Kompleksiteten til noen kryptologisk viktige algoritmer

Oversendes UB dato 31.10.1990
Kan/~~kan ikke~~ utlånes.

Eksamensinspektør

Universitetet i Tromsø

Hovedfagsoppgave Matematikk

Tore Brattli

Tromsø 9.3-1990

Veileder: Ben Johnsen

Kompleksiteten til noen kryptologisk viktige algoritmer

Universitetet i Tromsø

Hovedfagsoppgave
Matematikk

Tore Brattli

Tromsø 9.3-1990

Veileder: Ben Johnsen

Hovedfagsoppgave i Matematikk
Universitetet i Tromsø
Institutt for Matematiske Realfag

Av Tore Brattli

Oppgavens tittel:
Kompleksiteten til noen kryptologisk viktige algoritmer

Veileder: Ben Johnsen, UiTø

Innlevert 9. Mars 1990

© Tore Brattli 1990
Denne oppgaven er skrevet på en Macintosh datamaskin. Som tekstbehandlingsprogram har jeg brukt WriteNow 2.0. Til å skrive matematikk har Expressionist 2.03 vært til stor hjelp. Ellers er SuperPaint 2.0, MacDraw II, Matematica, DeltaGraph, Lightspeed C 3.0, Freehand 2.0, MORE og Pagemaker 3.5 vært med å gjøre oppgaven til det den har blitt.
Som hovedskrifttype er Times 12 brukt i tillegg til Symbol (Σψμβολ), Helvetica og noen få innslag av *Zapf Chancery*.

FORORD "Kompleksiteten til noen kryptologisk viktige algoritmer" har som mål å greie ut om algoritmer som har stor betydning for kryptologien. Jeg vil prøve å sammeligne den teoretiske med den praktiske kompleksiteten til en del av algoritmene som omhandles i oppgaven. Dvs. prøve å avsløre den skjulte konstanten bak O-notasjonen, slik at algoritmene kan sammenlignes på et reellt grunnlag.

I tillegg kan det være interessant å se litt på sammenhengen mellom sikkerhet, størrelsen på tall, asymptotisk og praktisk kompleksitet. Spesielt algoritmene som inngår i kryptering / dekryptering vil jeg forsøke å analysere i dybden. Algoritmer som er av mindre betydning er ikke like nøye analysert. Noen av algoritmene er kun analysert med tanke på den asymptotiske kompleksiteten, mens andre algoritmer kun er beskrevet.

De viktigste algoritmene har jeg progammert i C, og kjørt på en datamaskin for å bedre kunne sammenligne dem. Spesielt gjelder det multiplikasjon, der jeg totalt har sett på 7-8 algoritmer og programmert 3 av dem. Programmering av langtallsaritmetikken er forøvrig noe av det som har krevd mest tid i denne oppgaven.

Ellers synes jeg at oppgaven har blitt en brukbar oversikt over mer enn 30 tallteoretiske algoritmer som har betydning innen moderne kryptologi.

TAKK til alle som har lest korrektur, Oppfinnerne av Mac´en, Små og store Matematikere gjennom tidene, PC-Butikken, Universitetet i Tromsø, Veileder Ben Johnsen, Prof. Loren Olson, Venner, Kjente og spesielt Sonja-Kristin som har holdt ut med en "matematiker" som stort sett har vært nedgravd i bøker det siste året.

Innholdsfortegnelse

Del 1 Om Kryptografi og Eksponensieringskryptosystemer

1.1 Om Kryptografi 3
1.2 Eksponensieringskryptosystemer 7

Del 2 Kompleksitet

2.1 Algoritmer og kompleksitet 11
2.2 Turingmaskiner 13
2.3 Kompleksitetsklasser 16
2.4 Reduksjoner 17
2.5 Om NP-Komplette problemer 17
2.6 Andre kjente NP-Komplette problemer 19
2.7 Kompleksitet og algoritmene som behandles i oppgaven 22

Del 3 RSA´s tidskritiske algoritmer

3.1 Langtallsaritmetikk........ 25
3.2 Addisjon – Skolealgoritmen........ 27
3.3 Subtraksjon – Skolealgoritmen........ 29
3.4 Multiplikasjon – Skolealgoritmen........ 31
3.5 Splitt og Hersk multiplikasjon........ 34
3.6 FFT – Rask Fourier Transformasjon........ 40
3.7 En Iterativ versjon av FFT........ 47
3.8 FFT multiplikasjon av heltall $\leq 10^{75.000.000}$........ 50
3.9 Modulær Multiplikasjon........ 59
3.10 Schönhage - Strassen metoden........ 63
3.11 Divisjon – Skolealgoritmen........ 66
3.12 Rask Eksponensiering........ 71
3.13 Konklusjon........ 72

Del 4 RSA – Andre algoritmer

4.1 Største Felles divisor........ 74
4.2 Inverser modulo n........ 77
4.3 Primtallstesting (Stokastiske algoritmer)........ 79
4.4 Rumely-Adleman algoritmen for primtallstesting........ 82

Del 5 Algoritmene som sikkerheten til RSA avhenger av

5.1 Innledning........ 85
5.2 Faktorisering generelt........ 86
5.3 Pollard´s Rho algoritme........ 87
5.4 Pollard´s $p - 1$ algoritme........ 90
5.5 Elliptiske kurve Faktorisering........ 92
5.6 Generelt om kvadratiske rest algoritmer........ 100
5.7 Faktorisering ved bruk av delbrøkoppspaltning........ 101
5.8 Kvadratisk sil metoden........ 103
5.9 Den diskrete logaritme........ 104
5.10 Konklusjon........ 107

Tillegg 1 Div. Programlisting

Del 1
Om Kryptografi og Eksponensieringskryptosystemer

I denne delen av oppgaven ser jeg kort på:

- *Kryptografi*
- *Datasikkerhet*
- *Eksponensieringskryptosystemer (for et skrekkelig ord...)*
- *Hvilke algoritmer som trengs for å realisere disse*

1.1 Om Kryptografi

1.1.1 Innledning

Denne hovedoppgaven handler om algoritmene til eksponensieringskryptosystemer av typen Pohlig-Hellman og RSA. Den første algoritmen ble utviklet i 1978 av Pohlig og Hellman. Like etter kom Rivest, Shamir og Adleman med en offentlig nøkkelversjon av denne. Det er denne algoritmen (RSA) som har fanget min interesse, men siden Pohlig-Hellman algoritmen ligger så nært, er det naturlig å se litt på den også. RSA metoden førte til en liten revolusjon innen kryptologien. Ved å bruke RSA er det ikke nødvendig å avtale nøkler før utvekslingen av meldinger tar til. Alle som er tilknyttet nettet har en offentlig og en hemmelig nøkkel. Den offentlige nøkkelen kan publiseres i en slags telefonkatalog som vi kan kalle kryptokatalogen, mens den hemmelige er det bare eieren (mottakeren) som vet.

Skal Anne sende melding til Bjørn slår Anne opp i kryptokatalogen og finner Bjørns offentlige nøkkel som hun krypterer meldingen med. Når Bjørn mottar meldingen dekrypterer han den med sin hemmelige nøkkel. Ingen kan i praksis beregne den hemmelige nøkkelen utfra den offentlige uten å få noe tilleggsinformasjon, som selvfølgelig også er hemmelig. Det er dette som kalles enveisfunksjoner. Pohlig-Hellman og RSA er dermed gode kandidater til å være henholdsvis enveisfunksjon og enveis felle funksjon.

