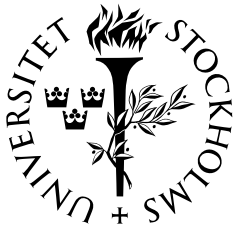

SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Asymmetriska krypteringssystem: hur de är konstruerade och vilka matematiska problem de bygger på

av

Sara Leufstadius

2010 - No 4

Asymmetriska krypteringssystem: hur de är konstruerade och vilka matematiska problem de bygger på

Sara Leufstadius

Självständigt arbete i matematik 15 högskolepoäng, grundnivå

Handledare: Rikard Bøgvad

2010

Sammanfattning

I denna uppsats behandlas två moderna krypteringssystem; RSA och ElGamal. Vi visar hur man använder dessa, hur nycklar genereras och hur man sedan krypterar och dekrypterar meddelanden. För båda systemen gäller att de bygger på var sitt svårlöst matematiskt problem, i RSA:s fall handlar det om primtalsfaktorisering och för ElGamal är det fråga om det diskreta logaritmproblemet. Utifrån respektive problem konstrueras en envägsfunktion som är lätt att beräkna åt ena hållet, men om man däremot vill finna dess invers är det i praktiken omöjligt om man inte också har tillgång till extra information, vilken gör beräkningen av inversen genomförbar. Vi studerar också några olika algoritmer och metoder för att lösa de två problem som systemen bygger på. Till sist tar vi upp något som berör båda krypteringssystemen, nämligen hur man avgör om ett tal är ett primtal.

Innehåll

1 Introduktion

Kryptering - läran om all sorts hemlig skrift, hur man använder (matematiska) metoder för att dölja meddelanden och budskap i någon sorts kommunikation - har troligtvis existerat sedan människor började överföra budskap till varandra. Kungar, generaler och andra makthavare har alltid varit i behov av sekretess, främst för att dölja meddelanden för fiender, för att kunna uppnå en effektiv kommunikation då de styrde sina länder, manövrerade armeér och tog viktiga beslut.
Därför utvecklades chiffer, hemliga skriftsystem där klartext (det ursprungliga meddelandet innan krypteringen tagit vid) överförs till kryptotext (meddelandet efter krypteringen) för att dessa människor skulle kunna tillgodose sina behov.
I denna uppsats kommer vi att studera två asymmetriska krypteringssystem, RSA och ElGamal, och redogöra för vilka matematiska problem de bygger på. Vi kommer även att studera vilka metoder som finns för att lösa dessa problem, metoder som skulle kunna göra krypteringssystemen sårbara om de skulle utvecklas så att de kan användas för att bryta systemen.

Som uppvärmning kommer vi först att titta på vad kryptering är och hur den delas in i symmetrisk och asymmetrisk kryptering. Vi kommer här även att studera begreppet öppen-nyckelkryptering lite närmare, samt diskutera kort vilka problem som är enkla att beräkna. Avsnitt 2 ägnar vi sedan åt RSA-systemet. Detta bygger på svårigheten att faktorisera stora tal i primtal varför vi här också studerar olika metoder för primtalsfaktorisering. Avsnitt 3 viger vi åt ElGamal-systemet där vi börjar med en beskrivning av Diffie-Hellmans nyckelutbyte som ligger till grund för detta krypteringssystem. Själva krypteringen och svårigheten att bryta denna bygger i detta fall på det diskreta logaritmproblemet, som vi här också studerar några algoritmer för att lösa. Till sist viker vi det avslutande avsnittet åt något som berör båda krypteringssystemen, det vill säga primtalstest och primtalsgenerering. Vi kommer att ta upp frågan hur man testar om ett tal är ett primtal och titta på några algoritmer och metoder för detta.
Framställningen i denna uppsats är i huvudsak baserad på [3], [5], [6] och [7].

1.1 Enkel substitutionskryptering

Kryptering, eller kryptografi, är indelat i 2 grenar, transposition och substitution. Vid transposition handlar det helt enkelt om att man använder sig av anagram. De individuella bokstäverna behåller sin ursprungliga betydelse men byter plats med varandra.
Vid substitution däremot så står bokstäverna kvar på sin plats men får en annan betydelse. I det allra enklaste fallet, ett monoalfabetiskt substitution-

schiffer, krypteras varje bokstav på samma sätt genom hela meddelandet.

1.1.1 Caesarkryptot

Som ett exempel på ett monoalfabetiskt substitutionschiffer kan vi ta Caesarkryptot. Den romerske kejsaren Julius Caesar krypterade meddelanden han skickade så att fiender inte skulle kunna läsa dem helt enkelt genom att han bytte ut varje bokstav i klartexten mot en bokstav 5 steg längre fram i alfabetet, som då blir den så kallade chiffertextbokstaven. Ska vi exempelvis skriva ett a byter vi ut det mot ett f. För att dekryptera meddelanden får vi helt enkelt skifta varje bokstav 5 steg bakåt istället. Bokstäver i slutet på alfabetet skiftas till de i början, vi kan enklast visualisera detta genom att skriva alfabetet i en ring istället för den vedertagna uppställningen i en rad. För en närmare beskrivning, se till exempel [6] sid. 1-3.

1.1.2 Viegenére-chiffret

Ett lite mer sofistikerat substitutionschiffer är det polyalfabetiska. Det karaktäriseras av att samma klartextbokstav kan krypteras på olika sätt beroende på dess plats i meddelandet. Idén är att man använder flera enkla monoalfabetiska substitutionschiffer. Som ett exempel på ett polyalfabetiskt substitutionschiffer kan vi nämna Viegenére-chiffret, som fick sin slutgiltiga form av Blaise de Viegenére på 1500-talet. Vi ger en kort beskrivning av hur man använder Vigenére-chiffret.
När vi vill kryptera och dekryptera meddelanden med detta så behöver vi en så kallad Viegéneretabell. Denna består av 29 rader, var och en bestående av ett alfabet som vart och ett är förskjutet ett steg i förhållande till alfabetet i raden ovanför. Man kan säga att varje rad motsvarar ett Caesarkrypto där förskjutningen är 1-29 steg, beroende på var i tabellen det står. Högst upp skrivs klartextalfabetet, det vill säga det utan några förskjutningar. Sedan har vi för varje bokstav vi skriver 29 möjliga krypteringsalternativ, beroende på vilket av de 29 kryptoalfabetena vi väljer. För att kunna tolka budskapet måste vår tilltänkte mottagare förstås veta vilken rad i tabellen som har använts för att kryptera varje bokstav och därför behöver vi också ett överenskommet system för hur radväxlingarna ska se ut. Lösningen på detta är att vi kommer överens om ett nyckelord.
Säg nu att vi använder nyckelordet guide och vill kryptera natthimlen. Den första bokstaven i klartexten, n, ska vi då kryptera med det alfabet som börjar med den första bokstaven i nyckelordet, det vill säga g. Detta kommer att ge oss att n ska krypteras med ett t. Den andra bokstaven i vår klartext, a, krypterar vi med det alfabet som börjar med den andra bokstaven i nyckelordet, det vill säga u. Detta ger att a ska krypteras med ett u. Sedan fortsätter vi på samma sätt. När vi använt hela vårt nyckelord börjar börjar vi om med den första bokstaven i nyckeln. För en utförligare beskrivning av

detta chiffer och exempel på en Vigenéretabell, se [9] kap. 2.

1.2 Symmetrisk och asymmetrisk kryptering

För alla typer av kryptografisk substitution används termerna chiffer eller krypto, och ett allmänt sådant består av en kombination av två delar; en algoritm, den så kallade generella krypteringsmetoden man använder (till exempel Caesarkryptot eller Viegenére-chiffret) samt en nyckel som bestämmer hur den valda algoritmen ska tillämpas i det aktuella fallet. I exempelvis Caesarkryptot gäller att nyckeln är antalet steg som bokstäverna flyttas och krypterar vi med Viegenére-chiffret är nyckeln det valda nyckelordet.
Både Caesarkryptot och Viegenére-chiffret är traditionella former av kryptering, man brukar kalla dem symmetrisk kryptering. Här används samma nyckel för att kryptera och dekryptera ett meddelande. I dessa system är alltså krypterings- och dekrypteringsnycklarna antingen identiska eller så kan de enkelt beräknas från varandra. Detta ger en del problem, exempelvis så måste de som vill kryptera meddelanden till varandra hitta ett säkert sätt att varsko varandra vilken nyckel de ska använda för krypteringen - detta utan att en eventuell fiende råkar få höra detta. Om deras nyckel skulle bli känd kan ju då vem som helst dekryptera informationen. Med detta problem som utgångspunkt utvecklades de asymmetriska krypteringssystemen.
Ett asymmetriskt krypteringssystem bygger på användning av öppna nycklar (eng. public key cryptography), därför kallas de även för öppen-nyckelsystem. I denna sortens kryptering använder man två nycklar, en privat för dekryptering och en publik för kryptering. Asymmetrin kommer av att dessa nycklar är helt olika och att man inte kan beräkna eller gissa den andra utifrån att man vet hur en av dem ser ut. Detta får till följd att man i de asymmetriska krypteringssystemen inte behöver utväxla några nycklar, vilket underlättar krypteringen.
Varje asymmetriskt krypteringssystem bygger på ett svårlöst matematiskt problem, utifrån vilket man konstruerar en envägsfunktion som är lätt att beräkna åt ena hållet. Om man däremot vill finna dess invers är det i praktiken omöjligt. Har man emellertid också tillgång till extra information är beräkningen av inversen genomförbar, vilket brukar kallas en matematisk trapdoor.

För att vi lättare ska kunna förstå hur denna sorts kryptering går till ska vi gå igenom följande exempel. Vi tänker oss något slags digitalt arkiv vari vem som helst kan publicera sin publika nyckel som alla sedan kan använda för att skicka krypterade meddelanden till en utvald person. Säg att Bob vill sända Alice ett krypterat meddelande (vi följer här traditionen för att namnge personerna som ska utbyta konfidentiell information). Han letar då upp hennes publika nyckel och krypterar ett meddelande med denna. Alice använder sedan sin privata nyckel, som hon håller hemlig, för att dekryptera

Bobs meddelande och eftersom Bob använde hennes publika nyckel är det endast Alice som kan dekryptera detta meddelande.
Idén till öppen-nyckelkryptering lades för första gången fram i en uppsats år 1976 av Whitfield Diffie och Martin Hellman ([5], sid 160). Detta kallas allmänt för Diffie-Hellmans nyckelutbyte och vi kommer att titta närmare på denna idé i avsnittet om krypteringssystemet ElGamal.

