On=0;
                }

        //If steps_until_change_j=0 then it is time to update...
        ...j_plus_1 = 2^h to 2^(h+1).
                //This means that steps_until_change_j = 2^(h+2) - k ...
                ...= 2^(h+2) - 2^(h+1) = 2^(h+1) = j_plus_1.
                if(mpz_cmp_ui(steps_until_change_j, 0)==0){
                        mpz_set(Xj, Xk);
                        mpz_mul_ui(j_plus_1, j_plus_1, 2);
                        mpz_set(steps_until_change_j, j_plus_1);
                }
        }
}

int main(){
    //n=p*d, X=X_0, f(x)=x^2+1.

        mpz_t p;
        mpz_init(p);
        mpz_set_str(p, "1609161918110111", 10);
        //mpz_nextprime(p, p);
        mpz_out_str(0, 10, p);
        printf("\n");

        mpz_t d;
        mpz_init(d);
        mpz_set_str(d, "1109", 10);
        //mpz_nextprime(d, d);
        mpz_out_str(0, 10, d);
        printf("\n");

        mpz_t n;
        mpz_init(n);
```

```
        mpz_mul(n, p, d);
        mpz_out_str(0, 10, n);
        printf("\n");

        mpz_t X;
        mpz_init_set_ui(X, 1);
        mpz_out_str(0, 10, X);
        printf("\n");

        mpz_t ans;
        mpz_init_set_ui(ans, 0);
        mpz_out_str(0, 10, ans);
        printf("\n");

        clock_t start, end;
        double cpu_time_used;
        start = clock();

        pollard_Rho_algorithm_two_steps(ans, X, n);

        end = clock();
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

        mpz_out_str(0, 10, ans);
        printf("\n");

        mpz_t q;
        mpz_init(q);
        mpz_tdiv_q(q, n, ans);
        mpz_out_str(0, 10, q);
        printf("\n");