I tillegg til å være hemmelig kan meldingene også signeres slik at man kan være sikker på hvem som har sendt dem. Vil man ha både krypterte og signerte meldinger er dette også enkelt å realisere.

Til tross for alle fordelene med RSA har systemet bare slått igjennom på enkelte områder og det er to hovedgrunner til dette. For det første tar RSA-kryptering/dekryptering mye tid. I dag (1.3-90) opererer man med en fart på 145 Kbit/s, eller ca. fem maskinskrevne A4 sider pr. sekund. Dette gjelder en hardware versjon med blokkstørrelse 507 bit [1]. De raskeste softwareutgaver opererer til sammenligning med en fart på 6 Kbit/s og raskeste PC utgave har en fart på 1,1 Kbit/s, begge med blokkstørrelse 512 bit.

---

[1] RSA-performance – Mark Shand, Denis Laurichesse og Yves Deswarte

I tillegg til dette har utviklingen på faktoriseringsfronten vært stor. Faktorisering av tall på rundt 100 siffer er ikke uvanlig. Likevel er RSA i bruk bl.a på områder der sikkerheten har høy prioritet, datamengden er moderat og der kravet til krypteringshastighet ikke er for stort. RSA brukes også som nøkkelutvekslingsmetode for andre raskere krypteringsalgoritmer.

Både Pohlig-Hellman og RSA krever svært mange beregninger i forhold til andre krypteringsalgoritmer. Denne hovedoppgaven vil derfor ta for seg de fleste beregningsproblemene som er forbundet med eksponensieringskryptosystemer, slik som langtallsaritmetikk, rask eksponensiering, faktorisering, primtallstesting, osv. Jeg vil spesielt ta for meg de tidskritiske algoritmene som krypteringen og dekrypteringen er avhengig av for at RSA (og Pohlig-Hellman) skal kunne arbeide raskt. I tillegg kommer algoritmer som trengs for å sette opp systemet, slik som primtallstesting, største felles divisor osv. Jeg skal også se litt på hvilke algoritmer som er viktige i kryptoanalysen, siden det gir oss et mål på hvor sikre kryptosystemene er. Et kapittel om kompleksitet er tatt med for å lettere kunne sammenligne forskjellige algoritmer. Kort sagt skal denne oppgaven prøve å gi svar på hvilke algoritmer som egner seg best i forhold til hvor store oppgaver de skal løse. Men først litt generelt om kryptografi.

1.1.2 Hvorfor er kryptografi nødvendig?

Helt fra gammel tid har folk innsett fordelene med å ha hemmeligheter, og det å eventuelt kunne dele dem med noen få utvalgte. Spesielt innenfor militærvesenet har hemmelige koder vært brukt i over 2000 år. Opplysninger som ikke måtte falle fienden i hende ble f.eks skrevet på lærreimer som var surret rundt trestokker med en spesiell diameter. Mottakerne måtte surre lærreima opp på en stokk med samme diameter for å kunne lese meldingen. Kryptografien har stort sett vært forbeholdt militær virksomhet og utenrikstjenesten helt frem til våre dager. Men i dag er situasjonen forandret. Datateknologien har gjort det mulig å lagre enorme mengder informasjon på svært liten plass. Denne informasjonen kan lett aksesseres, kopieres eller endres på brøkdeler av et sekund. Dette er svært praktisk for dem som skal bruke teknologien, men det åpner samtidig for andre som vil misbruke den, siden det i mange tilfeller er lett å komme seg sporløst inn og ut av dataanlegg.

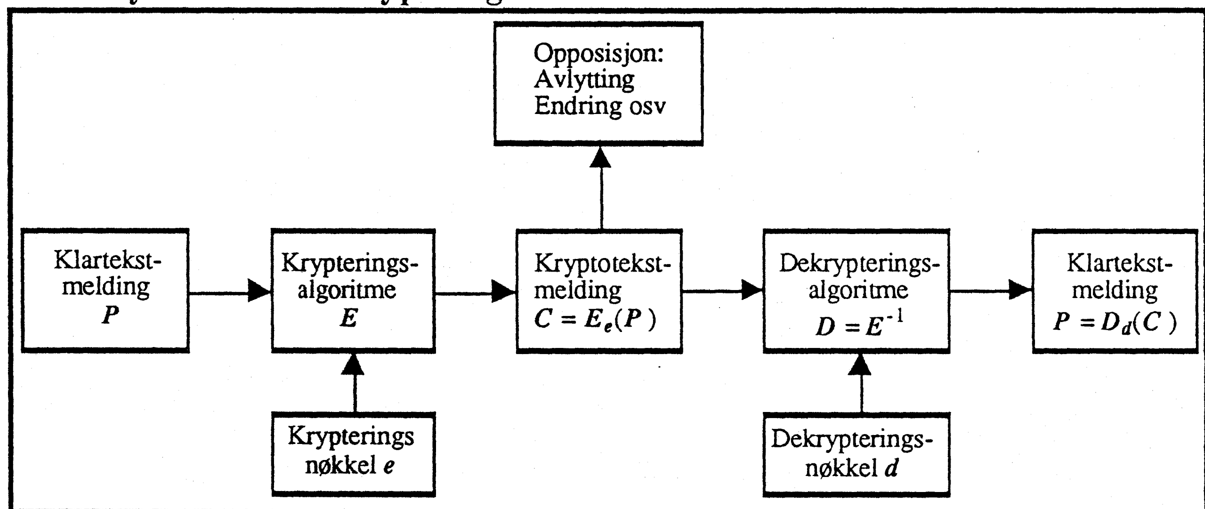
Informasjonen som ligger lagret på datamedia har etterhvert fått et stort omfang, og det er grunn til å tro at dette vil øke i årene som kommer. De fleste bedrifter lagrer opplysninger om varer, kunder, ansatte, bedriftens mål, styrker og svakheter på data. Offentlige institusjoner har opplysninger som sykehusjournaler, strafferegistre, sosiale forhold, skole osv. Bankene vet alt om dine økonomiske forhold, ditt pengeforbruk, når, hvor og hvor mye penger du brukte på hva osv. Mer og mer av posten sendes elektronisk, også den delen som inneholder sensitive opplysninger. Satelittkommunikasjon kringkaster meldingene som sendes, alt fra kanal TV til telefonsamtaler. I tillegg har vi ambassadene over hele verden som rapporterer hjem, og sist, men ikke minst, det militære. Alt dette og mer til er opplysninger som ikke alle bør ha adgang til, og det må derfor beskyttes på en eller annen måte. Kryptografi er i så måte bra egnet. I moderne datasystemer der et stort antall datamaskiner er knyttet sammen i nett, er det rett og slett ikke mulig, eller ihvertfall meget upraktisk, å bruke tradisjonelle kryptografiske metoder der deltagerne på forhånd må avtale nøkler. Det er her de offentlige nøkkel systemene kommer inn.

1.1.3 Kryptosystemer [2]

Ei melding skrevet i **klartekst** (norsk, pascal, talldata...) skal sendes over en usikker **kanal** (telefonlinje, radio, diskett,...) til en mottaker (person, datamaskin,...). Prosedyren med å omskrive meldingen, slik at usikkerheten til kanalen motvirkes, kalles **koding** dersom målet er at tilfeldige feil kan oppdages og/eller rettes, og **kryptering** dersom målet er å gjøre meldingen uleselig for andre enn den er beregnet for. **Kryptografi** er læren om hvordan meldinger kan skrives på en slik måte at uvedkommende ikke kan lese dem. **Kryptoanalyse** er læren om hvordan kryptert tekst kan analyseres for å bestemme meldingen som var sendt, og **kryptologi** omfatter begge disse.