1.3 Enkla och svårare beräkningar

Innan vi går över till de två krypteringssystemen vi ska studera i denna uppsats så ska vi kort nämna vad som faktiskt är lätt att beräkna, något vi har nytta av för flera av de algoritmer vi ska titta på. Vi kommer i många av de algoritmer vi tar upp att använda oss av beräkningar där $SGD(a, b), a$ och b heltal, ingår. Att beräkna $SGD(a, b)$ är ett exempel på ett enkelt problem och bygger på Euklides algoritm, vilken är mycket effektiv även för stora tal a och b (se [6] kap. 1).
Något annat vi kommer använda i flera av algoritmerna vi ska beskriva är att det är enkelt att beräkna stora potenser av ett heltal g modulo ett annat heltal n, där n har hundratals siffror. Den specifika algoritmen brukar oftast benämnas the fast powering algorithm, se till exempel [6] sid. 25 för en närmare beskrivning.
De svårlösta matematiska problem som RSA och ElGamal bygger på skulle faktiskt kunna lösas med någon av de algoritmer som finns idag, problemet är bara att de är alltför långsamma. I denna uppsats kommer vi inte att ta upp hur snabba de olika algoritmerna vi beskriver är, men det är såklart en väsentlig del av deras implementering.

2 RSA

Vi är nu redo för att studera ett välanvänt, välkänt och (åtminstone till idag, [7] sid. 160) säkert asymmetriskt krypteringssystem, nämligen RSA. RSA, som var det första öppen-nyckelsystem som uppfanns, beskrevs för första gången i en uppsats år 1978 ([5], sid. 160) publicerad av Ronald Rivest, Adi Shamir och Leonard Adleman, vilka alltså även, som synes, fick ge namn åt krypteringssystemet.

Det svårlösta matematiska problem som RSA bygger på är primtalsfaktorisering, det vill säga att faktorisera positiva, sammansatta heltal i deras primtalsfaktorer. Ett (mycket) enkelt exempel på detta är $15 = 3 \cdot 5$.
I RSA-systemet ska heltalet vara mycket stort och en produkt av endast två stora primtal. [5] sid. 162 föreslår en storlek på primtalen av $\approx 10^{80}$ eller större, vilket gör att heltalet får storleken $\approx 10^{160}$.
Vi ska nu se hur krypteringen i RSA-systemet går till och vi börjar med att studera hur man går till väga för att generera den privata och den publika nyckeln.

2.1 Generering av nycklar

Först genererar vi slumpmässigt och oberoende två stora primtal; p och q. Dessa två tal måste hållas hemliga. (Primtalsgenerering och hur man testar om det genererade talet verkligen är ett primtal, som för övrigt också spelar en huvudroll i ElGamal-systemet, ska vi titta närmare på i avsnitt 4). Sedan beräknar vi deras produkt: $n = pq$ och väljer slumpmässigt ett heltal e med

$$1 < e < f(n) = (p-1)(q-1)$$

och

$$SGD(e, (p-1)(q-1)) = 1.$$

e kommer då alltid att vara udda eftersom $(p-1)$ är jämnt. Talet n kallas vår RSA-modulus och e kallas vår RSA-krypteringsexponent. Med hjälp av Euklides algoritm beräknar vi också ett heltal d med

$$1 < d < (p-1)(q-1)$$

och

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

Eftersom $SGD(e, (p-1)(q-1)) = 1$ existerar ett sådant tal d, och vi beskriver nu lösningsförfarandet för hur man hittar en sådan lösning. För att förenkla notationen något, sätt $(p-1)(q-1) = b$. Nu fås

$$de \equiv 1 \pmod{b} \iff b \mid (de-1) \iff by = de - 1 \text{ där } y \in \mathbb{Z}.$$

Vi får alltså

$$de + my = 1 \quad \text{där } m = -b.$$

Vi vet nu att eftersom $SGD(b, e) = SGD(m, e) = 1$ så existerar det alltid heltal n och k sådana att

$$mn + ek = 1. \tag{1}$$

Detta är en diofantisk ekvation och vi ska med ett numeriskt exempel se hur man löser en sådan med hjälp av Euklides algoritm.

Exempel 1. *Sätt $m = 7$ och $e = 5$ i (1) så fås $7n + 5k = 1$. Vi tillämpar Euklides algoritm:*

$$7 = 1 \cdot 5 + 2,$$

$$5 = 2 \cdot 2 + 1.$$

Nu ger den senare och den förra likheten i tur och ordning att

$$1 = 1 \cdot 5 - 2 \cdot 2 = 1 \cdot 5 - 2 \cdot (1 \cdot 7 - 1 \cdot 5) = 3 \cdot 5 - 2 \cdot 7.$$

Så med Euklides algoritm har vi nu fått att

$$3 \cdot 5 - 2 \cdot 7 = 1.$$

Reducerar vi detta modulo 7 får vi

$$3 \cdot 5 \equiv 1 \pmod{7}.$$

Exemplet visar att det existerar ett tal d sådant att

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

För att kunna beräkna detta d så måste vi veta produkten $(p-1)(q-1)$ och detta innebär i praktiken att vi måste veta vilka talen p och q är och inte bara deras produkt n. d kallas dekrypteringsexponenten.
Vår publika nyckel är nu paret (n, e) som vi kan publicera i något allmänt arkiv eller någon databas, och vår privata nyckel är d, som vi ska hålla hemlig.

2.2 Kryptering och dekryptering med RSA

Nu när vi har båda våra nycklar kan vi börja kryptera meddelanden. Säg att vi vill skriva ett krypterat meddelande till Alice, som också hon har publicerat sin publika nyckel (n, e) i det allmänna arkivet. Först skriver vi ett klartextmeddelande och antar för enkelhetens skull att vi omvandlar detta till siffror, närmare bestämt binära tal, med hjälp av till exempel ASCII (se [9] sid. 273). Antag nu att vårt meddelande är m med $0 \leq m < n$.
Vi krypterar m genom att beräkna

$$c \equiv m^e \pmod{n}$$

där talet e är den ena delen i Alices publika nyckel. Detta är möjligt eftersom vi genom det tänkta arkivet har tillgång till Alices publika nyckelpar (n, e). c är nu den krypterade texten som vi skickar till Alice.

Nu måste Alice dekryptera vårt meddelande med sin privata nyckel d. För att bättre förstå hur hon gör detta ska vi först titta på de två följande satserna.

Sats 1. *Fermats lilla sats.*
Låt p vara ett primtal och låt a vara ett godtyckligt heltal. Då gäller

$$a^{p-1} \equiv \begin{cases} 1 \pmod{p} & \text{om } p \nmid a, \\ 0 \pmod{p} & \text{om } p \mid a. \end{cases}$$

Bevis. Om $p \mid a$ är det klart att varje positiv heltalspotens av a är delbar med p, så vi behöver bara undersöka fallet $p \nmid a$.
Betrakta talen $a, 2a, 3a \ldots (p-1)a$ och låt $b_1, b_2 \ldots b_{p-1}$ vara deras rester vid division med p. Notera att för dessa rester gäller $1 \leq b_i \leq (p-1)$ och $ai \equiv b_i \pmod{p}$. Vi har även att $b_1, b_2 \ldots b_{p-1}$ alla är olika och för att inse detta resonerar vi på följande sätt: välj först två av talen, till exempel b_i och b_j och antag att $b_i = b_j$. Då fås följande ekvivalenser:

$$ai \equiv aj \pmod{p} \iff a(i-j) \equiv 0 \pmod{p} \iff p \mid a(i-j).$$

Vi vet nu att antingen gäller $p \mid a$ eller $p \mid (i-j)$. Men vi har ju att $p \nmid a$ så därför gäller $p \mid (i-j)$ och av detta följer $i = j$ eftersom både i och j ligger mellan 1 och $(p-1)$.
Talen b_i är alltså $(p-1)$ stycken och alla olika, vilket betyder att de utgör talen $1, 2, 3 \ldots (p-1)$, möjligen i en annan ordning. Av detta följer att

$$b_1 \cdot b_2 \cdot \cdots \cdot b_{p-1} = 1 \cdot 2 \cdot \cdots \cdot (p-1) = (p-1)!$$

Vi får nu även

$$a \cdot 2a \cdot \cdots \cdot (p-1)a = a^{p-1}(p-1)!$$

och med $ai \equiv b_i \pmod{p}$ följer det att

$$a^{p-1}(p-1)! \equiv b_1 \cdot b_2 \cdot \cdots b_{p-1} \equiv (p-1)! \pmod{p}$$

$$\iff a^{p-1}(p-1)! \equiv (p-1)! \pmod{p}.$$

Men $p \nmid (p-1)!$ eftersom om det gällde att $p \mid (p-1)!$ så skulle p dela någon av faktorerna i produkten $(p-1)!$ och dessa är alla $< p$. Alltså fås

$$a^{p-1} \equiv 1 \pmod{p}$$

och beviset är klart. □

Sats 2. *Låt (n, e) vara en publik RSA-nyckel och låt d vara den motsvarande privata nyckeln. Då gäller*

$$(m^e)^d \equiv m \pmod{n}$$

för godtyckligt heltal m med $0 \leq m < n$.

Bevis. På grund av att vi har

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

så finns ett heltal l med

$$ed = 1 + l(p-1)(q-1).$$

Därför gäller följande likheter:

$$(m^e)^d = m^{ed} = m^{1+l(p-1)(q-1)} = m(m^{(p-1)(q-1)})^l.$$

Det följer sedan att

$$(m^e)^d \equiv m(m^{(p-1)})^{(q-1)l} \equiv m \pmod{p}.$$

Om p inte delar m följer den sista ekvivalensen från Fermats lilla sats, sats (1), annars är den trivial eftersom båda sidor av kongruensen är $0 \pmod{p}$. Analogt visas att

$$(m^e)^d \equiv m \pmod{q}.$$

På grund av att p och q är skilda primtal fås därför

$$(m^e)^d \equiv m \pmod{n}.$$

□

Så Alice har nu alltså av oss fått chiffertexten c skickad till sig och hon kan därför få fram klartexten genom

$$c^d \equiv (m^e)^d \equiv m \pmod{n} \iff c^d \equiv m \pmod{n}.$$

Detta är enkelt för Alice eftersom hon har tillgång till sin privata nyckel d och faktoriseringen $n = pq$. Om en fiende däremot skulle få tag på chiffertexten c men inte vet hur man ska faktorisera n så kommer han eller hon att ha svårt att lösa ekvationen

$$c \equiv m^e \pmod{n} \quad \text{för } m.$$

Notera också att vår privata nyckel d kan beräknas från krypteringsexponenten e om n:s primtalsfaktorer p och q är kända, så om vår fiende visste eller kom på ett sätt att faktorisera stora tal i primtal så skulle han eller hon ha lätt att, utifrån n, kunna få reda på p och q och därmed läsa alla hemliga meddelanden vi skickar. Att beräkna d från (n, e) är lika svårt som att finna primtalsfaktorerna p och q av n, se [3] sid 145, så detta visar att säkerheten av RSA bygger på (precis som vi fick reda på i inledningen av denna uppsats) svårigheten att faktorisera stora heltal i primtal.
Det är ännu inte bevisat att det är svårt att faktorisera RSA-talen n, men om p och q är tillräckligt stora så finns det till dags dato ingen som vet hur man ska faktorisera n i dessa. Innan vi går över till att titta lite närmare på olika metoder för primtalsfaktorisering ska vi med ett enkelt exempel illustrera hur nyckelgenerering, kryptering och dekryptering går till med RSA.