        printf("Time: %f", cpu_time_used);
}
```

D.7 Kvadratiskt Såll

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdint.h>
#include <gmp.h>
#include <mpfr.h>


int main() {

    // Steg 0: initialisera variabler
    uint32_t  sieveMax, count, antalPrim;
    mpz_t N, sqrtN, square, B, tmp, faktor, x, y, d;
    mpz_inits(square, B, faktor, tmp, x, y, NULL);
    mpz_init_set_ui(d,1);


    /***         Har ar alla variabler man staller in manuellt.       ***/
    //           Vill man testa "pa riktigt" satt ena variabeln nedan
    //           till 1 och den andra till talet som ska faktoriseras.

    // Nagra exempel primtal for att test
    mpz_init_set_str(N, "502217", 10);     //502217 1613 2143 1609161918110111
    mpz_init_set_str(tmp,"2087", 10);          //524287 2087 1109 1510553619999637
    uint32_t kandidatMax=200000;                        // Hur manga tal vi sallar
    uint32_t linDep=10;                    // Hur manga extra tal vill vi hitta?


    /***                                                                  ***/
    mpz_mul(N, N, tmp);

    mpz_t xKandidat[kandidatMax], yKandidat[kandidatMax];    // Kandidattal
    for (uint32_t i=0; i<kandidatMax; i++) {
        mpz_init(xKandidat[i]);
        mpz_init(yKandidat[i]);
    }

    // Rakna ut upper bound B for att valja ut B-glatta tal.
    // B=ceil(sqrt(e^(sqrt(ln n * ln ln n))))
    mpfr_t LofN, lnN;
    mpfr_inits(LofN, lnN, NULL);
    mpfr_set_z(lnN, N, MPFR_RNDD);
    mpfr_log(lnN, lnN, MPFR_RNDD);
    mpfr_log(LofN, lnN, MPFR_RNDD);
    mpfr_mul(LofN, LofN, lnN, MPFR_RNDD);
    mpfr_sqrt(LofN, LofN, MPFR_RNDD);
    mpfr_exp(LofN, LofN, MPFR_RNDD);
    mpfr_sqrt(LofN, LofN, MPFR_RNDU);
    mpfr_get_z(B, LofN, MPFR_RNDU);
    mpfr_clears(LofN, lnN, NULL);

    if (mpz_cmp_ui(B, 4294967295) > 0) {
        sieveMax=4294967295;              // Av implementeringsskal begransar vi
    }                                  // oss till max for en 32 bit int,
```

```
else if (mpz_cmp_ui(B,120)<0) { // vi har aven en minimigrans for att
    sieveMax=120;               // faktorisera sma tal snabbt
}
else {
    sieveMax=mpz_get_ui(B);
}

// Steg 1: Berakna faktorbasen med hjalp av Erastothenes sall
uint32_t sieveTable[sieveMax];
for (uint32_t i=0; i<sieveMax; i++) {
    sieveTable[i]=i;
}
for (uint32_t i=2; i<sqrt(sieveMax); i++) {
    if (sieveTable[i]!=0) {
        for (uint32_t j=i*2; j<sieveMax; j=j+i) {
            sieveTable[j]=0;
        }
    }
}

antalPrim=1;
for (uint32_t i=3; i<sieveMax; i++) {       // Rakna antalet primtal,
    if (sieveTable[i] != 0) {                   // Och behall bara de med
        if (mpz_kronecker_ui(N,sieveTable[i])==1) {
            antalPrim++;                        // Legendre(N,p)=1.
        }
        else {
            sieveTable[i]=0;
        }
    }
}

uint32_t faktorBas[antalPrim];
count=0;
// Varan slutgiltiga faktorbas
for (uint32_t i=2; i<sieveMax; i++) {
    if (sieveTable[i]!=0) {
        faktorBas[count]=sieveTable[i];
        count++;
    }
}



// Steg 2: Salla sekvensen x^2-N for glatta tal.
mpz_t xVec[antalPrim+linDep];
mpz_t yVec[antalPrim+linDep];
for (uint32_t i=0; i<antalPrim+linDep; i++) {
    mpz_init(xVec[i]);
    mpz_init(yVec[i]);
}

uint32_t aVec1[antalPrim];          // Startvarden for sallet, varje tal
uint32_t aVec2[antalPrim];          // har tva st.
for (uint32_t i=0; i<antalPrim; i++) {
    aVec1[i]=0;
```

```
        aVec2[i]=0;
    }

    count=0;
    mpz_set_ui(tmp,0);

    // Medan vi inte har tillrackligt med glatta polynom
    while (count < antalPrim+linDep) {
        for (uint32_t j=0; j<kandidatMax; j++) {     // Fyll listan med kandidater
            mpz_add_ui(tmp,tmp,1);
            mpz_root(sqrtN, N, 2);
            mpz_add(sqrtN,sqrtN,tmp);
            mpz_set(xKandidat[j],sqrtN);
            mpz_pow_ui(sqrtN,sqrtN,2);
            mpz_sub(yKandidat[j],sqrtN,N);
        }
        // Hitta startvarden for sallet
        for (uint32_t j=0; j<antalPrim; j++) {
            for (uint32_t k=0; k<kandidatMax; k++) {
                if (mpz_divisible_ui_p(yKandidat[k], faktorBas[j])) {
                    aVec1[j]=k;
                    for (uint32_t l=k+1; l<kandidatMax; l++) {
                        if (mpz_divisible_ui_p(yKandidat[l], faktorBas[j])) {
                            aVec2[j]=l;
                            break;
                        }
                    }
                    break;
                }
            }
        }
        for (uint32_t j=0; j<antalPrim; j++) {
            mpz_set_ui(faktor, faktorBas[j]);

            for (uint32_t l=aVec1[j]; l<kandidatMax; l=l+faktorBas[j]){
                mpz_remove(yKandidat[l], yKandidat[l], faktor);
            }
            for (uint32_t l=aVec2[j]; l<kandidatMax; l=l+faktorBas[j]){
                mpz_remove(yKandidat[l], yKandidat[l], faktor);
            }
        }
        for (uint32_t j=0; j<kandidatMax; j++) {
            if (!mpz_cmp_ui(yKandidat[j], 1)) {
                mpz_set(xVec[count],xKandidat[j]);
                mpz_pow_ui(sqrtN,xVec[count],2);
                mpz_sub(yVec[count],sqrtN,N);
                count++;
            }
            if (count==(antalPrim+linDep)) {
                break;
            }
        }

    }

    // Steg 3: Skapa matris med exponentvektorer, radreducera
```

```
//for att hitta kvadratisk kongruens.
uint32_t *exponentMatrix = (uint32_t *)malloc...
...( (antalPrim+linDep) * antalPrim * sizeof(uint32_t));
for (uint32_t i=0; i<(antalPrim*(antalPrim+linDep)); i++) {
    *(exponentMatrix+i)=0;
}
for (uint32_t i=0; i<antalPrim+linDep; i++) {
    for (uint32_t j=0; j<antalPrim; j++) {
        if (mpz_divisible_ui_p(yVec[i], faktorBas[j])) {
            mpz_set_ui(faktor, faktorBas[j]);