Vi skiller mellom **symmetriske** og **asymmertriske** kryptosystemer. Eksempler på symmetriske er de klassiske metodene inkludert Pohlig-Hellman, der krypterings- og dekrypteringsnøkkelen er like, eller at de er lett å beregne ut fra hverandre. Asymmertriske (offentlig nøkkel) systemer har to nøkler som har den egenskapen at det ikke finnes noen effektiv algoritme for å beregne den ene fra den andre. RSA tilhører denne gruppen.

Den asymmetriske kryptologiske kanalen kan skisseres slik:



Formelt er et (asymmetrisk) kryptosystem [3] en familie av invertible transformasjoner E med en parameter $e \in \mathcal{K}$, der $\mathcal{K}$ er et **nøkkelrom** med endelig størrelse. Kryptosystemet skal ha følgende egenskaper:

1. Hvis $\mathcal{P}$ er **klartekstrommet** og $\mathcal{C}$ er **kryptotekstrommet** så har vi at krypterings-algoritmen $E_e : \mathcal{P} \rightarrow \mathcal{C}$ for en fiksert nøkkel $e \in \mathcal{K}$ er en invertibel transformasjon fra klartekstrommet inn i kryptotekstrommet. Dvs. $E_e(P) = C$ der $P \in \mathcal{P}$ og $C \in \mathcal{C}$.
2. Vi har en invers algoritme $E_d^{-1} = D_d$ som kalles **dekrypteringsalgoritmen.** $D_d : \mathcal{C} \rightarrow \mathcal{P}$ slik at $D_d(C) = D_d(E_e(P)) = P$.
3. Nøklene bør entydig definere den krypterte klarteksten: $E_{e_1}(P) \neq E_{e_2}(P)$ hvis $e_1 \neq e_2$.
4. Krypterings / dekrypteringsalgoritmen må være effektiv for alle nøkler.
5. Systemet må være lett å bruke.
6. Systemets sikkerhet skal bare avhenge av nøklene, ikke av algoritmene E og D.

[2] Kryptografi – Ben Johnsen s.1
[3] Cryptography: An Introduction to Computer Security – Seberry/Pieprzyk s.2
Cryptography and Data Security – Denning s.7

7. Det må være beregningsmessig umulig å beregne P ut fra C uten å kjenne nøkkelen d. Det må også være umulig å finne nøkkelen d fra C selv om korresponderende P er kjent.
8. Det må være beregningsmessig umulig å finne d fra e (uten hemmelig tilleggsinformasjon).

Et symmetrisk kryptosystem blir på samme måte som over, bortsett fra at nøklene e og d er de samme, eller at det er lett å beregne den ene nøkkelen fra den andre. Punkt 8 faller helt bort.

Et kryptosystem er **kommutativt** hvis
1. $D_d(E_e(P)) = E_e(D_d(P))$.
2. Klarteksten og kryptoteksten ligger i samme rom.

1.1.4 Datasikkerhet

Datasikkerhet er et stort område som omfatter alt fra brannsikring til kryptografi. Jeg vil imidlertid bare ta for meg de sidene av datasikkerheten som angår RSA. Disse kan deles inn i 4 punkter:

Avlytting er å tilegne seg informasjon som man ikke har adgang til. Det kan gjøres ved å koble seg på telefonkabelen, oppfange stråling fra kabelen, ta inn radiosignaler fra radiokommunikasjon osv. En mulig måte å gjøre avlytting uinteressant på er å kryptere meldingene som sendes.

Endring av meldinger kan være å endre innholdet i ei melding slik at feil eller misforståelser oppstår. En svindler kan endre innholdet på sin egen bankkonto eller en konkurrent kan endre viktig infomasjon for å lage kaos i en bedrift. Løsninger på slike problemer kan være å kryptere meldingene, siden det er svært vanskelig å endre uleselige meldinger slik at de dekrypteres til noe fornuftig.

Sending av gamle meldinger på nytt er vanskelig å beskytte seg mot hvis man ikke har f.eks. et nytt nummer eller et klokkeslett som knyttes til hver sending. Hvis vi i tillegg krypterer meldingene må en eventuell avlytter endre meldingene for at de skal virke troverdige.

Autentisering har med å lage en melding slik at mottakeren er sikker på hvem som sendte den. Hvem sendte den elektroniske posten, hvem logget seg inn på datamaskinen eller hvem tok ut penger fra minibanken? Dette er ofte viktig å vite.

RSA kan brukes til å løse alle disse problemene.

1.2 Eksponensierings kryptosystemer

For å kryptere ei melding må den først oversettes til tall på en eller annen måte, siden alle moderne krypteringsalgoritmer tar utgangspunkt i tall. Det finnes mange måter å gjøre dette på uten at jeg skal gå nærmere inn på det. En av de enklere er: A → 00, B → 01,..., Å → 28 osv. Deretter grupperes tallene i blokker på $2m$ desimalsiffer, der m er antall bokstaver i blokken, slik at tallet 2828 . . . 28 ($2m$ siffer) $< p$ (Pohlig-Hellman) eller n (RSA). En slik blokk kaller vi en **klartekstblokk** og betegner den P.

1.2.1 Pohlig-Hellman metoden [4]

Pohlig-Hellman metoden er et kommutativt, symmetrisk kryptosystem. La p være et odde primtall og la $e \in \{3, p-2\}$ et positivt heltall med $\gcd(e,p-1) = 1$ være krypteringsnøkkelen. For hver klartekstblokk P, beregnes **kryptotekstblokken** C ved å benytte krypteringsalgoritmen $E_e(P) = P^e \pmod p$.

For å dekryptere en kryptotekstblokk C må dekrypteringsnøkkelen d beregnes. d er en invers av e (mod $p-1$), dvs. $e \cdot d \equiv 1 \pmod{p-1}$. Inversen d eksisterer siden $\gcd(e,p-1) = 1$. Vi kan nå få tilbake P ved å bruke dekrypteringsalgoritmen $D_d(C) = C^d \pmod p$.

Metoden fungerer siden $C^d \equiv (P^e)^d = P^{ed} = P^{k(p-1)+1} = (P^{p-1})^k \cdot P \equiv P \pmod p$ der $d \cdot e = k(p-1) + 1$, dvs. $d \cdot e \equiv 1 \pmod{p-1}$, og pga. Fermats lille teorem som sier at $P^{p-1} \equiv 1 \pmod p$ hvis $\gcd(P,p) = 1$.

Pohlig-Hellman metoden er bra egnet til systemer som krever stor sikkerhet og har få brukere, og der hurtigheten ikke spiller så stor rolle.

Hvis nettet tillater toveiskommunikasjon kan metoden lett brukes av mange. Dette siden Pohlig-Hellman metoden også kan brukes til å avtale nøkkelen e på en sikker måte.

La oss se på et eksempel med to brukere Anne og Bjørn. De avtaler først a og p som ikke trenger å holdes hemmelig. Anne velger seg nøkkel k_1 og Bjørn velger seg nøkkel k_2. Anne sender så $y_1 \equiv a^{k_1} \pmod p$ til Bjørn, som finner felles nøkkel e ved å beregne $e \equiv {y_1}^{k_2} \equiv a^{k_1 k_2} \pmod p$. Bjørn sender så $y_2 \equiv a^{k_2} \pmod p$ til Anne, som beregner $e \equiv {y_2}^{k_1} \equiv a^{k_2 k_1} \pmod p$ og de har felles nøkkel e. Ved å bruke denne metoden har de ingen kontroll med hva den hemmelige nøkkelen blir, men det spiller vel mindre rolle så lenge sikkerheten ivaretas.