2.3 Ett enkelt krypteringsexempel

Vi ska nu få se hur Alice och Bob gör när de använder sig av krypteringssystemet RSA för att skicka meddelanden till varandra. För överskådlighetens skull låter vi dem använda små tal, vilket i praktiken gör krypteringen mycket osäker eftersom det då är lättare för en fiende att faktorisera n, men det blir lätt otympligt att exemplifiera denna kryptering med verklighetstrogen storlek på siffrorna.

Antag nu att Alice vill skicka ett krypterat meddelande till Bob. Till att börja med så måste Bob då konstruera sin publika och sin privata nyckel. Han väljer först två primtal som han håller hemliga, $p = 1223$ och $q = 1987$. Han beräknar sedan $n = 1223 \cdot 1987 = 2430101$. Bob väljer nu också en publik krypteringsexponent $e = 948047$ med egenskapen att

$$SGD(e, (p-1)(q-1)) = SGD(948047, 2426892) = 1.$$

Alice skriver en klartext m och omvandlar denna till siffror, säg exempelvis att hon får $m = 1070777$ vilket satisfierar $0 \leq m < n$.
Hon letar sedan upp Bobs publika nyckel $(n, e) = (2430101, 948047)$ som

han publicerat i något slags offentligt arkiv och beräknar kryptotexten c på följande sätt:

$$c \equiv m^e \pmod{n} \iff c \equiv 107077^{948047} \equiv 1473513 \pmod{2430101}.$$

Alice skickar sedan c till Bob.
Bob vet ju nu att $(p-1)(q-1) = 1222 \cdot 1986 = 2426892$ så han har tillräckligt med extra information för att med hjälp av Euklides algoritm lösa ut d i ekvationen

$$ed \equiv 1 \pmod{(p-1)(q-1)} \iff 948047 \cdot d \equiv 1 \pmod{2426892}.$$

Han kommer då finna att $d = 1051235$. Han tar nu kryptotexten $c = 1473513$ och beräknar

$$c^d \pmod{n} \iff 1473513^{1051235} \equiv 1070777 \pmod{2430101}.$$

Bob har nu fått fram klartexten m som Alice skrev.

2.4 Olika metoder för primtalsfaktorisering

Primtalsfaktoriseringen av ett positivt heltal a är representationen av a som en produkt av primtal. Vi vet ju att varje heltal $a \geq 2$ är, bortsett från ordningen, en unik produkt av primtal och problemet att hitta primtalsfaktoriseringen av a brukar kallas heltalsfaktoriseringsproblemet (eng. integer factorization problem). Krypteringssystemets RSA:s säkerhet bygger, som vi tidigare sett, på att man ännu inte känner till några effektiva algoritmer för att lösa detta problem. Dock är det inte bevisat att heltalsfaktoriseringsproblemet faktiskt *är* svårt att lösa, så utvecklas en mer effektiv metod för att faktorisera heltal i primtal är RSA:s säkerhet hotad. Under årens lopp har det faktiskt utvecklats många effektiva faktoriseringsalgoritmer vilket gjort att antalet siffror som måste användas i RSA-talen n för att RSA ska anses som säkert har ökat från 512 till 1024 bits ([3], sid. 171).
För att faktorisera ett tal kan det ju vara bra att först försäkra sig om att talet verkligen är sammansatt och det gör man med ett så kallat primtalstest (eng. primality test). Detta problem tar vi upp i avsnitt 4 och för närvarande antar vi att talen vi ska faktorisera har visat sig vara sammansatta med hjälp av ett sådant test. I alla nedanstående metoder betecknas talet som vi ska faktorisera med n.

2.4.1 Klassiska metoder

Vi börjar med att gå igenom några äldre metoder som fortfarande har viss relevans för dagens algoritmer.

- **Prövning med division** (eng. trial division).
  Detta är den troligen äldsta metoden för att hitta n:s primtalsfaktorer, och går helt enkelt ut på att dela n med alla primtal upp till $\sqrt{n}$. Om $n < 10^8$ ([7] sid. 207) borde detta inte vara något problem för dagens datorer, men som vi tidigare fick reda på är RSA-talen n av storlek $\approx 10^{160}$ vilket gör att denna metod lätt blir otymplig om man vill bryta RSA.

- **Fermats faktoriseringsmetod.**
  Denna metod brukar ibland också benämnas differensen av kvadrater (eng. difference of squares) och utvecklades av Fermat år 1643 ([7] sid. 201). Den moderna faktoriseringsmetoden Quadratic sieve bygger på utveckling och förbättring av Fermats metod. Allt bygger på en av de enklaste identiteterna i matematiken, nämligen $a^2 - b^2 = (a-b)(a+b)$, det vill säga att skillnaden av två kvadrater är lika med en produkt. Användningen inom faktoriseringen ser ut såhär:
  För att faktorisera n letar vi efter ett heltal b sådant att $n + b^2$ är en perfekt kvadrat, till exempel $n + b^2 = a^2$. Då har vi att $n = a^2 - b^2 = (a + b)(a - b)$ och vi har hittat en faktorisering av n.
  Om n är stort, som i RSA, kan det dock vara svårt att hitta ett b sådant att $n + b^2$ är en perfekt kvadrat. Följande trick kan då förenkla proceduren: hitta tal sådana att $kn = a^2 - b^2$ för något positivt heltal k. Då finns det nämligen en chans att n har en icke-trivial faktor gemensam med både $(a - b)$ och $(a + b)$. Dessa hittar vi förstås genom att med hjälp av Euklides algoritm beräkna $SGD(n, a+b)$ och $SGD(n, a - b)$. Förenklat kan vi då säga att vi söker efter tal a och b sådana att $a^2 \equiv b^2 \pmod{n}$.
  I praktiken använder man följande procedur, som i den ena eller andra formen ligger bakom många av dagens faktoriseringsmetoder ([6] sid. 138).

  1. Bygg en relation
     Hitta många positiva heltal $a_1, a_2, \ldots a_r$ med egenskapen att kvantiteten $c_i \equiv a_i^2 \pmod{n}$ kan faktoriseras som en produkt av små primtal.
  2. Eliminering
     Beräkna en produkt $c_{i_1} \cdot c_{i_2} \cdot \cdots \cdot c_{i_s}$ av några av talen c_i så att varje primtal i produkten är en jämn potens. Då är $c_{i_1} \cdot c_{i_2} \cdot \cdots \cdot c_{i_s} = b^2$ en perfekt kvadrat.
  3. Beräkning av SGD
     Låt nu $a = a_{i_1} \cdot a_{i_2} \cdot \cdots a_{i_s}$ och beräkna $d = SGD(n, a - b)$. Eftersom

     $$a^2 = (a_{i_1} \cdot a_{i_2} \cdot \cdots \cdot a_{i_s})^2 \equiv a_{i_1}^2 \cdot a_{i_2}^2 \cdot \cdots a_{i_s}^2 \equiv c_{i_1} \cdots c_{i_s} \equiv b^2 \pmod{n}$$

så finns det en rimlig chans att d är en icke-trivial faktor av n.

Vi tittar på ett exempel för att illustrera metoden.

Exempel 2. *Antag att vi vill faktorisera* $n = 914387$ *med hjälp av metoden som beskrivits ovan. Först måste vi då leta efter heltal* a *sådana att* $a^2 \pmod{n}$ *är en produkt av små primtal. Vi observerar att*

$$\begin{aligned} 1869^2 &\equiv 750000 \pmod{914387} \quad \textit{och} \quad 750000 = 2^4 \cdot 3 \cdot 5^6, \\ 1909^2 &\equiv 901120 \pmod{914387} \quad \textit{och} \quad 901120 = 2^{14} \cdot 5 \cdot 11, \\ 3387^2 &\equiv 499125 \pmod{914387} \quad \textit{och} \quad 499125 = 3 \cdot 5^3 \cdot 11^3. \end{aligned}$$

Inget av talen till höger är en kvadrat, men om vi multiplicerar dem fås:

$$\begin{aligned} 1869^2 \cdot 1909^2 \cdot 3387^2 &\equiv 750000 \cdot 901120 \cdot 499125 \pmod{914387} \\ &\equiv (2^4 \cdot 3 \cdot 5^6)(2^{14} \cdot 5 \cdot 11)(3 \cdot 5^3 \cdot 11^3) \pmod{914387} \\ &= (2^9 \cdot 3 \cdot 5^5 \cdot 11^2)^2 \\ &= 580800000^2 \\ &\equiv 164255^2 \pmod{914387} \end{aligned}$$

Vi noterar nu också att $1869 \cdot 1909 \cdot 3387 \equiv 9835 \pmod{914387}$ *så vi beräknar därför*

$$SGD(914387, 9835 - 164255) = SGD(914387, 154420) = 1103.$$

Vi delar $n = 914387$ *med* 1103 *och har nu funnit faktoriseringen* $914387 = 1103 \cdot 829$.

Nu ska vi studera två faktoriseringsalgoritmer som uppfanns i modern tid, nämligen Pollards två algoritmer.