            *(exponentMatrix + i*(antalPrim)+j)=(mpz_remove(tmp,yVec[i],faktor));
        }
    }
}



// Radreduceringen
uint16_t *reducedMatrix = (uint16_t *)malloc( (antalPrim+linDep) *...
...antalPrim * sizeof(uint16_t));
for (uint32_t i=0; i<(antalPrim+linDep)*antalPrim; i++) {
    *(reducedMatrix+i)=*(exponentMatrix+i)%2;
}
uint16_t *historyMatrix = (uint16_t *)malloc( (antalPrim+linDep) *...
...(antalPrim+linDep) * sizeof(uint16_t));
for (uint32_t i=0; i<(antalPrim+linDep)*(antalPrim+linDep);...
... i=i+antalPrim+linDep+1) {
    *(historyMatrix+i)=1;
}
uint16_t onetotheleft=0;
uint32_t startRow=0, row=0;
for (uint32_t col=0; col<antalPrim; col++) {
    onetotheleft=0;
    row=startRow;
    while (!*(reducedMatrix+row*antalPrim+col)&&row<antalPrim+linDep) {
        row++;
    }
    if(col) {
        for (uint32_t i=0; i<col; i++) {
            if (*(reducedMatrix+row*antalPrim+i)) {
                onetotheleft=1;
            }
        }
    }
    // Om det inte fanns nagon etta i kolonnen gar vi till nasta
    if (row >= antalPrim+linDep) {
        startRow=0;
        continue;
    }
    else if (onetotheleft) {
        startRow=row+1;
        col--;
    }
    // Dags att plussa pivot raden
    else {
        startRow=0;
```

```
            for (uint32_t i=0; i<antalPrim+linDep; i++) {
                if (*(reducedMatrix+i*antalPrim+col)&&i!=row) {
                    for (uint32_t j=0; j<antalPrim+linDep; j++) {
                        *(historyMatrix+i*(antalPrim+linDep)+j)=...
                        ...(*(historyMatrix+i*(antalPrim+linDep)+j)+...
                        ...(*(historyMatrix+row*(antalPrim+linDep)+j)))%2;
                    }
                    for (uint32_t j=0; j<antalPrim; j++) {
                        *(reducedMatrix+i*antalPrim+j)=...
                        ...((*(reducedMatrix+row*antalPrim+j)+...
                        ...(*(reducedMatrix+i*antalPrim+j)))%2);
                    }
                }
            }
        }
    }



    // Steg 4: Titta efter nollrad i radreducerad matris
    // historyMatrix later oss se vilka rader som summeras
    uint32_t yExpVec[antalPrim];
    uint32_t nollRad=0;
    uint32_t sum=0;
    uint16_t done=0;
    // done markerar om vi far trivial faktor
    while (!done) {
        for (uint32_t i=0; i<antalPrim; i++) {
            yExpVec[i]=0;                           // Rensa denna vektorn varje loop
        }

        //Hitta noll rad
        for (uint32_t row=nollRad; row<antalPrim+linDep; row++) {
            sum=0;
            for (uint32_t col=0; col<antalPrim; col++) {
                sum=sum+(*(reducedMatrix+row*antalPrim+col));
            }
            if(!sum) {
                nollRad=row;
                break;
            }

        }

        //Skapa exponentvektorn for slutgiltiga y samt skapa x
        mpz_set_ui(x,1);
        for (uint32_t whichY=0; whichY<antalPrim+linDep; whichY++) {
            if (*(historyMatrix + nollRad*(antalPrim+linDep)+whichY)) {
                for (uint32_t primeIndex=0; primeIndex<antalPrim; primeIndex++) {
                    yExpVec[primeIndex]=yExpVec[primeIndex]+...
                    ...(*(exponentMatrix+whichY*(antalPrim)+primeIndex));
                }
                mpz_mul(x,x,xVec[whichY]);
            }
        }
        mpz_mod(x,x,N);
```

```
        // Skapa y
        mpz_set_ui(y,1);
        for (uint32_t i=0; i<antalPrim; i++) {
            yExpVec[i]=yExpVec[i]/2;      //Forst tar vi roten ur
            mpz_ui_pow_ui(tmp,faktorBas[i],yExpVec[i]);
            mpz_mul(y, y, tmp);
        }
        mpz_mod(y,y,N);


        mpz_sub(x,x,y);
        mpz_gcd(d,x,N);           // Vart svar, kandidatfaktorn


        done=1;
        nollRad++;
        if (!mpz_cmp_ui(d,1)){   // Leta efter triviala faktorer
            done=0;
        }
        else if(!mpz_cmp(d,N)){
            done=0;
        }
        if(nollRad>=antalPrim+linDep){
            done=1;
        }
    }


    printf("N=␣");
    mpz_out_str(stdout,10,N);
    putchar('\n');

    printf("faktorBas:");
    for (uint32_t i=0;i<antalPrim; i++) {
        printf("␣%d", faktorBas[i]);
    }

    printf("\n\nMatris␣av␣exponentvektorer:\n");
    for (uint32_t i=0; i<antalPrim + linDep; i++) {
        for (uint32_t j=0; j<antalPrim; j++) {
            printf("%d␣", *(exponentMatrix + i*(antalPrim)+j));
        }
        putchar('\n');
    }

    printf("\n␣\n␣xVec=,␣␣␣␣␣␣␣␣yVec=\n");
    for (uint32_t i=0; i<antalPrim+linDep; i++) {
        mpz_out_str(stdout, 10, xVec[i]);
        printf("␣␣␣␣");
        mpz_out_str(stdout, 10, yVec[i]);
        putchar('\n');
    }
    printf("\n␣aVec1=");
    for (uint32_t i=0; i<antalPrim; i++) {
        printf("␣%d",aVec1[i]);
```

```
    }
    printf("\n aVec2=");
    for (uint32_t i=0; i<antalPrim; i++) {
        printf(" %d",aVec2[i]);
    }

    printf("\n\nRadreducerad matris:\n");
    for (uint32_t i=0; i<antalPrim +linDep ; i++) {
        for (uint32_t j=0; j<antalPrim; j++) {
            printf("%d ", *(reducedMatrix + i*(antalPrim)+j));
        }
        putchar('\n');
    }

    printf("\nHistoriematris:\n");
    for (uint32_t i=0; i<antalPrim +linDep ; i++) {
        for (uint32_t j=0; j<antalPrim+linDep; j++) {
            printf("%d ", *(historyMatrix + i*(antalPrim+linDep)+j));
        }
        putchar('\n');
    }

    printf("\nnollRad=%d\n",nollRad-1);
    printf("\nyExpVec=");
    for (uint32_t i=0; i<antalPrim; i++) {
        printf(" %d",yExpVec[i]);
    }

    printf("\nx=");
    mpz_out_str(stdout, 10, x);
    printf(", y=");
    mpz_out_str(stdout,10,y);
    printf(", gcd(x-y,N)=Faktor=");
    mpz_out_str(stdout,10,d);
    putchar('\n');


    mpz_clears(N, sqrtN, square, B, tmp, faktor, x, y, d, xKandidat, yKandidat, NULL);
// Rensa minnet



}
```