1.2.2 RSA (Shamir, Rivest, Adleman) metoden [5]

RSA er et kommutativt, asymmetrisk kryptosystem. Til RSA algoritmen trengs det et heltall n som er produktet av 2 store primtall p og $q \neq p$. Velg så en krypteringsnøkkel $e \in \{3, n-2\}$ slik at $\gcd(e,\phi(n)) = 1$. $\phi(n)$ er antall heltall i der $1 \leq i < n$ slik at $\gcd(i,n) = 1$.

For å sende ei melding P beregner vi kryptotekstblokken C ved å bruke krypteringsalgoritmen $E_e(P) = P^e \pmod n$.

[4] Elementary Number Theory and its Applications – K.Rosen s.224
[5] Elementary Number Theory and its Applications – K.Rosen s.230

For å dekryptere meldinga må vi beregne en dekrypteringsnøkkel d slik at $e \cdot d \equiv 1 \pmod{\phi(n)}$.
Så bruker vi dekrypteringsalgoritmen $D_d(C) = C^d \pmod n$, som fungerer siden $C^d = (P^e)^d = P^{ed} = P^{k\phi(n)+1} \equiv (P^{\phi(n)})^k P \equiv P \pmod n$. Dette pga. Eulers teorem som sier at $P^{\phi(n)} \equiv 1 \pmod n$ hvis $\gcd(P,n) = 1$.
Hvis $\gcd(P,n) > 1$ (meget usannsynlig !!), kan vi anta at $P = p \cdot j$ for $j < q$. Da har vi at $(p \cdot j)^{q-1} \equiv 1 \pmod q$ ved Fermat's teorem. Det følger at $(p \cdot j)^{k\phi(n)+1} \equiv p \cdot j \pmod q$ siden $\phi(n) = (p-1)\cdot(q-1)$. Vi har også at $(p \cdot j)^{k\phi(n)+1} \equiv p \cdot j \pmod p$. Ved å bruke det kinesiske restleddsteoremet (CRT) følger det at $(p \cdot j)^{k\phi(n)+1} \equiv p \cdot j \pmod n$.
RSA fungerer på samme måte som Pohlig-Hellman, men siden det er asymmetrisk har vi i tillegg muligheter for å sende krypterte meldinger til hvem det måtte være uten å ha avtalt nøkkel på forhånd. Dette siden hver enhet som er tilkoblet nettet har offentliggjort nøkkelen e og tallet n. p, q og d er hemmelige.
Siden systemet er både asymmetrisk og kommutativt kan Anne (A) sende ei signert melding til Bjørn (B) ved å "kryptere" den med sin hemmelige nøkkel d. Hun beregner da $D_{dA}(P) = P^{dA} \pmod n$. Bjørn "dekrypterer" meldingen ved $E_{eA}(C) = C^{eA} \pmod n$. Han vet at meldingen kommer fra Anne siden det bare er hun som kan lage ei melding som kan dekrypteres til noe fornuftig ved å bruke Annes offentlige nøkkel.
Anne kan i tillegg gjøre meldingen hemmelig på vanlig måte ved å kryptere med Bjørns e i tillegg, dvs. hun beregner $E_{eB}(D_{dA}(P)) = \left(P^{dA}\right)^{eB} = C$.
Bjørn dekrypterer ved å beregne $D_{eA}(E_{dB}(C)) = \left(\left(\left(P^{dA}\right)^{eB}\right)^{dB}\right)^{eA} = P^{eA \cdot dA \cdot eB \cdot dB} = P$ siden $eA \cdot dA \equiv 1 \pmod{\phi(n)}$ og $eB \cdot dB \equiv 1 \pmod{\phi(n)}$.

1.2.3 Et eksempel på RSA metoden

Anne har:
$nA = pA \cdot qA = 5737 \cdot 7433 = 42643121$
$eA = 521$
$dA = 39602489$

Bjørn har:
$nB = pB \cdot qB = 6287 \cdot 7001 = 44015287$
$eB = 911$
$dB = 30864191$

Anne skal sende den hemmelige meldingen "GOD PÅSKE" til Bjørn. Klartekstblokkene blir:
$P_1 = 06140315$ $P_2 = 28181004$

Anne beregner:
$C_1 = E_{eB}(P_1) = P_1^{eB} \pmod{nB} =$
$E_{911}(06140315) = 06140315^{911} \pmod{44015287} = 39373276.$
På samme måte blir $C_2 = 04858652$.
Hun sender C_1 og C_2 til Bjørn.

Bjørn dekrypterer meldingen ved å beregne:
$P_1 = D_{dB}(C_1) = C_1^{dB} \pmod{nB} =$
$D_{30864191}(39373276) = 39373276^{30864191} \pmod{44015287} = 06140315.$
På samme måte blir $P_2 = 28181004$ og Bjørn leser med stor glede "GOD PÅSKE".

1.2.4 Algoritmene til RSA / Pohlig-Hellman

La oss se litt på algoritmene som brukes til å opprette systemet, krypteringen, dekrypteringen og kryptoanalysen av RSA og Pohlig-Hellman systemene. For hver klarteksblokk P beregner vi $C \equiv P^e$ (mod p eller n) ved å bruke **Rask ekponensiering**. Denne gjør bruk av de 4 elementære operasjonene **Addisjon, Subtraksjon, Multiplikasjon** og **Divisjon**. Rask eksponensiering kan gjøres ved å bruke $k \cdot M(n) \cdot n$ operasjoner, der k er en konstant og $M(n)$ er antall operasjoner som trengs for å multiplisere n-bits tall. Disse algoritmene er de tidskritiske, dvs. de må kunne utføres raskt dersom eksponensieringskryptosystemene skal ha noen hensikt. Disse algoritmene blir hovedtemaet i del 3 i oppgaven.

For å sette opp systemet trenger vi en del andre algoritmer. **Største felles divisor** trengs for å kontrollere at enkelte tall er innbyrdisk primiske, f.eks e og $\phi(n)$. Denne algoritmen trenger $k \cdot M(n) \cdot \log n$ operasjoner for n-bits tall. Vi må også kunne beregne en **Invers** av e modulo $(p - 1)$ eller $\phi(n)$ for å finne dekrypteringsnøkkelen. Det kan også gjøres ved å bruke $k \cdot M(n) \cdot \log n$ operasjoner, siden vi kan finne inverser ved å gjøre noen små endringer på algoritmen for største felles divisor.

For å finne primtallet p (og q) må vi ha en **Primtallstestealgoritme** som kan finne store primtall. De raskeste slike algoritmer bruker $(\log n)^{c \log \log \log n}$ operasjoner der n er tallet som testes og c er en konstant. Siden de fleste heltallene er sammensatte må denne algoritmen kjøres noen ganger før vi finner et primtall.

De tre siste algoritmene utføres bare når man setter opp systemet, skifter nøkkel eller skifter primtallene p og q. Siden dette skjer sjelden trenger de ikke å være spesielt raske. De sistnevnte algoritmene ser jeg på i del 4 av oppgaven.