2.4.2 Pollards $(p-1)$-faktoriseringsalgoritm

Denna metod utvecklades av John Pollard 1974 ([7] sid. 214) och fungerar bra för sammansatta heltal med en primtalsfaktor p sådan att $(p-1)$ bara har små primtalsdelare.
Antag nu att vi har ett tal $n = pq$ och vill hitta primtalsfaktorerna p och q. Innan vi ger oss på en beskrivning av själva algoritmen ska vi gå igenom teorin för hur den är uppbyggd.
Antag nu först att vi på något sätt hittat ett positivt heltal L sådant att $(p-1) \mid L$ och $(q-1) \nmid L$, det vill säga att $L = i(p-1)$ och $L = j(q-1) + k$ där i, j, k är heltal och $k \neq 0$. Låt oss sedan slumpmässigt välja ett heltal a och beräkna a^L. Med hjälp av sats (1) får vi då

$$a^L = a^{i(p-1)} = (a^{p-1})^i \equiv 1^i \equiv 1 \pmod{p}$$

och

$$a^L = a^{j(q-1)+k} = a^k(a^{q-1})^j \equiv a^k \cdot 1^j \equiv a^k \pmod{q}.$$

På grund av att $k \neq 0$ är det högst otroligt att $a^k \equiv 1 \pmod{q}$, så alltså får vi för de flesta val av a att $p \mid (a^L - 1)$ och $q \nmid (a^L - 1)$. Detta betyder att $p = SGD(a^L - 1, n)$ och vi har hittat en av n:s primtalsfaktorer.
Men hur ska vi nu hitta talet L? Pollard observerade att om $(p-1)$ råkar vara en produkt av många små primtal så kommer $(p-1) \mid m!$ för något värde av m som inte är alltför stort, så idén är följande: för varje $m = 2, 3, 4 \ldots$ välj ett värde på a och beräkna $SGD(a^{m!} - 1, n)$. Om detta är $= 1$ så går vi vidare till nästa värde på m. Om detta istället är $= n$ så har vi misslyckats, men då kan vi istället prova med ett annat värde på a. Och till slut, om vi får att $1 < SGD(a^{m!} - 1, n) < n$ har vi hittat en icke-trivial faktor p av n och vi är då klara.
Innan vi går vidare med en beskrivning av själva algoritmen ska vi notera att vi faktiskt inte behöver beräkna $(a^{m!} - 1)$ exakt, vilket kan vara svårt redan om $a = 2$ och $m = 100$ (talet får då mer än 10^{157} siffror, [6] sid. 134). Eftersom vi endast är intresserade av $SGD(a^{m!} - 1, n)$ räcker det för vår del att beräkna $(a^{m!} - 1)$ mod n och sedan beräkna SGD med n. Vi behöver inte heller beräkna $m!$ för om vi nu redan beräknat $a^{m!}$ mod n ovan beräknas nästa värde som $a^{(m+1)!} \equiv (a^{m!})^{m+1} \pmod{n}$.
Faktoriseringsproceduren (som, i likhet med alla beräkningar i dessa sammanhang, utförs med hjälp av en dator) går nu till på följande sätt.

1. Låt $a = 2$ (eller något annat lämpligt värde).

2. Konstruera en loop såhär: låt $j = 2, 3, 4 \ldots$ upp till ett specifierat värde och sätt $b = a^j \pmod{n}$. Beräkna $d = SGD(b - 1, n)$.

3. Om $1 < d < n$ så har vi nått framgång, om inte så öka j och kör loopen i steg 2 igen.

Vi illustrerar nu algoritmen med ett numeriskt exempel.

Exempel 3. *Vi ska nu använda Pollards $(p-1)$-metod för att faktorisera $n = 13927189$. Vi börjar med $SGD(2^{9!} - 1, n)$ och beräknar nu successivt*

$$SGD(2^{9!}-1, n) = 1, \quad SGD(2^{10!}-1, n) = 1, \quad \ldots \quad SGD(2^{14!}-1, n) = 3823.$$

Detta ger oss nu en icke-trivial faktor $p = 3823$ av n. Denna faktor är ett primtal och vi får även $q = \frac{n}{p} = 3643$. Anledningen till att detta fungerade är att $(p-1)$ är en produkt av små primtal, närmare bestämt så gäller $(p-1) = 3822 = 2 \cdot 3 \cdot 7^2 \cdot 13$. q satisfierar $(q-1) = 3642 = 2 \cdot 3 \cdot 607$, vilket inte är en produkt av endast små primtal.

$(p-1)$-metoden är viktig att känna till för att man i RSA-systemet då kan undvika att välja talen n så att dess primtalsfaktorer p och q inte har egenskapen att $(p-1)$ och $(q-1)$ kan faktoriseras helt i små primtal.

2.4.3 Pollards rho-metod

Pollard utvecklade också en annan faktoriseringsmetod år 1975 ([7] sid 215) som även brukar benämnas Monte-Carlo faktoriseringsmetoden. Vi ska lite kort gå igenom denna algoritm.
Givet ett positivt sammansatt heltal n och ett tal p, hittills okänt, som en primtalsfaktor av n så ser utförandet ut såhär:

1. Välj ett polynom f med $deg(f) \geq 2$. Vanligtvis väljs $f(x) = x^2 + 1$ för enkelhetens skull.

2. Välj ett slumpmässigt genererat heltal $x = x_0$ (brukligt är att starta på $x_0 = 2$) och beräkna

$$x_1 = f(x_0), \quad x_2 = f(x_1) \quad \dots \quad x_{j+1} = f(x_j), \quad j = 0, 1, \dots B,$$

   där B bestäms i nästa steg.

3. Reducera alla resultat modulo n och beräkna $SGD(x_i - x_j, n)$ för $i \neq j$. Så snart som $SGD(x_B - x_j, n) \neq 1$ för något heltal B har vi funnit en icke-trivial faktor av n.

Resultatet i steg 3 ovan betyder att

$$x_B \not\equiv x_j \pmod{n}$$

men

$$x_B \equiv x_j \pmod{p}$$

för något naturligt tal $B > j \geq 1$. Då gäller att $SGD(x_B - x_j, n)$ är en icke-trivial delare till n. Om p är en faktor av n så kommer vi att få se denna repetition av tal mod$p$ (se [4] sid. 179) och detta betyder att $SGD(x_B - x_j, n)$ kommer att vara delbart med p eftersom x_B är en repetition av x_j. För denna metod gäller även att den relativt snabbt hittar små primtalsfaktorer av sammansatta tal, så är det så att ett tal har en liten primtalsfaktor (enligt [5] sid. 389 definieras en liten primtalsfaktor som en i storleken ≈ 1000000) så kommer Pollards rho-metod att hitta denna snabbare än den hittar en större faktor. Däremot är det svårt att faktiskt bevisa att denna algoritm fungerar så väl som den gör. För en beskrivning av varför metoden kallas just Pollards rho-metod, se [7] sid. 216.
Avslutningsvis tittar vi nu på ett numeriskt exempel där vi tillämpar Pollards rho-metod.

Exempel 4. *Om vi har* $n = 37351, \quad x_0 = 2$ *och* $f(x) = x^2 + 1$ *så får vi successivt*

$$x_1 = f(x_0) = 5, \quad x_2 = f(x_1) = 26, \quad x_3 = f(x_2) = 677, \quad x_4 = f(x_3) = 3146,$$

$$x_5 = f(x_4) = 36653 \quad och \quad x_6 = f(x_5) = 1642$$

där alla resultat är mod$n$. *Man får att alla* $SGD(x_i - x_j, n) = 1$ *för* $i \neq j$ *tills* $SGD(x_6 - x_0, n) = SGD(1640, 37351) = 41$. *Faktum är att* $37351 = 41 \cdot 911$.

3 ElGamal

Nu ska vi gå över till att studera det andra asymmetriska krypteringssystemet som denna uppsats handlar om, nämligen ElGamal. Basen för ElGamal ligger i en procedur som kallas Diffie-Hellmans nyckelutbyte (som alltså inte är ett krypteringssystem i sig själv). Säkerheten av Diffie-Hellmans nyckelutbyte bygger på det diskreta logaritmproblemet och vi börjar med att gå igenom en del definitioner och satser som vi behöver för att kunna definiera dels vad en diskret logaritm är, dels vad det diskreta logaritmproblemet är för något.

3.1 Nödvändiga definitioner och satser

Definition 1. *Låt a vara ett godtyckligt element i en ändlig grupp G. Ordningen av a i G är det minsta tal l för vilket $a^l = e$, där e är identitetselementet.*

Sats 3. *Lagranges sats.*
Låt p vara ett primtal. Då gäller att en kongruens av grad n i $\mathbb{F}_p$, kroppen av heltal modulo p, säg

$$a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \equiv 0 \pmod{p}, \qquad \text{där } a_n \not\equiv 0 \pmod{p} \quad (2)$$

inte kan ha mer än n lösningar.

Bevis. Om x_1 är en godtycklig lösning av kongruensen (2) så kan polynomet på vänstra sidan faktoriseras och en av faktorerna är $x - x_1$. Om x_1 satisfierar (2) har vi att

$$a_n x_1^n + a_{n-1}x_1^{n-1} + \cdots + a_1 x_1 + a_0 \equiv 0 \pmod{p}.$$

Om vi nu drar detta från (2) får vi ett uttryck bestående av termer av formen $a_k(x^k - x_1^k)$ där k är ett av talen $0, 1, \ldots n$. Varje sådan differens innehåller det linjära polynomet $x - x_1$ som en faktor. Detta betyder att vi kan skriva (2) som

$$(x - x_1)(b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_0) \equiv 0 \pmod{p},$$

där $b_{n-1}, \ldots b_0$ är heltal som beror på $a_n, \ldots a_0$ och x_1. Varje annan godtycklig lösning, säg till exempel x_2, av (2) måste (på grund av att p är ett primtal) satisfiera

$$b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \ldots b_0 \equiv 0 \pmod{p}$$

och dessutom ge upphov till en faktor $x - x_2$ av detta polynom. Detta betyder att vi nu har två linjära faktorer av vårt ursprungliga polynom. Vi fortsätter på samma sätt tills antingen vänsterledet av (2) är helt faktoriserat eller tills vi kommer till en olösbar kongruens. I det förra fallet har kongruensen (2) exakt n lösningar och i det senare fallet mindre än n lösningar. Detta bevisar satsen. □

Sats 4. *Satsen om existens av ett primitivt element i en ändlig kropp (eng. Primitive Root Theorem).*
Låt p vara ett primtal. Då existerar ett element $g \in \mathbb{F}_p$, där $\mathbb{F}_p$ är kroppen av heltal modulo p, vars potenser ger varje element av $\mathbb{F}_p$, det vill säga $\mathbb{F}_p = \{1, g, g^2, g^3 \dots g^{p-2}\}$. Element g med denna egenskap kallas primitiva rötter (eller generatorer) av $\mathbb{F}_p$ och har ordning $(p-1)$.