D.8 Kedjebråksmetoden

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;


namespace Kedjebraksfaktorisening
{
    class Program

    {
        //************************* Legendre symbol ***************
        List<sbyte> tab2 = new List<sbyte> { 0, 1, 0, -1, 0, -1, 0, 1 };
        sbyte v = 0;
        int kretur = 0;
        int res;
        int a;

        private int Legendre(int b, BigInteger N)
        // Returnerar 0 om (N/p)=-1, 1 annars.
        {
            //Console.WriteLine("Legendre "+ N);
            //2
            v = 0;
            kretur = 1;

            //3
            //Console.WriteLine("i steg 3: N=" + N + " a=" + a + " b=" + b);
            a = (int)(N % b);

            //4
            while (true)
            {
                //Console.WriteLine("N="+N+" b="+b);
                if (a == 0)
                {
                    if (b > 1) { return 0; }
                    if (b == 1) { return kretur; }
                }
                //5
                v = 0;
                while (a % 2 == 0)
                {
                    v++;
                    a = a / 2;
                }
                if (v % 2 == 1)
                {
                    kretur = kretur * tab2[b % 8];
//(int)Math.Pow(-1, (b * b - 1) / 8);
                }
                //6
```

```
                res = b - a;
                if (res > 0)
                {

                    if ((2 & b & a) > 0) { kretur = -kretur; }
// k = k * (int)Math.Pow(-1, (N - 1) * (b - 1) / 4);
                    b = a;
                    a = res;
                }
                else
                { a = -res; }


            }

        }

        //************************ Trial and Error *****************
        List<BigInteger> qF = new List<BigInteger>();
        BigInteger fp;
        private List<BigInteger> TrialAndError(BigInteger Q, List<BigInteger>...
                        ...PrimtalLista)
        {
            qF.Clear();

            while (Q != FirstPrime(Q))
            {
                //Console.WriteLine("I W");
                fp = FirstPrime(Q);
                qF.Add(fp);
                Q = Q / fp;
            }
            qF.Add(Q);


            BigInteger FirstPrime(BigInteger K)
            {
                foreach (BigInteger p in PrimtalLista)
                {
                    //Console.WriteLine("Trial 2 "+ p+" ");
                    if (BigInteger.Remainder(K, p) == 0 && p!=-1)
                    {
                        return p;
                    }
                }
                return K;
            }
            //Console.WriteLine("QFakorisering klar");
            return qF;

        }

        //**********************Eratostenes Soll ******************************
        private List<BigInteger> EratosthenesSoll(int n)
        {
            //Console.WriteLine("ERA "+n);
```

```
        DateTime st = DateTime.Now;
        List<BigInteger> primvekt = new List<BigInteger>();
        List<BigInteger> ErasResultat = new List<BigInteger>();
        BigInteger nsq = Sqrt(n);

        //lista alla tal tom n
        for (int i = 0; i < n; i++) { primvekt.Add(i); }

        //Rensa multiplar
        for (int i = 0; i < nsq; i++)
        {
            //Console.WriteLine("primvekt= " + primvekt[i]+" i= "+i+" nsq= "+nsq);
            if (primvekt[i] > 1)
            {

                for (int j = i + i; j < n; j = j + i)
                {
                    primvekt[j] = 0;
                }
            }
        }
        primvekt[0] = 0;
        primvekt[1] = 0;
        for (int i = 3; i < primvekt.Count - 1; i++)
        {
            //Console.WriteLine(primvekt[i]);
            if (primvekt[i] > 2 && Legendre((int)primvekt[i], NN) == -1)
            {
                //Console.WriteLine("**** soll " + primvekt[i]);
                primvekt[i] = 0;
            }
        }

        //Rensa nollor och baka ihop
        for (int i = 0; i < n; i++)
        {
            if (primvekt[i] != 0) { ErasResultat.Add(primvekt[i]); }
        }
        DateTime tt = DateTime.Now;
        //Console.WriteLine("Tid EratosthenesSoll " + (tt - st).Milliseconds);
        //foreach (int pr in primvekt)
        //{ Console.WriteLine(pr); }
        //Console.WriteLine("Eratostenes klart! " + ErasResultat.Count);
        return ErasResultat;
    }
    //********************** Roten ur ************************
    List<BigInteger> kvoter = new List<BigInteger>();
    public BigInteger Sqrt(BigInteger n)
    {
        //Console.WriteLine("Roten ur " + n);
        //Console.WriteLine("n="+n);
        BigInteger X0 = 0;
        BigInteger X1 = n;
        int i = 0;
        while (X0 != X1)
        {
```

```
            X0 = X1;
            X1 = X0 - (BigInteger.Pow(X0, 2) - n) / (2 * X0);
            i++;


        }
        while (!(BigInteger.Pow(X1, 2) <= n && BigInteger.Pow(X1 + 1, 2) > n))
        {
            //Console.WriteLine(BigInteger.Pow(X1, 2) + "\t" + BigInteger.Pow...
            ...(X1 + 1, 2));
            if (BigInteger.Pow(X1 + 1, 2) < n) { X1++; }
            if (BigInteger.Pow(X1, 2) > n) { X1--; }
            //Console.WriteLine("sqrt 3");
        }
        //Console.WriteLine(i);

        return X1;

    }




List<BigInteger> Q = new List<BigInteger>();
List<BigInteger> P = new List<BigInteger>();
List<BigInteger> r = new List<BigInteger>();
List<BigInteger> q = new List<BigInteger>();
List<BigInteger> A = new List<BigInteger>();
List<BigInteger> QfaktorerTemp = new List<BigInteger>();
List<List<BigInteger>> Qfaktorer = new List<List<BigInteger>>();

BigInteger NN;
int k = 0;
//****************************** Faktorisering ***************
void QF(BigInteger N)
{
    Console.WriteLine("QF");
    NN = N;


    int varv = 1200;
    int limit =800;

    BigInteger resultat = 1;
    BigInteger g;
    bool fortsatt = true;
    List<BigInteger> Primtal = new List<BigInteger>();
    List<List<bool>> expoMatris = new List<List<bool>>();

    List<int> indexVektor = new List<int>();

    bool nyVektor = false;

    List<bool> Li = new List<bool>();
    Console.WriteLine("N=_____" + N);
    Console.WriteLine("Limit=_" + limit);
```

```
Console.WriteLine("Varv=  " + varv);
int numberOfRows = Primtal.Count;
Console.WriteLine("NoRows= " + numberOfRows);
while (fortsatt)
{