Ser vi det hele fra kryptoanalysens synsvinkel så er det meget vanskelig å knekke meldinger som er kryptert med Pohlig-Hellman metoden. La oss si at vi vet primtallet p, klartekstblokken P og den korresponderende kryptotekstblokken C slik at $C \equiv P^e$ (mod p). Målet er å finne krypteringnøkkelen e. Når vi har at $C \equiv P^e$ (mod p) sier vi at e er den **diskrete logaritmen** til C med base P modulo p. Det eksisterer forskjellige algoritmer for å finne Diskrete logaritmer til en gitt base modulo et primtall, og de raskeste trenger $\exp(\sqrt{\log p \log \log p})$ operasjoner. Det vil si at å finne diskrete logaritmer modulo et primtall med n desimalsiffer krever omtrent like mye arbeid som å **Faktorisere** tall med samme lengde. RSA bygger mye av sikkerheten på at det er vanskelig å faktorisere store tall og beregne diskrete logaritmer. Disse to algoritmene bør følgelig ta lengst mulig tid. Algoritmene som sikkerheten til RSA avhenger av vil bli tatt opp i del 5 av oppgaven.

Litteratur som er brukt til del 1:

Cipher Systems : The protection of Communications
Henry Beker og Fred Piper, Northwood Books 1982

Cryptography and data security
Dorothy Denning, Addison-Wesley 1983

Kryptografi
Kompendium av Ben Johnsen, UiTø 1987

Elementary Number Theory and its Applications
Kenneth H. Rosen, Addison-Wesley 2. Utgave 1988

Cryptography : An Introduction to Computer Security
Jennifer Seberry, Josef Pieprzyk, Prentice-Hall 1989

RSA performance – Artikler fra sci. crypt (gruppe i Usenet News)
Mark Shand, Denis Laurichesse og Yves Deswarte, Jan/Feb 1990

Del 2
Kompleksitet

Denne delen av oppgaven omhandler:

- *Generelt om algoritmer og kompleksitet*
- *Turingmaskiner*
- *Kompleksitetsklasser*
- *NP-komplette problemer*

2.1 Algoritmer og kompleksitet

2.1.1 Algoritmer

Ei **algoritme** [6] er ei oppskrift som består av en endelig mengde entydige regler, som spesifiserer en endelig sekvens av operasjoner som gir løsningen til et problem eller klasse av problemer, dvs. ei slags nøyaktig kokebokoppskrift på hvordan man skal løse et problem.

2.1.2 Kompleksitet

Kompleksiteten til en algoritme er målt i tidsforbruk (T) og lagringsplassbehov (S). T og S er funksjoner av n, der n er lengden på input. Lengden er som regel målt i antall bits som trengs for å representere inputdataene i maskinen. Dette gjelder ikke alltid. Noen ganger er n tallet selv og ikke lengden. Sorteringsalgoritmer måler lengden på input ved antall ting som skal sorteres.

En tidsfunksjon $f(n)$ er vanligvis uttrykt på formen $O(g(n))$, der $f(n) = O(g(n))$ betyr at det eksisterer en konstant c slik at $f(n) \leq c \cdot |g(n)|$ når n går mot uendelig. Her er det viktig å merke seg at konstanten c forsvinner i O-notasjonen.

At en funksjon $g(n)$ vokser asymptotisk raskere enn en annen funksjon $f(n)$ kan vi skrive som:

$$f(n) < g(n) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

For eksempel har vi at $n < n^2$. Vi kan si at n vokser langsommere enn n^2. Det er mange funksjoner av n utenom potenser av n. Vi kan bruke < relasjonen til å rangere funksjonene asymptotisk sett, dvs når n går mot uendelig. Eksempeler på dette kan være:

$\log\log n < \log n < n^e < n \cdot \log n < n^c < n^{\log n} < c^n < n^n < c^{c^n}$ for $0 < e < 1 < c$.

Alle funksjonene som er listet opp her går mot uendelig når n går mot uendelig, men det som er interessant er hvor fort.

Skal vi drive på med asymptotiske analyser må vi tenke stort, for det er lett å få feil inntrykk for små verdier av n. For eksempel sier rangeringen av funksjonene at $\log n < n^{0,0001}$. Kan dette være riktig da? Setter vi inn et for oss stort tall, nemlig 1 googol, eller 10^{100}, for n får vi at log

[6] Algorithms : Their Complexity and Efficiency – Lydia I. Kronsjö

n blir 100 mens $n^{0,0001}$ blir rundt 1,0233. Men når ting går mot uendelig er 1 googol et lite tall, så la oss prøve noe større, slikt som en googolplex [7] som er 10^{googol}. Her blir situasjonen forandret. log n er nå blitt 10^{100}, mens $n^{0,0001}$ er blitt $10^{10^{96}}$ som er endel større. Dette viser at man ikke kan se isolert på den asymptotiske kompleksiteten. I tillegg til dette kommer den skjulte konstanten bak O-notasjonen, som i praksis villeder oss mest. For å finne den raskeste algoritmen for å løse et gitt problem må vi mao. finne den skjulte konstanten til hver enkelt algoritme som løser problemet. Så kan vi sammenligne algoritmene med den lengden på input som vi skal bruke dem til. Det kan hende at en metode er raskest for korte input, en annen er raskest for middels lange input og en tredje er raskest deretter. Målet blir å finne skjæringspunktene til kjøretiden for hver enkelt algoritme. En tredje ting som vi må ta hensyn til er at ikke alle algoritmer er like lett å realisere på en datamaskin.

2.1.3 Kompleksiteten til en algoritme *f* er det maksimale tidsforbruk (eller plassforbruk) for et gitt inndata som en funksjon av n. Dvs. "worst case". Det kan være praktisk å se på worst case av flere grunner. For det første garanterer det at algoritmen ihvertfall ikke trenger mere tid eller plass. I tillegg er det ofte lettere å finne worst case enn gjennomsnitts-kompleksiteten.
Bakdelen er at det er enkelte algoritmer som har betydelig bedre ytelse gjennomsnittlig enn worst case. Et elsempel på dette er sorteringsalgoritmen Quicksort som har worst case kompleksitet på $O(n^2)$, mens gjennomsnitts-kompleksiteten er $O(n \cdot \log_2 n)$.
Dette er viktig å sjekke for algoritmer som skal brukes til kryptologi. Det kan tenke seg at en lovende enveisfunksjon har worst case f^{-1} i **NP–P** mens gjennomsnittstilfellet ligger i **P**. For kryptologiske algoritmer er det ikke engang godt nok at gjennomsnittstilfellene har eksponensiell vekst. Vi må i tillegg være sikker på at andelen av tilfeller som kan løses i polynomiell tid er forsvinnende liten.

2.1.4 Kompleksiteten til problemet *f* er minste kompleksitet av alle tenkelige algoritmer som løser f, dvs. problemets egentlige kompleksitet som slett ikke trenger å være oppdaget. Det er forøvrig de færreste problemer som er bevist å ha en gitt kompleksitet.

2.1.5 Kompleksitet og datamaskiner

Mange har den oppfatningen at for å få et program (algoritme) til å gå raskere må man kjøpe en raskere datamaskin. Dette er bare delvis rett. Ser vi på den øverste tabellen (bakerst i denne delen) ser vi utviklingen i antall maskinoperasjoner for algoritmer med forskjellig kompleksitet. Vi ser at det har dramatiske konsekvenser om vi kan bytte ut ei 2^n algoritme med ei n^2. Generelt ser vi at de polynomielle algoritmene klarer store inputlengder, mens de eksponensielle algoritmene raskt får problemer selv for korte inputlengder. Grensen for antall operasjoner som det er mulig å utføre på en datamaskin i løpet av et år ligger på rundt 10^{16}.
Tabellen under viser effekten av å innvistere i kraftigere datautstyr. Kjøper vi en datamaskin som er 100 ganger raskere, får vi brukbar uttelling for de polynomielle funksjonene, mens for de eksponensielle funksjonene oppnår vi nærmest ingenting. Med uttelling mener jeg å kunne løse større tilfeller av et problem, dvs. større lengde på input.