Bevis. Vi bevisar satsen genom att konstruera en primitiv rot. Låt $(p-1)$ kunna faktoriseras i primtalspotenser som $(p-1) = q_1^{a_1} q_2^{a_2} \cdots$. Om vi nu kan hitta ett tal x_1 vars ordning är $q_1^{a_1}$, och ett tal x_2 vars ordning är $q_2^{a_2}$, och så vidare, så kommer produkten av alla dessa tal ha ordning $(p-1)$ (se [4] sid. 50) och dessutom vara en primitiv rot. Nu måste vi alltså bevisa att om q^a är en av primtalspotenserna som $(p-1)$ består av, så finns det något tal vars ordning $\mathrm{mod} p$ är exakt q^a. Ett tal x vars ordning är q^a måste satisfiera kongruensen

$$x^{q^a} \equiv 1 \pmod{p}. \tag{3}$$

Men ett tal som satisfierar (3) måste inte nödvändigtvis ha ordning q^a, ordningen kan vara en godtycklig faktor av q^a. Det vill säga ordningen av x kan vara allt från $1, q, q^2 \dots$ upp till q^{a-1}. Dock, om ordningen inte är q^a så måste ordningen vara en faktor av q^{a-1}, och x kommer då att satisfiera kongruensen

$$x^{q^{a-1}} \equiv 1 \pmod{p}. \tag{4}$$

Vi letar nu alltså efter ett tal som satisfierar (3) men inte (4) och vi kan bevisa att det finns ett sådant tal genom att finna hur många lösningar dessa kongruenser har. Av sats (3) följer att (3) som mest har q^a lösningar och att (4) som mest har q^{a-1} lösningar. Vi visar nu att dessa kongruenser har exakt q^a och q^{a-1} lösningar. Då följer att det finns $q^a - q^{a-1}$ tal som satisfierar (3) men inte (4) och eftersom $q^a > q^{a-1}$ ger detta oss vad vi behöver.
Betrakta nu mer generellt kongruensen

$$(x^d - 1) \equiv 0 \pmod{p},$$

där d är en godtycklig faktor av $(p-1)$. Av sats (3) följer att denna kongruens som mest har d lösningar och vi måste nu visa att den har exakt d lösningar. Vi noterar nu att polynomet $x^d - 1$ är en faktor av polynomet $x^{p-1} - 1$, ty om vi sätter $y = x^d$ och $(p-1) = de$ så följer det att

$$x^{p-1} - 1 = y^e - 1 = (y-1)(y^{e-1} + y^{e-2} + \cdots + 1).$$

Eftersom $y - 1 = x^d - 1$ ger detta en likhet av formen

$$x^{p-1} - 1 = (x^d - 1) f(x),$$

där $f(x)$ är ett visst polynom i x av grad $(p-1-d)$. Nu gäller att kongruensen

$$x^{p-1} - 1 \equiv 0 \pmod{p}$$

har $(p-1)$ lösningar och alla dessa måste satisfiera antingen

$$x^d - 1 \equiv 0 \pmod{p}$$

eller

$$f(x) \equiv 0 \pmod{p}.$$

Den senare har som mest $(p-1-d)$ lösningar, enligt sats (3), så den förra måste ha åtminstone d lösningar, och därför har den exakt d lösningar. Sätter vi d till q^a eller q^{a-1} ger detta oss det önskade beviset av denna sats. □

Vi illusterar satsen med ett enkelt exempel.

Exempel 5. *Kroppen $\mathbb{F}_{11}$ har 2 som primitiv rot eftersom följande likheter gäller i $\mathbb{F}_{11}$:*

$$2^0 = 1, \quad 2^1 = 2, \quad 2^2 = 4, \quad 2^3 = 8, \quad 2^4 = 5, \quad 2^5 = 10, \quad 2^6 = 9,$$

$$2^7 = 7, \quad 2^8 = 3, \quad 2^9 = 6.$$

Det vill säga, alla 10 element $\neq 0$ i $\mathbb{F}_{11}$ har blivit genererade som en potens av 2.

Definition 2. *Låt p vara ett primtal och g en primitiv rot i kroppen av heltal modulo p, som vi här betecknar $\mathbb{F}_p$. För ett godtyckligt heltal $A \in \{1, 2 \ldots (p-1)\}$ (det vill säga A är ett element i $\mathbb{F}_p$ och $A \neq 0$) finns då en exponent $a \in \{0, 1 \ldots (p-2)\}$ sådan att $A \equiv g^a \pmod{p}$.*
a kallas den diskreta logaritmen av A i basen g.

Definition 3. *Att hitta det minsta icke-negativa heltalet x sådant att $g^x \equiv A \pmod{p}$ kallas det diskreta logaritmproblemet.*

Nu har vi vad vi behöver för att kunna beskriva Diffie-Hellmans nyckelutbyte och krypteringssystemet ElGamal.

3.2 Diffie-Hellmans nyckelutbyte

Vi ska nu beskriva hur proceduren för Diffie-Hellmans nyckelutbyte går till. Vi låter som vanligt Bob och Alice vara de personer som vill utbyta konfidentiell information och därför önskar kryptera sina meddelanden till varandra.

1. Bob och Alice kommer överens om ett stort primtal p och en primitiv rot $g \bmod p$, där $2 \leq g \leq (p-2)$ (se definition (2)). Dessa två tal kan göras offentliga.

2. Nu väljer Alice slumpmässigt ett heltal a, $0 \leq a \leq (p-2)$. Hon beräknar också $A \equiv g^a \pmod{p}$. På samma sätt väljer Bob slumpmässigt ett heltal b, $0 \leq b \leq (p-2)$ och beräknar $B \equiv g^b \pmod{p}$. a och b måste hållas hemliga av Bob och Alice.

3. Sedan skickar Alice A till Bob och Bob skickar B till Alice. Dessa tal blir självklart också offentliga eftersom vem som helst kan läsa okrypterad information. Den publika nyckeln är nu (g, p, A) för Alices del och (g, p, B) för Bobs del.

4. För att nu få fram den privata nyckeln beräknar Alice $B^a \bmod p \iff g^{ab} \bmod p$ och Bob beräknar $A^b \bmod p \iff g^{ab} \bmod p$. Den privata nyckeln för båda är nu $K \equiv g^{ab} \pmod{p}$.

Svårigheten för en eventuell fiende kan vi beskriva såhär: om vi tänker oss att fienden har fått reda på talen A, B, p och g så vet han eller hon automatiskt också värdena på g^a och g^b. Skulle nu denna fiende kunna lösa det diskreta logaritmproblemet betyder det att han eller hon kan hitta värdena på a och b och därmed beräkna den privata nyckeln g^{ab}. Det diskreta logaritmproblemet i Diffie-Hellmans nyckelutbyte är alltså att lösa $g^a \bmod p$ för a och $g^b \bmod p$ för b.
Att hitta $K \equiv g^{ab} \pmod{p}$ givet att man vet A, B, p och g kallas allmänt för Diffie-Hellman problemet. Om vi lyckas beräkna diskreta logaritmer mod$p$ kan vi också lösa Diffie-Hellman problemet. Detta problem är inte svårare att lösa än det diskreta logaritmproblemet. Det står klart att om vi hittar en lösning till det diskreta logaritmproblemet så kan vi beräkna den privata nyckeln K, med andra ord så kommer vi då att hitta en lösning till Diffie-Hellman problemet. Dock finns det inget svar på om man kan lösa det diskreta logaritmproblemet givet att man först löst Diffie-Hellman problemet.
Konsensus inom krypteringsvärlden är dock ([7], sid. 168 och sid. 182-183) att dessa problem är ekvivalenta och att man därför kan anta att säkerheten av Diffie-Hellmans nyckelutbyte (och därmed också säkerheten för krypteringssystemet ElGamal) bygger på det diskreta logaritmproblemet.
Nu går vi över till krypteringssystemet ElGamal och precis som när vi tittade på RSA börjar vi med nyckelgenereringen.

3.3 Generering av nycklar

Krypteringssystemet ElGamal utvecklades av Taher ElGamal år 1985 ([6], sid. 68). Systemets säkerhet bygger som redan nämnts på det diskreta logaritmproblemet och Diffie-Hellmans nyckelutbyte, båda vilka vi gick igenom ovan. För att generera den privata och den publika nyckeln gör vi på följande sätt: vi väljer först ett stort primtal p och en primitiv rot $g \bmod p$. Enligt [5], sid. 172, bör p vara av storleken $p > 10^{150}$. Sedan väljer vi slumpmässigt en

exponent a som uppfyller $0 \leq a \leq (p-2)$ och beräknar $A \equiv g^a \pmod{p}$.
Vår publika nyckel är nu (g, p, A) och vår privata nyckel är a. Notera likheterna i denna procedur med Diffie-Hellmans nyckelutbyte. Vi ska också kort nämna att talen g och p måste alla som vill kryptera meddelanden till varandra förstås vara överens om.

3.4 Kryptering och dekryptering med ElGamal

Nu är det dags att börja kryptera meddelanden. Säg att vi nu vill skicka ett krypterat meddelande till Bob, vars publika nyckel är (g, p, B), som han såklart beräknat på samma sätt som vi beräknade vår egen publika nyckel ovan. Bobs privata nyckel är b, vilket är ett tal som han genererat slumpmässigt och håller hemligt.
Säg nu att vårt klartextmeddelande är m (omvandlat till siffror) där $0 \leq m \leq (p-1)$. Detta ska nu krypteras. Vi letar därför upp Bobs publika nyckel (p, g, B), som vi antar finns arkiverad i någon offentlig databas, och beräknar vår kryptotext c på följande vis:

$$c \equiv B^a m \pmod{p} \iff c \equiv g^{ab} m \pmod{p}.$$

(Enligt ovan är ju $B \equiv g^b \pmod{p}$).
Den kompletta kryptotexten, vilket är en bruklig benämning, som vi skickar till Bob är (A, c) där $A \equiv g^a \pmod{p}$ är en del av vår publika nyckel. Bob tar emot (A, c) och vill nu dekryptera kryptomeddelandet c. Han använder för detta ändamål sin privata nyckel b och beräknar

$$(A)^{-b} \equiv (g^a)^{-b} \pmod{p}.$$

Sedan dekrypterar han c genom att beräkna

$$(g^a)^{-b} \underbrace{m g^{ab}}_{\text{c}} \bmod p \iff g^{-ab} g^{ab} m \bmod p \iff m \bmod p.$$

Bob har nu alltså fått fram meddelandet m.