    Q.Clear(); P.Clear(); r.Clear();
    q.Clear();A.Clear(); Qfaktorer.Clear();
    expoMatris.Clear(); Primtal.Clear(); indexVektor.Clear();
    k++;
    while (k % 4 == 0 || k % 9 == 0 || k % 25 == 0 || k % 49 == 0)
    {
        k++;
    }
    NN = k*N;
    //Console.WriteLine("K=" + k);
    Primtal = EratosthenesSoll(limit);
    Primtal.Insert(0,-1);
    g = Sqrt(k * N);

    //PrimBas = EratosthenesSoll(g);
    A.Add(0);       //-2
    A.Add(1);       //-1
    Q.Add(0);       //-2 .... nonsens
    Q.Add(k * N); //-1
    Q.Add(1);       //0
    r.Add(0);       //-2 .....nonsens
    r.Add(g);       // -1
    P.Add(0);     //-2.....nonsens
    P.Add(0);     //-1.....nonsens
    P.Add(0);       // 0
    q.Add(0);     // -2.... nonsens
    q.Add(0);      // -1.... nonsens
    Qfaktorer.Add(new List<BigInteger>()); //-2 nonsens;
    Qfaktorer.Add(new List<BigInteger>()); //-1 nonsens;
    Qfaktorer.Add(new List<BigInteger>()); // 0 nonsens;


    //Console.WriteLine("n\tg+Pn\tQn\tqn\trn\tA(n-1)\t\tQnfact");


    int i = 1;

    while (i < varv || expoMatris.Count < numberOfRows + 2)
    {
        i++;
        //Console.WriteLine("Varv nr " + i +" ,K= "+k);
        q.Add(BigInteger.Divide(g + P[i], Q[i]));
        r.Add(BigInteger.Remainder(g + P[i], Q[i]));
        A.Add(BigInteger.Remainder(q[i] * A[i - 1] + A[i - 2], N));
        P.Add(g - r[i]);
        Q.Add(Q[i - 1] + q[i] * (r[i] - r[i - 1]));

        if (i > 2)
        {
            //Console.WriteLine("TAE " + Q[i]);
```

```
            Qfaktorer.Add(TrialAndError(Q[i], Primtal));

        }

        //********************** ExponentMatris**********************

        nyVektor = false;

        foreach (BigInteger Qfaktor in Qfaktorer[i])
        {
            //Console.Write(Qfaktor + "\t");
            if (Qfaktorer[i][Qfaktorer[i].Count - 1] < limit &&...
            ...Qfaktorer[i].Count!=1)
            { nyVektor = true; }
        }


        if (nyVektor)              //***Gor en ny vektor till expomatris***
        {
            //Console.WriteLine("Gor ny vektor, " +expoMatris.Count);
            indexVektor.Add(i);

            bool udda = Math.Pow(-1, i) == -1;

            expoMatris.Add(new List<bool>());
            foreach(BigInteger p in Primtal)
            {
                expoMatris[expoMatris.Count-1].Add...
                ...(p == -1 ? udda : false);
            }

            foreach (BigInteger Qfakt in Qfaktorer[i])
            {
        expoMatris[expoMatris.Count - 1][Primtal.IndexOf(Qfakt)] =...
        ...!expoMatris[expoMatris.Count - 1][Primtal.IndexOf(Qfakt)];
            }


        }
        //-------------------------------------

    }

    //*************** Extend history companion matrix **************
    int compainionStart = expoMatris[0].Count;
    int j = 0;
    //Console.WriteLine("Gor history companion");
    foreach (List<bool> Rad in expoMatris)
    {
        for (int ku = 0; ku < expoMatris.Count; ku++)
        {
            if (ku == j) { expoMatris[j].Add(true); }
            else { expoMatris[j].Add(false); }
        }
```

```
    j++;
}
int kn = 0;
/*
//********** Skriv ut expomatr ******************
Console.WriteLine("Expmatr");
foreach (List<bool> L in expoMatris)
{
    Console.Write((indexVektor[kn] - 2) + "\t");
    kn++;
    foreach (bool b in L)
    {
        Console.Write(b + "\t");
    }
    Console.Write("\n");
}
*/
//************* Radreducering ******************
int rj = compainionStart;
int m = expoMatris.Count;
int totLangd = expoMatris[0].Count;
bool EttorFinns = false;
Console.WriteLine("radreducerar k=" + k);
while (rj > 0)
{
    j = 0;
    while (!expoMatris[j][rj - 1] && j < m - 1)
    {
        j++;
    }

    if (expoMatris[j][rj - 1])
    {
        for (int ku = j + 1; ku < m; ku++)
        {
            if (expoMatris[ku][rj - 1])
            {
                //kolla nollor efter
                EttorFinns = false;
                for (int q = rj; q < compainionStart - 1; q++)
                {
                    EttorFinns = EttorFinns || expoMatris[ku][q];
                }
                if (!EttorFinns)
                {
                    for (int h = 0; h < totLangd; h++)
                    {
                        expoMatris[ku][h] =...
                        ...expoMatris[ku][h] ^ expoMatris[ku][h];
                    }
                }

            }
        }
    }
    rj--;
```

```
}
/*
Console.WriteLine("**********************************");
Console.Write("\t");
foreach (BigInteger p in Primtal)
{
    Console.Write(p + "\t");
}
Console.Write("\n");
kn = 0;