[7] Betegnelsene googol og googolplex kommer fra Kasner,Newmann, 1940

2.1.6 O-manipulasjon

$f(n) = O(f(n))$

$c \cdot O(f(n)) = O(f(n))$

$O(O(f(n))) = O(f(n))$

$O(f(n))O(g(n)) = O(f(n)g(n))$

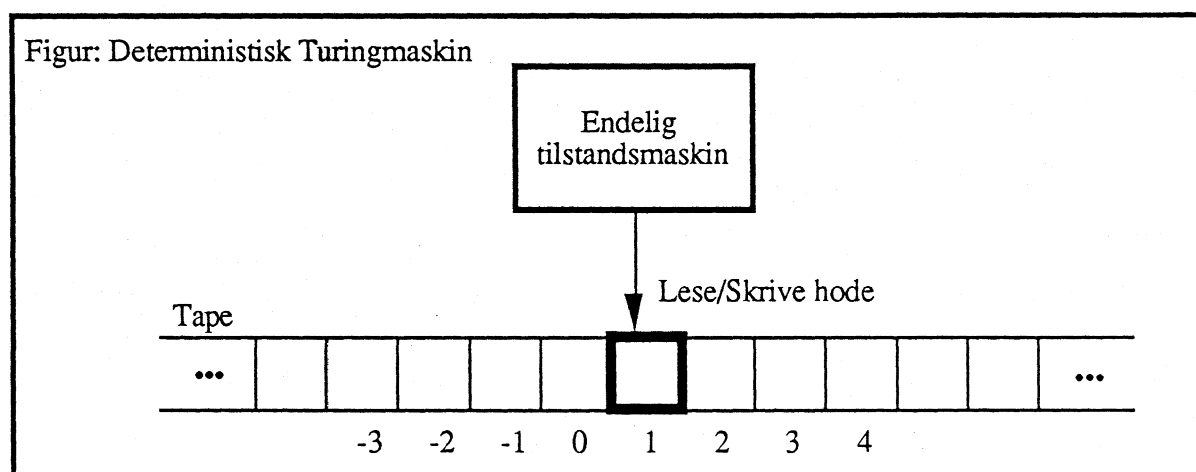
$O(f(n)g(n)) = f(n)O(g(n))$

2.2 Turingmaskiner

2.2.1 Deterministiske Turingmaskiner

For å formalisere notasjonen til algoritmer laget Alan Turing en enkel modell av en datamaskin. Dette for å frigjøre analysen av algoritmer fra spesielle implementasjoner. Vi bør også unngå restriksjoner rundt tids- eller plassforbruk. En vanlig Turingmaskin (TM) tilfredstiller disse kravene. En TM er en endelig tilstandsmaskin utstyrt med en uendelig tape som den kan spole frem og tilbake samt lese fra og skrive på. Til tross for sin enkelhet, så kan den, bare den får nok tid på seg, løse like vanskelig problemer som selv de mest avanserte datamaskiner kan løse. Problemer som er løselige i polynomiell tid på en TM er også løselig i polynomiell tid på en vanlig maskin og motsatt.

Kompleksitetsteori klassifiserer problemer avhengig av tids- og plassforbruk som trengs for å løse den "hardeste" utgaven av et problem på en Turingmaskin. Tiden til en beregning avhengiger av antall overganger, dvs. skifte av tilstand, som foretas.



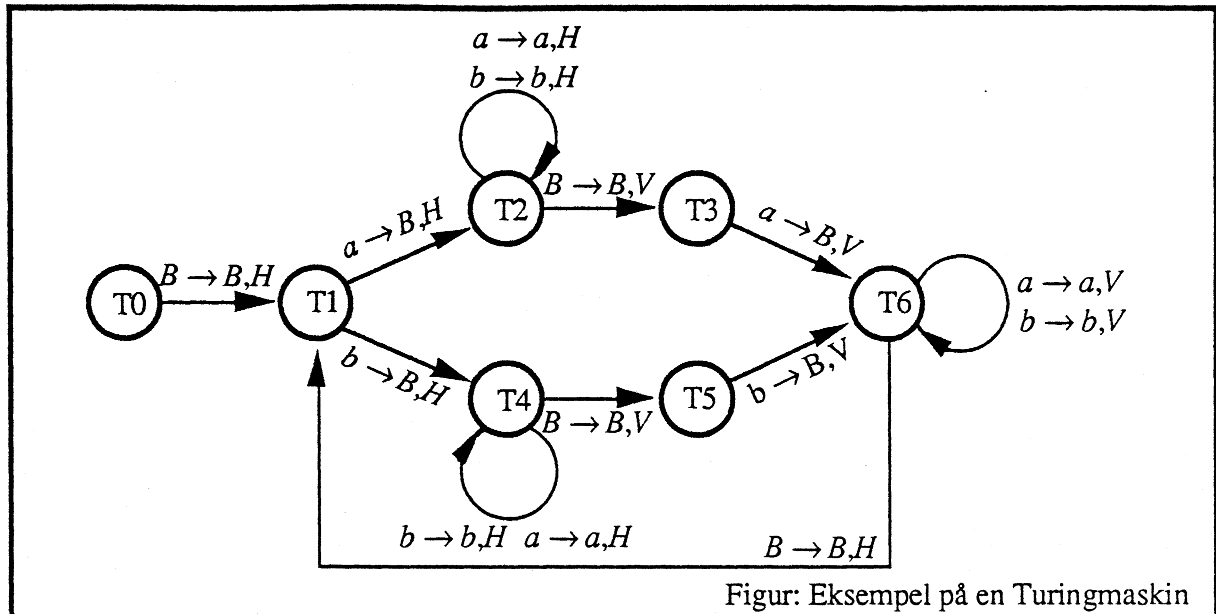
2.2.2 Et eksempel på en Turingmaskin: [8]

La oss finne tidskompleksiteten til en TM (skissert under) som aksepterer palindromer, dvs. symmetriske ord, over alfabetet $\{a,b\}$. At et ord er akseptert vil si at TM stopper i en bestemt tilstand.

F.eks en TM som aksepterer primtall vil dermed stoppe i en gitt tilstand hvis tallet er et primtall. I motsatt fall er det ikke sikkert at vi kan si noe som helst.

---

[8] Interaktive bevissystemer og "Zero-knowledge" i kryptografi – Live Stensholt



Figur: Eksempel på en Turingmaskin

Symbolforklaring:

T0 – T6 er de mulige tilstandene maskinen kan være i.

B betyr blank, og *a* → *B*,*V* betyr at maskinen har lest en *a*, skrevet *B* og flyttet skrive / lesehodet et steg mot venstre. *H* betyr høyre.