Innan vi går vidare med ett konkret exempel ska vi kort titta på varför vi bara kan använda vår krypteringsexponent a en gång. Vi tänker oss därför att vi använder samma a för att kryptera två olika klartextmeddelanden, m_1 och m_2 och att vår fiende har fått reda på m_1. Han eller hon kan nu få fram m_2 och vi ska beskriva hur. De två kryptotexterna är

$$c_1 = (g^a, m_1 g^{ab})$$

och

$$c_2 = (g^a, m_2 g^{ab}).$$

Vår fiende kan nu beräkna

$$m_2 g^{ab} m_1^{-1} m_1 g^{-ab} = m_2$$

och kan därmed läsa vårt andra meddelande som vi skickar med samma krypteringsexponent a.

3.5 Ett krypteringsexempel

Låt oss nu med ett numeriskt exempel se hur Alice och Bob använder sig av ElGamal för att kryptera meddelanden till varandra. Talen i exemplet är naturligtvis alldeles för små för att ge förfarandet någon verklig säkerhet, men större tal ger en onödig otymplighet då vi endast är ute efter att illustrera själva tillvägagångssättet.
Vi låter dem använda $p = 467$ och $g = 2$. Alice väljer nu $a = 153$ som sin privata nyckel och beräknar därefter A som en del av sin publika nyckel:

$$A \equiv g^a \equiv 2^{153} \equiv 224 \pmod{467}.$$

Bob vill nu skicka Alice meddelandet $m = 331$. Han väljer sin privata nyckel $b = 197$ och beräknar

$$B \equiv g^b \equiv 2^{197} \equiv 87 \pmod{467}.$$

Han krypterar sedan m till c genom

$$c \equiv mA^b \pmod{p} \iff c \equiv 331 \cdot 224^{197} \equiv 57 \pmod{467}.$$

Han skickar sedan paret $(B, c) = (87, 57)$ till Alice. Detta är alltså den kompletta kryptotexten.
Alice använder nu sin privata nyckel $a = 153$ och beräknar

$$(B)^{-a} \bmod p \iff (g^b)^{-a} \bmod p \iff (87)^{-153} \equiv 14 \pmod{467}.$$

Hon dekrypterar sedan c genom följande beräkning:

$$(B)^{-a} c \bmod p \iff 14 \cdot 57 \equiv 331 \pmod{467}.$$

Alice har nu fått fram klartexten $m = 331$.

3.6 Metoder för att lösa det diskreta logaritmproblemet

Vi ska nu gå igenom några olika metoder och algoritmer som man kan använda då man vill lösa det diskreta logaritmproblemet. Vi påminner oss först om att det diskreta logaritmproblemet går ut på att hitta det minsta icke-negativa heltalet x sådant att $g^x \equiv A \pmod{p}$. Vi skriver $x = log_g A$ där mod$p$ är underförstått från sammanhanget. x kallas ibland index av A i bas g modulo p för att klargöra att det inte rör sig om någon vanlig logaritm. Index är faktiskt till och med en äldre benämning på en diskret logaritm (se [6] sid. 63 och 163).

3.6.1 Prövning (eng. trial and error)

Denna metod för att angripa det diskreta logaritm-problemet går helt enkelt till så att man beräknar $g^x \bmod p$ för varje $x = 1, 2, 3 \ldots$ och jämför med värdet på A.
Om g som element i $\mathbb{F}_p$ har ordning n (det vill säga att $g^n = e$ där e är identitetselementet och att ingen mindre positiv potens av g är lika med e) så är denna algoritm garanterad att hitta en lösning genom att som mest utföra räkneoperationen ovan n gånger. Om n är stort, till exempel $n > 2^{80}$ ([6] sid. 75), så är denna metod inte praktisk att utföra med den datorkraft som finns tillgänglig idag. Att bryta ElGamal med hjälp av prövning blir alltså väldigt otympligt eftersom g har en väldigt hög ordning. g är ju som bekant en primitiv rot i $\mathbb{F}_p$ och det betyder att den har ordning $(p-1)$, vilket tillsammans med vetskapen om att p är ett tal med $p > 10^{150}$ ger att g har en mycket hög ordning.

3.6.2 Shanks Baby-step Giant-step algoritm

Denna algoritm för att lösa det diskreta logaritmproblemet utvecklades av Daniel Shanks. Idén är rätt enkel, vi skapar helt enkelt två listor och letar efter ett element som finns med i båda listorna. Prövningsmetoden som vi beskrev ovan är också ett exempel på denna metod. I det fallet består den ena listan av talen $g^1, g^2, \ldots$ och den andra av talet A. Vi beskriver Shanks algoritm i form av en sats.

Sats 5. *Låt G vara en grupp och låt $g \in G$ vara ett element av ordning $N \geq 2$. Följande algoritm löser då det diskreta logaritmproblemet $g^x = h$.*

1. *Låt $n = 1 + \lfloor \sqrt{N} \rfloor$, det vill säga $n > \sqrt{N}$.*
2. *Skapa två listor.*
   Lista 1: $1, g, g^2 \ldots g^n$.
   Lista 2: $h, hg^{-n}, hg^{-2n} \ldots hg^{-n^2}$.
3. *Hitta ett element som finns i båda listorna, säg till exempel att $g^i = hg^{-jn}$.*
4. *Då gäller att $x = i + jn$ är en lösning till $g^x = h$.*

Bevis. För att visa att algoritmen fungerar måste vi visa att listorna 1 och 2 som skapas alltid innehåller ett gemensamt element. Vi låter nu x vara den hittills okända lösningen till $g^x = h$ och skriver $x = nq + r$ där $0 \leq r < n$. r är alltså resten och q kvoten när vi delar x med n.
Vi vet nu att $1 \leq x < N$ så vi får

$$q = \frac{x - r}{n} < \frac{N}{n} < n,$$

eftersom vi har att $n > \sqrt{N}$. Vi kan nu alltså skriva om ekvationen $g^x = h$ som

$$g^x = h \iff g^{qn+r} = h \iff g^r = hg^{-qn}$$

där $0 \leq r < n$ och $0 \leq q < n$. Om vi nu tittar på våra två listor ser vi att g^r finns i lista 1 och att hg^{-qn} finns i lista 2. Detta visar att dessa listor alltid har ett gemensamt element. □

Namnet av algoritmen kommer av detta: när vi skapar lista 2 börjar vi med att beräkna $u = g^{-n}$ och får sedan fram listan genom beräkningarna $h, hu, hu^2 \ldots hu^n$. Multiplikationen med g kallas ett baby step och multiplikationen med $u = g^{-n}$ är ett giant step enligt [6] sid. 80. För att förstå lite bättre hur algoritmen fungerar ska vi tillämpa den på ett numeriskt exempel där vi försöker lösa ett diskret logaritmproblem.

Exempel 6. *Säg att vi vill lösa ekvationen* $g^x = h$ *i* $\mathbb{F}_p$ *med* $g = 9704, \quad h = 13896$ *och* $p = 17389$. $g = 9704$ *har ordning* 1242 *i* $\mathbb{F}_{17389}$. *Sätt nu*

$$n = \lfloor \sqrt{N} \rfloor + 1 = \lfloor \sqrt{1242} \rfloor + 1 = 36$$

och

$$u = g^{-n} = 9704^{-36} = 2494.$$

Vi skapar nu våra två listor (vi tar här med bara några få värden på k*):*

k	g^k	hu^k
1	9704	347
2	6181	13357
7	14567	6259
32	7583	14567

Vi hittar nu vårt gemensamma element som

$$9704^7 = 14567 = 13896 \cdot 2494^{32}$$

i $\mathbb{F}_{17389}$. *Om vi nu använder* $2494 = 9704^{-36}$ *så kan vi beräkna*

$$13896 = 9704^7 \cdot 2494^{-32} = 9704^7 \cdot (9704^{36})^{32} = 9704^{1159}$$

i $\mathbb{F}_{17389}$, *det vill säga* $x = 1159$ *löser det diskreta logaritm-problemet* $9704^x = 13896$ *i* $\mathbb{F}_{17389}$.

3.6.3 Pohlig-Hellmans algoritm

För att kunna använda denna algoritm behöver vi först gå igenom en sats som heter Kinesiska restsatsen (eng. The Chinese remainder theorem). Denna sats beskriver lösningarna till ett system av simultana linjära kongruenser. Det enklaste exemplet består av två ekvationer, till exempel

$$x \equiv a \pmod{m} \quad \text{och} \quad x \equiv b \pmod{n},$$

där $SGD(m,n) = 1$. I detta fall säger den Kinesiska restsatsen att det existerar en unik lösning modulo mn.

Sats 6. *Kinesiska restsaten.*
Låt $m_1, m_2, \ldots m_n$ vara en samling av parvis relativt prima heltal. Detta betyder att $SGD(m_i, m_j) = 1$ för alla $i \neq j$. Låt $a_1, a_2, \ldots a_n$ vara godtyckliga heltal. Då har systemet av simultana kongruenser

$$x \equiv a_1 \pmod{m_1}, \quad x \equiv a_2 \pmod{m_2}, \quad \ldots \quad x \equiv a_n \pmod{m_n}$$

en lösning $x = c$. Vidare, om $x = c$ och $x = b$ båda är lösningar så gäller att

$$c \equiv b \pmod{m_1 m_2 \cdots m_n}.$$

Bevis. Låt

$$m = \prod_{i=1}^{n} m_i, \qquad M_i = \frac{m}{m_i}, \qquad 1 \leq i \leq n.$$

Vi använder Euklides algoritm för att beräkna talen $y_i \in \mathbb{Z}, \quad 1 \leq i \leq n$ med

$$y_i M_i \equiv 1 \pmod{m_i}, \qquad 1 \leq i \leq n. \tag{5}$$

Sedan sätter vi

$$x \equiv (\sum_{i=1}^{n} a_i y_i M_i) \pmod{m} \tag{6}$$

och visar att x är en lösning av de simultana kongruenserna ovan. Av (5) får vi

$$a_i y_i M_i \equiv a_i \pmod{m_i}, \qquad 1 \leq i \leq n \tag{7}$$

och för $i \neq j$ har vi att heltalet m_i är en delare av M_j så därför har vi

$$a_j y_j M_j \equiv 0 \pmod{m_i}, \qquad 1 \leq i, j \leq n, \qquad i \neq j. \tag{8}$$

Av (6), (7) och (8) får vi

$$x \equiv a_i y_i M_i + \sum_{j=1, j\neq i}^{n} a_j y_j M_j \equiv a_i \pmod{m_i}, \qquad 1 \leq i \leq n,$$

det vill säga, x löser de simultana kongruenserna ovan.
Nu återstår det för oss att visa att denna lösning är unik. Låt därför $x = c$ och $x = b$ vara två lösningar till de simultana kongruensekvationerna. Då gäller att

$$c \equiv b \pmod{m_i}, \qquad 1 \leq i \leq n$$

och på grund av att talen m_i är parvis relativt prima följer det att

$$c \equiv b \pmod{m}.$$

□

Nu har vi allt vi behöver för att kunna formulera en beskrivning av Pohlig-Hellmans algoritm.