//********** Skriv ut expomatr matr reducerad*****************
Console.WriteLine("Reducerad");
foreach (List<bool> L in expoMatris)
{
    Console.Write((indexVektor[kn]-2) + "\t");
    kn++;
    foreach (bool b in L)
    {
        Console.Write(b + "\t");
    }
    Console.Write("\n");
}
*/
//****** Nollrader ****************
bool nollKoll = false;
List<List<BigInteger>> AQ = new List<List<BigInteger>>();
BigInteger Qsquare = 1;
BigInteger Qu = 1;
BigInteger Qmod = 1;
BigInteger Amod = 1;
for (int rad = 0; rad < expoMatris.Count; rad++)
{
    nollKoll = false;
    for (int kol = 0; kol < compainionStart; kol++)
    {
        nollKoll = nollKoll || expoMatris[rad][kol];
    }
    if (!nollKoll)
    {
        Qsquare = 1;
        for (int kol = compainionStart; kol < expoMatris[rad].Count;..
        ...kol++)
        {
            if (expoMatris[rad][kol])
            {
                Console.WriteLine(Q[indexVektor[kol -...
                ...compainionStart]] + "␣" +...
                ...A[indexVektor[kol - compainionStart] - 1]);
                Qsquare = Qsquare * Q[indexVektor[kol...
                ...- compainionStart]];
                Amod = (Amod * A[indexVektor[kol -...
                ...compainionStart] - 1]) % N;
            }
        }
        Qu = Qsquare == 1 ? 1 : Sqrt(Qsquare);
```

```
                        Qmod = Qu % N;

                        resultat = BigInteger.GreatestCommonDivisor(N, Amod - Qmod);
                        if (!(resultat == 1 || resultat == N))
                        {
                            fortsatt = false;


                            Console.WriteLine("Resultat: " + resultat);
                            Console.WriteLine("Fortsatt= " + fortsatt);
                            break;
                        }
                    }
                }
                //Console.WriteLine(k);
            }


        }



        //************************MAIN*******************

        static void Main(string[] args)
        {
            DateTime st = DateTime.Now;

            Program p = new Program();
            p.QF(BigInteger.Parse("1784560567184113099"));

            // Extra tal att testa:
            //1111611911141458617084413 1009168486435149 5886154573
            //24307253605667876038755460297O7 87465768453 768764568745

            // Printa tid
            Console.WriteLine("Tid; " +(DateTime.Now-st).Minutes+"min "+...
            ...(DateTime.Now - st).Seconds + "s " + (DateTime.Now - st).Milliseconds -
            Console.ReadKey();
        }
    }
}
```

D.9 Solovay-Strassens test

```
#include <stdio.h>
#include <gmp.h>


int main() {
    gmp_randstate_t rstate;
    gmp_randinit_default(rstate);                          // For att slumpa tal

    mpz_t vanster, hoger, b, N, tmp, expo;      // Alla stora varden vi behover
    mpz_inits(vanster, hoger, b, tmp, expo, NULL);


    int rundor = 200;                                  // Valj antal rundor
    mpz_init_set_str(N, "15 ", 10);       // Valj tal


    mpz_sub_ui(tmp,N,1);
    mpz_divexact_ui(expo, tmp, 2);

    for (int i=1; i<rundor; i++) {
        // b ar slumpat i (0, tmp-1) = (0, n-2)
        mpz_urandomm(b, rstate, tmp);
        mpz_add_ui(b, b, 1);     // ratta intervallet till (1,n-1)
        mpz_powm(vanster, b, expo, N);

        mpz_set_si(hoger, mpz_jacobi(b,N));
        mpz_mod(hoger, hoger, N);

        if (mpz_cmp(vanster, hoger)) {
            mpz_out_str(stdout, 10, N);
            printf(" ar ej ett primtal.\n");

            mpz_clears(vanster, hoger, b, N, tmp, expo, NULL);
            return 0;
        }

        if (mpz_gcd(b,n)!=1) {        // Om b delar n ar det saklart sammansatt
            mpz_out_str(stdout, 10, N);
            printf(" ar ej ett primtal.\n");

            mpz_clears(vanster, hoger, b, N, tmp, expo, NULL);
            return 0;
        }
    }

    mpz_out_str(stdout, 10, N);
    printf(" ar ett primtal med sannolikhet 2^-%d.\n", rundor);

    mpz_clears(vanster, hoger, b, N, tmp, expo, NULL);
    return 0;
}
```

D.10 RSA tillsammans med Miller-Rabin

```
using System;
using System.Collections.Generic;
using System.Security.Cryptography;
using System.Numerics;

// Krypterar ett meddelande enligt RSA-algoritmen med
// tva slumpmassigt valda primtal.
// Inkluderade Algoritmer:
// Eratosthenes soll, Miller-Rabin Primtalstest och Euclides.

namespace erReSsA
{

    class Program
    {
        class Algos

        {
            //************************PRIMTATHET*****************
            private double PrimDens(int strl)
            {
                //primtalsatsen
                double dense = strl * Math.Log(10);
                Console.WriteLine("-----------------------------------");
                Console.WriteLine("strl= " + strl);
                Console.WriteLine("dense= " + dense);
                return dense;
            }

            //********************** PRIMLISTA Eratostenes soll *******
            public List<int> EratosthenesSoll(int n)
            {
                Console.WriteLine("Eratosthenes Soll(" + n + ") ");

                DateTime st = DateTime.Now;

                List<int> primvekt = new List<int>();
                List<int> ErasResultat = new List<int>();
                int nsq = (int)Math.Round(Math.Sqrt(n));

                //lista alla tal tom n
                for (int i = 0; i < n; i++) { primvekt.Add(i); }

                for (int i = 0; i < nsq; i++)
                {
                    if (primvekt[i] > 1)
                    {
                        for (int j = i + i; j < n; j = j + i)
                        {
                            primvekt[j] = 0;
                        }
                    }
                }
                primvekt[0] = 0;
```

```
        primvekt[1] = 0;
        primvekt[2] = 0;