Beregningene består i at maskinen sammenligner det første ikke-blanke tegnet på tapen med det siste. Det første symblolet erstattes med en blank i tilstand 1 (T1). Så går vi 1 steg videre til høyre (H) på tapen og til T2 eller T4 avhengig av hvilket tegn som ble lest. Etter det går vi til T3 eller T5, der vi finner ut om det siste tegnet stemmer med det første eller ikke. Stemmer det ikke blir vi stående i T3 eller T5. La oss ta et eksempel på et palindrom: BabbaB

Tilstand	Tapen(Fet skrift der lese/skrive hodet er)
T0:	***B**abbaB*
T1:	*B**a**bbaB*
T2:	*BB**b**baB*
T2:	*BBb**b**aB*
T2:	*BBbb**a**B*
T2:	*BBbba**B***
T3:	*BBbb**a**B*
T6:	*BBb**b**BB*
T6:	*BB**b**bBB*
T6:	*B**B**bbBB*
T1:	*BB**b**bBB*
T4:	*BBB**b**BB*
T4:	*BBBb**B**B*
T5:	*BBB**b**BB*
T6:	*BB**B**BBB*
T1:	*BBB**B**BB*

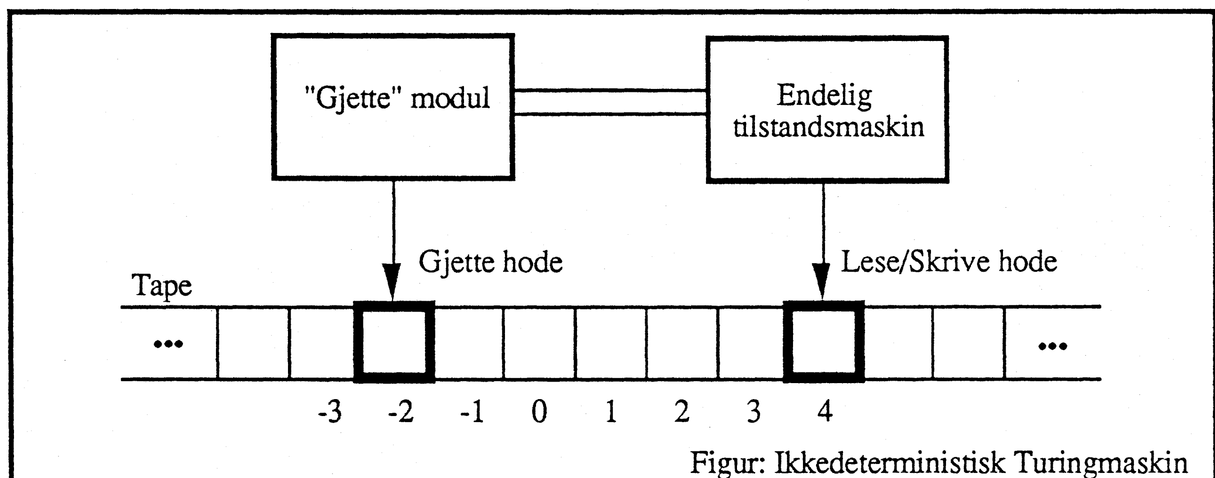
I dette tilfelle fikk vi 15 overganger for et ord med lengde 4 og det er maks antall overganger vi kan få for et ord med denne lengden. Vi kan selvfølgelig få færre overganger hvis vi ikke har et palindrom. Det vi er interesert i er imidlertid antall overganger for palindromer med en gitt lengde.

Definisjon La TM være en Turingmaskin. Tidskompleksiteten til TM er funksjonen $t_{TM} : N \rightarrow N$, slik at $t_{TM}(n)$ = maks antall overganger når inputordet har lengde n.

La oss beregne tidskompleksiteten til eksemplet med palindromene. TM aksepterer $\{uu^R \mid u \in \{a,b\}^*\}$, der u^R betyr det motsatte ordet til u og $\{a,b\}^*$ betyr alfabetet av a-er og b-er. Vi ser at første gang vi beveger oss mot høyre får vi $n + 1$ overganger siden vi ikke bare flytter oss innenfor ordet av a-er og b-er, men også fra første til siste blanke. På veien tilbake får vi n overganger, siden vi nå har blanket ut det som opprinnelig var første tegn i ordet. Slik fortsetter vi og tidskompleksiteten til TM blir:

$$t_{TM}(n) = \sum_{i=1}^{n+1} i = \frac{n^2 + 3n + 2}{2} \quad \text{eller } O(n^2).$$

2.2.3 Ikke-Deterministiske Turingmaskiner [9]



Figur: Ikkedeterministisk Turingmaskin

Til forskjell fra en deterministisk Turingmaskin (TM), kan en ikke-deterministisk TM (ITM) velge mellom flere, dvs. et endelig antall overganger til neste tilstand. Et ord (inputstreng) x aksepteres hvis minst en overgangssekvensene fører til at TM aksepterer ordet. Vi kaller disse overgangssekvensene for AO.

For et ord x kan vi se på en ITM som en vanlig TM som utfører alle mulige overganger i parallell helt til ordet er akseptert, eller at det ikke finnes flere mulige overganger. Det betyr at etter k overganger kan vi forestille oss at det finnes et visst antall kopier av ITM som er i forskjellige tilstander. Før den $(k + 1)$-te overgangen kopierer en gitt kopi C seg selv i j kopier hvis ITM C har j valg for neste overgang. Det betyr at de mulige AO-er som ITM kan få ut av ordet x, kan sees på som et tre av AO-er, der hver vei fra rota til et blad i treet representerer en sekvens av overganger. Hvis σ er den korteste sekvensen av AO-er, så vil ITM stoppe så snart den har utført σ overganger. Tiden for å prosessere x blir mao lengden til σ.

[9] The Design and Analysis of Computer Algorithms - Aho/Hopcroft/Ullman s.364

Hvis x ikke har noen AO-er forkastes x.
Som regel forestiller man seg ITM som en vanlig TM utstyrt med en "gjette" modul som alltid gjetter riktig. Det betyr at den gjetter seg til sekvensen σ for deretter å sjekke at σ virkelig er en AO.
Siden en vanlig TM ikke kan gjette seg til svar, vil en deterministisk simulering av ITM føre til at vi måtte sjekke alle mulige overgangsekvenser av x på en eller annen måte for å finne den korteste AO-en. Dette resulterer i at den deterministiske versjonen raskt vil holde på i "all evighet", enten fordi enkelte av overgangsekvensene er svært lange eller fordi de er mange.

2.3 Kompleksitetsklasser [10]

En algoritme er **Polynomiell** (tid) hvis kjøretiden T, dvs. antall maskinoperasjoner, er gitt ved $O(n^t)$ for en konstant t. Den er konstant hvis $t = 0$, lineær hvis $t = 1$ og kvadratisk hvis $t = 2$ osv.
En algoritme er **Eksponensiell** (tid) hvis $T = O(t^{h(n)})$ for en konstant t og et polynom $h(n)$.
Problemer som kan beregnes i polynomiell tid kalles "lette" og alle andre kalles "harde".
Enkelte problemer er så harde at de er uløselige. Et eksempel på dette er problemet om å avgjøre om et tilfeldig (data)program noen gang vil stoppe.
Se figur over kompleksitetsklassene bakerst i denne delen.
Klassen **P** består av alle problem som kan løses i polynomiell tid.
Klassen **NP** (Nondeterministic polynomial) består av alle problem som kan løses i polynomiell tid på en ikke-deterministisk TM. Dette betyr at hvis maskinen gjetter svaret, så kan den sjekke om det er rett i polynomiell tid. Men det er selvsagt ikke noen garanti på at maskinen vil gjette riktig. Å løse problemet systematisk ser ut til å trenge eksponensiell tid. **NP** $\supseteq$ **P** siden alle problem som kan løses på en deterministisk TM også kan løses i polynomiell tid på en ikke-deterministisk TM. Selv om problemene i **NP** ser "vanskeligere" ut så er det ikke bevist at **P** $\neq$ **NP**.
Klassen **CoNP** består av alle problemer som er komplementet til et eller annet problem i **NP**. Mens **NP** problemer er på formen: "avgjør om det eksisterer ei løsning" er **CoNP** på formen: "vis at det ikke eksisterer noen løsninger". Man vet ikke om **NP** = **CoNP**, men det er problemer som befinner seg i snittet **NP** $\cap$ **CoNP**. Et eksempel på dette er sammensatt-tall problemet. Gitt et heltall n. Er n sammensatt eller primtall? Ikke alle problemer er på denne formen, og det er derfor antatt at **NP** $\neq$ **CoNP** [11].
Av andre kompleksitetsklasser har vi **PSPACE**, som består av problemer som kan løses i polynomielt rom, men ikke nødvendigvis polynomiell tid. **PSPACE** inkluderer **NP** og **CoNP** men det er problemer i **PSPACE** som synes hardere enn problemene i **NP** og **CoNP**.
PSPACE-komplett har egenskapen at hvis et av problemene er i **NP**, så er **PSPACE** = **NP**, og hvis et av problemene er i **P** så er **PSPACE** = **P**.
Klassen **EXPTIME** består av alle problem som er løselige i eksponensiell tid.