Sats 7. *Pohlig-Hellmans algoritm.*
Låt G vara en grupp och $g \in G$ ett element av ordning N. Antag att vi har en algoritm för att lösa det diskreta logaritmproblemet i G för ett godtyckligt element vars ordning är en potens av ett primtal, och antag vidare att N kan faktoriseras som en produkt av primtal på följande sätt: $N = q_1^{e_1} q_2^{e_2} \cdots q_t^{e_t}$. Då gäller att man kan lösa det diskreta logaritmproblemet $g^x = h$ med följande procedur:

1. *För varje $1 \leq i \leq t$, låt*
$$g_i = g^{N/q_i^{e_i}}$$
och
$$h_i = h^{N/q_i^{e_i}}.$$
Notera att g_i har en primtalspotensordning $q_i^{e_i}$ så använd algoritmen vi antar att vi har tillgång till för att lösa det diskreta logaritmproblemet
$$g_i^y = h_i. \tag{9}$$
Låt $y = y_i$ vara en lösning till (9). (Se [6] sid. 89 för en beskrivning av en sådan algoritm.)

2. *Använd den Kinesiska restsatsen (sats (6)) för att lösa*
$$x \equiv y_1 \pmod{q_1^{e_1}}, \quad x \equiv y_2 \pmod{q_2^{e_2}}, \quad \ldots \quad x \equiv y_t \pmod{q_t^{e_t}}. \tag{10}$$

Bevis. Vi ska nu visa att dessa två steg ger en lösning till $g^x = h$. Låt x vara en lösning till (10). För varje i kan vi då skriva

$$x = y_i + q_i^{e_i} z_i \quad \text{för något } z_i. \tag{11}$$

Vi beräknar nu

$$\begin{aligned}
(g^x)^{N/q_i^{e_i}} &= (g^{y_i + q_i^{e_i} z_i})^{N/q_i^{e_i}} \quad \text{från (11)}, \\
&= (g^{N/q_i^{e_i}})^{y_i} g^{N z_i} \\
&= (g^{N/q_i^{e_i}})^{y_i} \quad \text{på grund av att } g^N \text{ är identitetselementet}, \\
&= g_i^{y_i} \quad \text{av definitionen av } g_i, \\
&= h_i \quad \text{av (9)}, \\
&= h^{N/q_i^{e_i}} \quad \text{av definitionen av } h_i.
\end{aligned}$$

I termer av diskreta logaritmer i basen g kan vi nu skriva om detta som

$$\frac{N}{q_i^{e_i}} \cdot x \equiv \frac{N}{q_i^{e_i}} \cdot log_g(h) \pmod{N}. \tag{12}$$

Vi påminner oss om att diskreta logaritmer i basen g bara är definierade modulo N eftersom g^N är identitetselementet. Nu noterar vi att talen

$$\frac{N}{q_1^{e_1}}, \quad \frac{N}{q_2^{e_2}}, \quad \cdots \quad \frac{N}{q_t^{e_t}}$$

inte har någon icke-trivial gemensam faktor, det vill säga för dem gäller att deras största gemensamma delare är $= 1$. Genom upprepad användning av Euklides algoritm kan vi därför hitta heltal $c_1, c_2, \ldots c_t$ sådana att

$$\frac{N}{q_1^{e_1}} \cdot c_1 + \frac{N}{q_2^{e_2}} \cdot c_2 + \cdots + \frac{N}{q_t^{e_t}} \cdot c_t = 1. \tag{13}$$

Om vi nu multiplicerar båda sidor av (12) med c_i och summerar över $i = 1, 2, \ldots t$ får vi

$$\sum_{i=1}^{t} \frac{N}{q_i^{e_i}} \cdot c_i \cdot x \equiv \sum_{i=1}^{t} \frac{N}{q_i^{e_i}} \cdot c_i \cdot log_g(h) \pmod{N}$$

och av (13) följer nu att

$$x \equiv log_g(h) \pmod{N}.$$

Detta visar att x satisfierar $g^x = h$. □

Som en avslutning ska vi nu se hur algoritmen används genom att studera ett numeriskt exempel.

Exempel 7. *Låt oss anta att vi nu vill lösa det diskreta logaritmproblemet* $23^x = 9689$ *i* $\mathbb{F}_{11251}$. *Basen* 23 *är en primitiv rot i* $\mathbb{F}_{11251}$, *det vill säga, den har ordning* 11250 *och vi har också att* $11250 = 2 \cdot 3^2 \cdot 5^4$. *Med samma notation som i satsen sätter vi*
$p = 11251$, $g = 23$, $h = 9689$ *och* $N = (p-1) = 2 \cdot 3^2 \cdot 5^4$.
Första steget består nu i att lösa tre diskreta logaritmproblem med hjälp av den speciella algoritmen för det diskreta logaritmproblemet för element vars ordning är en potens av ett primtal. (Se [6] sid. 89). Vi illusterar lösningen i nedanstående tabell.

q	e	$g^{(p-1)/q^e}$	$h^{(p-1)/q^e}$	*lös* $(g^{(p-1)/q^e})^x = h^{(p-1)/q^e}$ *för* x
2	1	11250	11250	1
3	2	5029	10724	4
5	4	5448	6909	511

Andra steget består nu i att använda den Kinesiska restsatsen (se sats (6)*) för att lösa de simultana kongruenserna*

$$x \equiv 1 \pmod{2}, \quad x \equiv 4 \pmod{3^2}, \quad x \equiv 511 \pmod{5^4}.$$

Den minsta lösningen är $x = 4261$. *Som kontroll för lösningens riktighet beräknar vi*

$$23^{4261} = 9689 \ i \ \mathbb{F}_{11251}.$$

Detta betyder att vi hittat en lösning till det diskreta logaritmproblem vi ville lösa med hjälp av Pohlig-Hellmans algoritm.

4 Primtalstest och primtalsgenerering

I både ElGamal och RSA som vi studerat i denna uppsats behöver vi tillgång till väldigt stora primtal för att kunna kryptera meddelanden. För att kunna använda dessa krypteringssystem måste vi alltså kunna skilja på primtal och sammansatta tal. Hittar vi en metod för detta kan vi generera slumptal av rätt storlek tills vi hittar ett som vi vet är ett primtal.

4.1 Eratosthenes såll och prövning med division

Den (antagligen) första kända metoden för att hitta primtal kallas Eratosthenes såll efter den grekiske vetenskapsmannen Eratosthenes, 276 f.Kr - 194 f.Kr. Vi illustrerar metoden med ett exempel.

Exempel 8. *Antag att vi vill hitta alla primtal* ≤ 20.

1. *Börja med att skriva ned alla naturliga tal* n *med* $1 < n \leq 20$*:* $2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20$. *Stryk sedan alla tal större än* 2 *som är multiplar av* 2 *(*2 *är ju det minsta primtalet):* $2, 3, \not{4}, 5, \not{6}, 7, \not{8}, 9, \not{10}, 11, \not{12}, 13, \not{14}, 15, \not{16}, 17, \not{18}, 19, \not{20}$.

2. *Stryk sedan alla tal större än* 3 *som är multiplar av* 3 *(*3 *är ju nästa primtal):* $2, 3, 5, 7, \not{9}, 11, 13, \not{15}, 17, 19$. *Vad vi nu har kvar är mängden av alla primtal* ≤ 20.

Eftersom vi numera vet (vilket dock inte Eratosthenes gjorde, [7] sid. 6) att om n är ett positivt sammansatt heltal så har n en primtalsdelare p som är $\leq \sqrt{n}$, så räcker det i vårt exempel att stryka tal $\sqrt{20} < 5$. Detta såll hjälper oss alltså att hitta tal som säkert är primtal, men vi vill ju också kunna testa om ett tal är ett primtal eller inte. Som vi redan förstått blir Eratosthenes såll ganska så svårt att använda då vi vill ha primtal att använda i ElGamal och RSA, där primtalen ju som vi fått reda på ovan helst ska vara av storlek $p > 10^{150}$ respektive $p \approx 10^{80}$.
Man kan utveckla sållet till ett så kallat primtalstest, det är en algoritm som avgör huruvida ett givet positivt heltal n är ett primtal. En algoritm som bygger på Eratosthenes såll (eller någon annan liknande metod att få fram primtal som inte är alltför stora) kallas prövning med division (eng. trial division) och används på samma sätt som dess namne inom primtalsfaktoriseringen vi gick igenom i avsnittet om RSA. Denna metod går till så att vi testar för alla primtal p som är $\leq \sqrt{n}$ om de delar n. Om vi hittar en primtalsdelare så är n sammansatt, annars ett primtal. Primtalen $p \leq \sqrt{n}$ kan fås antingen med Eratosthenes såll eller med någon slags förutbestämd primtalstabell.
Detta är den enklaste metoden för att testa om n är ett primtal, en så kallad direkt metod. Vi inser såklart att metoden, trots datorhjälp, kommer att bli obrukbar för att testa tal av den storlek som krävs i RSA eller ElGamal. För att kunna använda dessa krypteringssystem krävs därför bättre metoder.

4.2 Olika sorters primtalstest

Primtalstest kan vara både sannolikhetsbaserade och deterministiska. De primtalstest som är av störst betydelse för oss är de probabilistiska eftersom de är snabbare när det gäller stora tal. Dessa test bevisar att ett positivt heltal är ett primtal med hög sannolikhet. Som [5], sid. 264, uttrycker det: testen kan bevisa att ett tal är sammansatt med säkerhet men bevisar att ett tal är ett primtal bara med en viss sannolikhet. [5] kallar därför dessa test för test om ett tal är sammansatt. Testen går till så att de sållar bort sammansatta tal, men ibland kan sammansatta tal felaktigt utpekas som primtal. Slutsatsen av ett sådant test kommer att bli definitivt inte primtal eller kanske/troligen primtal, beroende på sannolikheten för att få rätt svar. Det finns också deterministiska primtalstest som givet ett heltal n verifierar en hypotes av en sats vars slutsats är att heltalet n är ett primtal. Vi kan alltså använda sådana metoder för att säkert bevisa att ett tal är ett primtal. I regel är dock dessa metoder långsammare än de probabilistiska metoderna och i praktiska tillämpningar är de sannolikhetsbaserade algoritmerna ofta fullt tillräckliga då sannolikheten för ett felaktigt svar kan göras väldigt låg. Vi ska nu gå igenom två probabilistiska metoder; Fermats primtalstest och Miller-Rabins test.