        //Rensa nollor och baka ihop
        for (int i = 0; i < n; i++)
        {
            if (primvekt[i] != 0) { ErasResultat.Add(primvekt[i]); }
        }
        DateTime tt = DateTime.Now;
        Console.WriteLine("Tid EratosthenesSoll " + (tt - st).Milliseconds);

        return ErasResultat;
    }

    //********************** PRIMTEST ***********
    public bool PrimTestMillerRabin(BigInteger PrimKandidat,...
    ...int nvarv, List<int> Primvect)
    {
        //0
        DateTime st = DateTime.Now;

        // Kolla delbarhet med de mindre primtalen
        if (boost1)
        {
            foreach (int x in Primvect)
            {
                if (BigInteger.GreatestCommonDivisor...
                ...(PrimKandidat, x) > 1) { inik++; return false; }
            }
        }

        BigInteger PminusEtt = PrimKandidat - 1;
        bool probPrime = true;
        BigInteger vittne;
        BigInteger q;
        BigInteger b;

        // MillerRabin Algotitmen
        for (int i = 0; i < nvarv; i++)
        {
            vittne = Slump(3, false);

            while (vittne < 3) { vittne = Slump((int)Math.Sqrt(qstrl),...
            ...false); }

            q = PminusEtt;
            probPrime = false;

            //1
            if (BigInteger.GreatestCommonDivisor(vittne, PrimKandidat) > 1)
            { return false; }//probPrime = false; break; }

            //2 rensa 2-faktorer
            int s = 0;
            while (q % 2 == 0) { q = q / 2; s++; }
```

```
            //3
            b = BigInteger.ModPow(vittne, q, PrimKandidat);

            //4
            if ((b - 1) % PrimKandidat == 0)
            { probPrime = true; }
            else
            {
                //5
                for (int j = 0; j < s; j++)
                {
                    if ((b + 1) % PrimKandidat == 0)
                    { probPrime = true; break; }
                    b = BigInteger.ModPow(b, 2, PrimKandidat);
                }
            }
            //Console.WriteLine(probPrime);
            //double P = Math.Pow(0.25, i + 1);

            if (probPrime == false) { break; }
        }
        DateTime tt = DateTime.Now;
        Console.WriteLine("Tid MilleRabin " + (tt - st).Seconds +...
        ..." s " + (tt - st).Milliseconds + " ms");
        return probPrime;


    }

    //********************** SLUMPGENERATOR ***************************
    public BigInteger Slump(int strl, bool prim)
    {
        RNGCryptoServiceProvider rnd = new RNGCryptoServiceProvider();
        byte[] tal = new byte[1];
        string SlumptalString = "";
        BigInteger slumpTal;

        for (int i = 0; i < strl; i++)
        {
            rnd.GetBytes(tal);

            if (tal[0] < 250) { SlumptalString = SlumptalString +...
            ...Convert.ToString(tal[0] % 10); }
            else { i--; }

        }

        slumpTal = BigInteger.Parse(SlumptalString);

        if (prim)
        {
            List<int> Primvektor = new List<int>();

            if (boost1)
            {
```

```
            if ((int)(EraFaktor * PrimDens(strl)) > 0)
            { Primvektor = EratosthenesSoll((int)(EraFaktor...
            ...* PrimDens(strl))); }
            else
            { Primvektor.Add(2); }
        }
        Console.WriteLine("--------------------");
        Console.WriteLine("Primtestar");
        DateTime start2 = DateTime.Now;
        if (slumpTal % 2 == 0)
        { slumpTal = slumpTal + 1; }
        int antalTest = 0;



        if (boost1)
        {
            while (!PrimTestMillerRabin(slumpTal,...
            ...MillerRabinHardness, Primvektor))
            { slumpTal = slumpTal + 2; antalTest++; }

        }
        DateTime sluttid2 = DateTime.Now;
        Console.WriteLine("Antal primtest: " + antalTest);
        if (boost1)
        {
            Console.WriteLine("Varav Boost: " + inik);
            inik = 0;
            Console.WriteLine("Tid: " + (sluttid2 - start2).Seconds...
            ...+ " s  " + (sluttid2 - start2).Milliseconds + " ms");
            Console.WriteLine("Snittid per test: " + (1000 *...
            ...(sluttid2 - start2).Seconds + (sluttid2 - start2). ...
            ...Milliseconds) / antalTest + " ms");
        }
        //Console.WriteLine("slumptal: " + slumpTal);
    }
    //Console.WriteLine("slumptal: " + slumpTal);
    return slumpTal;

}
//********************* TEXT TILL TAL ************
public BigInteger TextToNum(string TXT)
{
    BigInteger mess = new BigInteger(0);
    int n = TXT.Length;
    BigInteger expo = new BigInteger(1);
    int asvalu;
    foreach (char s in TXT)
    {
        asvalu = Convert.ToInt32(s);
        //Console.WriteLine(asvalu);
        expo = expo * 1000;
        mess = mess + asvalu * expo;
    }
    Console.WriteLine("text i nummer: " + mess);
    return mess;
```

```
}
//********************* TAL TILL TEXT ***********
public string NumToText(BigInteger NUM)
{
    BigInteger tioPotens = new BigInteger(1);

    string meddelande = "";

    int tal = 0, koll = 0;
    while (NUM > tioPotens) { tioPotens = tioPotens * 10; koll++; }
    for (int i = 0; i < koll; i = i + 3)
    {
        NUM = NUM / 1000;
        tal = (int)(NUM % 1000);
        meddelande = meddelande + Convert.ToChar(tal);
    }
    return meddelande;
}