4.2.1 Fermats primtalstest

Denna algoritm är baserad på Fermats lilla sats (se sats (1)) som används för att bestämma om ett positivt heltal n är sammansatt. Metoden tillämpas på följande sätt: vi väljer ett positivt heltal a som satisfierar $1 < a < (n-1)$ och beräknar

$$y \equiv a^{n-1} \pmod{n}.$$

Om $y \neq 1$ är n sammansatt enligt sats (1). Om $y = 1$ kan vi varken dra slutsatsen att n är sammansatt eller slutsatsen att n är ett primtal. Tal a med egenskaperna att $1 < a < (n-1)$ och $a^{n-1} \not\equiv 1 \pmod{n}$ kallas ett Fermat-vittne till att n är ett sammansatt tal. Vi illusterar detta med ett exempel.

Exempel 9. *Betrakta* $n = 341 = 11 \cdot 31$. *Vi får*

$$2^{341-1} = 2^{340} \equiv 1 \pmod{341}$$

trots att vi vet att n *är ett sammansatt tal. Så Fermats test bevisar ingenting i detta fall. Å andra sidan får vi*

$$3^{341-1} = 3^{340} \equiv 56 \pmod{341},$$

så i detta fall ger Fermats test resultatet att 341 *är ett sammansatt tal.*

Fermats primtalstest kan alltså bevisa att ett positivt heltal n är sammansatt (ty om $a^{n-1} \not\equiv 1 \pmod{n}$ är n med full säkerhet inte ett primtal), men vi kan inte använda denna algoritm för att bevisa att ett tal är ett primtal. Dock, om vi för många val av a inte får bevis på att n är sammansatt så är det troligt ([5] sid 257 och [3] sid. 125) att n är ett primtal. Men det finns undantag och dessa kallas Carmichael-tal. De har egenskapen att de är sammansatta tal som inte kan bevisas att vara sådana med hjälp av Fermats primtalstest för godtycklig bas a. Det vill säga, n är ett Carmichael-tal om $a^{n-1} \equiv 1 \pmod{n}$ för alla heltal a med $SGD(a, n) = 1$.
På grund av existensen av dessa tal är Fermats algoritm inte optimal för praktisk användning utan måste kompletteras med någon bättre metod, till exempel Miller-Rabins test som vi beskriver härnäst.

4.2.2 Miller-Rabins test

Testet utvecklades av Gary Miller och Michael Rabin år 1976. Denna algoritm kan, till skillnad från Fermats primtalstest, bevisa att ett godtyckligt positivt heltal är sammansatt. Med andra ord finns det inte något analogt till Carmichael-talen för detta test och dessutom har det egenskapen att varje sammansatt tal har ett stort antal vittnen till att n är ett sammansatt tal. Först går vi igenom en proposition för att nedan kunna formulera proceduren för Miller-Rabins test.

Proposition 1. *Låt p vara ett udda primtal och skriv*

$$(p-1) = 2^k q \quad \text{där } q \text{ är udda.}$$

Låt a vara ett godtyckligt tal som inte är delbart med p. Då gäller att ett av följande två villkor är sanna:

1. $a^q \equiv 1 \pmod{p}$.
2. *Ett av talen* $a^q, a^{2q}, a^{4q}, \ldots a^{2^{k-1}q}$ *är* $\equiv -1 \pmod{p}$.

Bevis. Fermats lilla sats (se sats (1)) säger oss att $a^{p-1} \equiv 1 \pmod{p}$ och detta betyder att när vi betraktar talen

$$a^q, a^{2q}, a^{4q}, \ldots a^{2^{k-1}q}, a^{2^k q}$$

så vet vi att för det sista talet i listan gäller $a^{2^k q} = a^{p-1} \equiv 1 \pmod{p}$. Vidare är varje tal kvadraten av det närmast föregående talet och därför gäller ett av följande:

1. Det första talet i listan är $\equiv 1 \pmod{p}$.
2. Något av talen i listan är $\not\equiv 1 \pmod{p}$ men när det kvadreras så gäller att det är $\equiv 1 \pmod{p}$. Men det enda talet som satisfierar både

   $$b \not\equiv 1 \pmod{p} \quad \text{och} \quad b^2 \equiv 1 \pmod{p}$$

   är -1, så ett av talen i listan är $\equiv -1 \pmod{p}$.

Detta bevisar propositionen. □

Vi ska innan vi fortsätter göra följande definition:

Definition 4. *Låt n vara ett udda tal och skriv $(n-1) = 2^k q$ där q är udda. Ett heltal a som satisfierar $SGD(a, n) = 1$ kallas ett Miller-Rabin vittne för sammansattheten av n om båda av följande två villkor är sanna:*

1. $a^q \not\equiv 1 \pmod{n}$.
2. $a^{2^i q} \not\equiv -1 \pmod{n}$ *för alla* $i = 0, 1, 2, \ldots k-1$.

Det följer från vår proposition ovan att om det existerar ett a som är ett Miller-Rabin vittne till n så är n definitivt ett sammansatt tal. Allt detta leder nu till att vi kan beskriva Miller-Rabins test för sammansatta tal. Vi är alltså intresserade av att ta reda på om heltalet n är sammansatt och vi väljer heltalet a som ett potentiellt Miller-Rabin vittne till detta.

1. Om n är jämnt eller $1 < SGD(a, n) < n$ är n sammansatt och vi kan stanna i detta steg.
2. Skriv $(n-1) = 2^k q$ där q är udda.
3. Sätt $b = a^q \pmod{n}$.
4. Om $b \equiv 1 \pmod{n}$ har testet misslyckats, det vill säga, a är inte ett Miller-Rabin vittne till n. Välj ett annat värde på a och börja från början. Om $b \not\equiv 1 \pmod{n}$ så fortsätt processen.
5. Konstruera en loop på följande sätt: för $i = 1, 2, \ldots k-1$
   - Om $b^i \equiv -1 \pmod{n}$ för $i = 1$ har testet misslyckats, det vill säga, a är inte ett Miller-Rabin vittne. Välj ett annat värde på a och börja från början. Om $b^i \not\equiv -1 \pmod{n}$ för $i = 1$ så fortsätt processen.
   - Sätt $c = b^{2i} \pmod{n}$.
6. Om $c \equiv 1 \pmod{n}$ för $i = 1$ så är a ett Miller-Rabin vittne till att n är sammansatt. Om $c \equiv -1 \pmod{n}$ så har testet misslyckats, a är inte ett Miller-Rabin vittne till n och vi får välja ett nytt värde på a och börja om från början. Om ingen av de föregående kongruenserna gäller så öka i och kör loopen i steg 5 igen.
7. Till slut fås resultatet att n är ett sammansatt tal eller troligen ett primtal.

Enligt [6] sid. 127 har man att om n är ett udda sammansatt tal så är åtminstone 75% av talen mellan 1 och $(n-1)$ Miller-Rabin vittnen till n. Detta gör att detta test är bättre än Fermats primtalstest eftersom dess tillförlitlighet kan justeras till accepterad nivå genom att välja rätt uppsättning tal a som vittnen till att n är ett sammansatt tal. Vi beskriver detta genom följande resonemang:
Säg att Bob vill hitta stora primtal för att kunna kryptera meddelanden till Alice. Vi antar nu att han genererar ett stort tal n som är ett potentiellt primtal och kör Miller-Rabins test på n för 10 olika värden på a. Om ett godtyckligt a-värde är ett Miller-Rabin vittne till n så vet Bob att n är sammansatt. Men låt oss nu istället anta att inget av Bobs utvalda a-värden är ett Miller-Rabin vittne till n. Enligt [6] vet vi då att om n vore sammansatt så har Bob varje gång han testar ett värde på a 75% chans att det är ett vittne. Eftersom han nu på 10 försök inte hittade något vittne är det därför rimligt att dra slutsatsen att sannolikheten att n är sammansatt är som mest $(25\%)^{10}$, vilket approximativt är 10^{-6}. Om Bob nu inte tycker att detta är tillräckligt kan han använda 100 värden på a istället och om inget av dessa bevisar att n är sammansatt så är sannolikheten att n faktiskt är sammansatt mindre än $(25\%)^{100} \approx 10^{-60}$. Sannolikheten att vi med Miller-Rabins test faktiskt får tag på ett tal som är ett primtal kan därför som synes göras tillräckligt stor för vårt syfte.
Vi illustrerar nu Miller-Rabins test med ett exempel.

Exempel 10. *Säg att vi nu vill testa om talet $n = 172947529$ är ett primtal eller inte. Vi följer proceduren som vi beskrev i algoritmen ovan.*
Vi beräknar

$$(n-1) = 2^k q \iff 172947528 = 2^3 \cdot 21618441.$$

Vi väljer nu $a = 17$ och får då

$$17^{21618441} \equiv 1 \pmod{172947529}.$$

Detta visar att $a = 17$ inte är ett Miller-Rabin vittne till vårt n.
Vi provar nu istället med $a = 3$ och får

$$3^{21618441} \equiv -1 \pmod{172947529}.$$

Så $a = 3$ misslyckas också med att vara ett Miller-Rabin vittne.
Vi testar nu med $a = 23$ och får

$$23^{21618441} \equiv 40063806 \pmod{172947529},$$

$$23^{2 \cdot 21618441} \equiv 2257065 \pmod{172947529},$$

$$23^{4 \cdot 21618441} \equiv 1 \pmod{172947529}.$$

Det vill säga $a = 23$ är ett Miller-Rabin vittne till n och då vet vi att n är ett sammansatt tal.

5 Referenser

[1] Biggs, Norman L., Discrete Mathematics, Oxford University Press 2002

[2] Bogvad, Rikard, Kompendium i Algebra 1, Stockholms Universitet

[3] Buchmann, Johannes A., Introduction to Cryptography, Springer 2001

[4] Davenport, James H., The Higher Arithmetic, Cambridge University Press 2008

[5] Garrett, Paul, Making, Breaking Codes, Prentice-Hall 2001

[6] Hoffstein, Jeffrey., Pipher, Jill., Silverman, Joseph H., An Introduction to Mathematical Cryptography, Springer 2008

[7] Mollin, Richard A., An Introduction to Cryptography, Chapman&Hall/CRC, Taylor&Francis Group 2007

[8] RSA Laboratories, Frequently Asked Questions About Today's Cryptography, version 4.1 Maj 2000 `http://www.rsa.com/rsalabs/faq/files/rsalabs_faq41.pdf`

[9] Singh, Simon, Kodboken, Nordstedts Förlag 1999