//********** Hitta invers d=e^-1 (mod m) ************
public BigInteger InvModFind(BigInteger Number, BigInteger Modul)
{
    Console.WriteLine("-------------------------------");
    Console.WriteLine("Beraknar inversen");
    //Console.WriteLine("Beraknar Inversen d sa att d*\n"...
    ...+ Number + "\n=1 mod\n " + Modul);
    //Finn inversen d till Number sa att d*Number=1 mod Modul
    //Box algorithm
    BigInteger rest = new BigInteger(1);
    BigInteger qvot = new BigInteger(1);
    BigInteger Invers = new BigInteger(1);
    List<BigInteger> q = new List<BigInteger>();
    List<BigInteger> r = new List<BigInteger>();
    List<BigInteger> above = new List<BigInteger> { 0, 1 };
    List<BigInteger> below = new List<BigInteger> { 1, 0 };

    //................. Euclid .........................
    int i = 0;
    qvot = (Modul) / Number;
    rest = (Modul) - Number * qvot;
    q.Add(qvot);
    r.Add(rest);
    qvot = Number / rest;
    rest = Number % rest;
    q.Add(qvot);
    r.Add(rest);
    while (r[i + 1] != 0)
    {
        q.Add(r[i] / r[i + 1]);
        r.Add(r[i] % r[i + 1]);
        //Console.WriteLine(q[i+1] + " qvot " + r[i+1] + " rest");
        i++;

    }
    Console.WriteLine("Euclid loops: " + i);
    i = 1;
```

```
            //.............. Box ...........
            while (above[i] != Modul)
            {
                i++;
                above.Add(q[i - 2] * above[i - 1] + above[i - 2]);
                below.Add(q[i - 2] * below[i - 1] + below[i - 2]);
                //Console.WriteLine(above[i]+" above "+below[i]+" below");
            }
            Console.WriteLine("Box loops: " + i);

            if (above[i - 1] >= 0 && BigInteger.ModPow(above[i - 1]...
            ...* Number, 1, Modul) == 1)
            { Invers = above[i - 1]; Console.WriteLine("Hit3"); }
            if (above[i - 1] >= 0 && BigInteger.ModPow(-(above[i - 1]...
            ...- Modul) * Number, 1, Modul) == 1)
            { Invers = -(above[i - 1] - Modul); Console.WriteLine("Hit4"); }
            Console.WriteLine("Returnerar inversen");
            //Console.WriteLine("Returnerar inversen:\n " + Invers);
            return Invers;
        }
    }

    //***************** Static variabler ********

    static bool boost1 = true;
    static int inik = 0;
    static readonly int MillerRabinHardness = 100;
    static readonly int pstrl = 173;
    static readonly int qstrl = 173;
    static readonly double EraFaktor = 10;
    static bool mrStandAlone = false;
    static bool rsaStandAlone = !mrStandAlone;


    //*************MAIN*******************

    static void Main(string[] args)
    {
        if (rsaStandAlone)
        {
            Console.WriteLine("Antal siffror i p: " + pstrl);
            Console.WriteLine("Antal siffror i q: " + qstrl);
            Console.WriteLine("MillerRabin loops: " + MillerRabinHardness);
            Console.WriteLine("MillerRabin sakerhet: " +...
            ...(1 - Math.Pow(0.25, MillerRabinHardness)) * 100);
            Console.WriteLine("Ange ett meddelande!");
            string meddelande = Console.ReadLine();

            DateTime start = DateTime.Now;
            Algos algos = new Algos();


            BigInteger mIn;

            BigInteger p = algos.Slump(pstrl, true);
```

```
        mIn = algos.TextToNum(meddelande);

        string estring = "32132135454541313513132325";

        Console.WriteLine("p= " + p);
        BigInteger n = new BigInteger(0);
        BigInteger eulerfi = new BigInteger(0);

        BigInteger e;
        e = BigInteger.Parse(estring);
        BigInteger d = new BigInteger(1);
        BigInteger mUt;
        BigInteger c = new BigInteger(0);
        BigInteger GCD = new BigInteger(0);
        BigInteger j = new BigInteger(1);

        Random slump = new Random();
        n = p * q;

        d = algos.InvModFind(e, eulerfi);



        //************Enkrypt**********************
        //Console.WriteLine("Krypterar \n"+mIn);
        Console.WriteLine("Krypterar");
        DateTime s1 = DateTime.Now;
        c = BigInteger.ModPow(mIn, e, n);
        DateTime t1 = DateTime.Now;
        Console.WriteLine("Tid Kryptering: " + (t1 - s1).Milliseconds);
        Console.WriteLine("Cipher:\n" + c + "\n");

        //***********Dekrypt**********************
        //Console.WriteLine("Dekrypterar \n" + c);
        Console.WriteLine("Dekrypterar");
        DateTime s2 = DateTime.Now;
        mUt = BigInteger.ModPow(c, d, n);
        DateTime t2 = DateTime.Now;
        Console.WriteLine("Tid Deryptering: " + (t2 - s2).Milliseconds);
        Console.WriteLine("Meddelande ater:\n" + mUt + "\n");
        Console.WriteLine(algos.NumToText(mUt));

        //tid

        DateTime sluttid = DateTime.Now;
        Console.WriteLine((sluttid - start).Seconds +...
        ..." s och " + (sluttid - start).Milliseconds + " ms");


        Console.ReadKey();
    }
    if (mrStandAlone)
    {
        string pstring = "13138797184561042259219955";
        Algos a = new Algos();
        List<int> l = new List<int>();
```

```
                    l = a.EratosthenesSoll(500000);
                    Console.WriteLine(a.PrimTestMillerRabin...
                    ...(BigInteger.Parse(pstring), 10, l));

                    Console.ReadKey();
                }
            }
        }
    }
```