---

[10] Cryptography and Data Security – Denning s.30
[11] Cryptography: An Introduction to Computer Security – Seberry/Pieprzyk s.36

2.4 Reduksjoner [12]

Det er som regel vanskelig å beregne kompleksiteten til problemene vi møter. Derfor er det praktisk å sammenligne den relative kompleksiteten til to eller flere problemer. Det vil si hvor vanskelig et problem er i forhold til et annet. Det er to grunner til å gjøre dette. La oss si at vi er istand til å bevise at noen problemer er ekvivalente på den måten at de har omtrent samme kompleksitet. Hvis vi på et senere tidspunkt finner en mer effektiv algoritme for et av disse problemene, vil vi samtidig ha funnet en mer effektiv algoritme for de andre også. Dette virker motsatt også. Kan vi vise at en av algoritmene ikke kan løses på noen raskere måte gjelder det da også for de andre algoritmene.

La A og B være to problemer. A er **lineært reduserbart** til B, betegnet $A \leq^l B$, hvis det at det eksisterer en algoritme for B som arbeider i $O(t(n))$ tid, for en eller annen funksjon $t(n)$, impliserer at det eksisterer en algoritme for A som også arbeider i $O(t(n))$ tid. Hvis $A \leq^l B$ og $B \leq^l A$ så er A og B lineært ekvivalente, betegnet $A \equiv^l B$.
Eksempeler på problemer som er lineært ekvivalente er multiplikasjon, divisjon og kvadrering av heltall (se del 2.7).

La X og Y være to (eksponensielle) problemer. X er **polynomielt reduserbart** til Y, hvis vi kan løse X i polynomiell tid dersom det eksisterer en algoritme som løser Y uten å bruke tid. Dvs algoritmen som skal løse X kan få så mye hjelp den vil ha fra en imaginær algoritme som kan løse Y uten å bruke tid. En annen måte å si dette på er at vi kan omforme problemet X til problemet Y i polynomiell tid.
Dette betegnes $X \leq^p Y$. Når $X \leq^p Y$ og $Y \leq^p X$ samtidig, sier vi at X og Y er ekvivalente og betegner det $X \equiv^p Y$.

2.5 Om NP-Komplette problemer [13]

NP-Komplette problemer er ei samling problemer som er interesante, siden det ikke eksisterer effektive algoritmer for å løse noen av dem. De er også spesielle på den måten at hvis det blir funnet ei effektiv algoritme for å løse et av dem, så betyr det at det finnes ei effektiv algoritme for alle. Dette gjelder også motsatt. Kan det bevises at et av problemene ikke har noen effektiv løsning, så gjelder det for de andre også. Selv om ingen har bevist at at disse problemene er eksponensielle, så er det sterke indisier på at de er det. Teorien for **NP-Komplette** problemer er nyttig i den forstand at hvis et nytt problem viser seg å være **NP-Komplett**, så er det gode grunner for å la de eksakte løsningene ligge til fordel for gode approksimasjoner.

Det er kjent at **NP** $\supseteq$ **P**, men om klassene er like er ikke kjent, og det er derfor teorien for **NP-Komplette** problemer ble utviklet. Teorien ble opprinnelig utviklet for Ja/nei problemer (decision problems), der hver utgave av problemet har en ja eller nei løsning. Et eksempel på dette er å avgjøre om et tall er et primtall eller ikke. Det er ikke alle problemer i **NP** som er ja/nei problemer, men det blir enklere om vi formulerer dem på den måten siden kompleksiteten er den

[12] Algorithmics – Brassard, Bratley s.300 –
[13] Infeasible Computation: NP-Complete Problems – Edmund A. Lamagna

samme. Vi kan illustrere dette ved et eksempel som på "godt norsk" kalles subset sum problemet. Vi har en mengde $A = \{x_1, x_2, \ldots x_n\}$ med n positive heltall og et annet positivt heltall G, som vi kaller målet. Problemet er å finne en delmengde av mengden A med den største summen som ikke overskrider G. En praktisk versjon av problemet kan være hvis A er en mengde med n forskjellige gullklumper og G er maks vekt som ryggsekken vår tåler. Spørsmålet er nå hvor mye gull kan vi maksimalt ta med oss? En ja/nei versjon av problemet kan være om vi kan fylle sekken nøyaktig til bristepunktet. Vi har 2^n forskjellige delmengder av en mengde med n elementer, og skal vi sjekke alle brukes $O(2^n)$ tid.
Definisjonen av **NP-Komplette** problemer er som kjent alle (ja/nei) problemer som kan løses i polynomiell tid på en ikke deterministisk Turingmaskin. Siden ikke deterministiske TM er en imaginær modell av en datamaskin, må man i praksis ta til takke med de vanlige deterministiske maskinene. Da brukes ofte en teknikk som kalles backtracking, og som for disse problemene har en eksponensiell vekst i tid.
Et problem er komplett i en kompleksitetsklasse hvis det ligger i klassen, og er minst like vanskelig som hvilket som helst annet problem i klassen. De komplette problemene er derfor de "hardeste" problemene i klassen. For **NP** problemer vil det si at problemet kan løses deterministisk ved å bruke et tidsforbruk som er polynomielt forskjellig fra tidsforbruket til det "vanskeligste" problemet i **NP**. Det vil igjen si at et problem er **NP-Komplett** hvis det ligger i **NP**, og hvis man finner en polynomiell tid algoritme som løser problemet, så vil det implisere at alle andre problem i **NP** også kan løses i polynomiell tid. Det er forøvrig mulig for et problem å være i **NP** uten å være komplett for klassen.

2.5.1 Hvordan vise at et problem er NP-Komplett ? [14]

Å bevise at et problem et NP-Komplett kan være en tøff oppgave, men det finnes snarveier. Når vi først har bevist at et problem er **NP-Komplett** blir arbeidet med å bevise at andre problem også er **NP-Komplett** mye lettere. Gitt et problem X som vi vet er i **NP**, så kan vi vise at det er **NP-Komplett** ved å velge et allerede kjent **NP-Komplett** problem X', og vise at X' kan reduseres til X i polynomiell tid.

Fremgangsmåte:
1. Vis at X er i **NP**.
2. Velg et kjent **NP-Komplett** problem X'.
3. Konstruer en transformasjon f fra X' til X.
4. Bevis at f er en polynomiell transformasjon.

Husk at vi må vise at et eller annet kjent **NP-komplett** problem kan transformeres i polynomiell tid til vårt nye problem og ikke motsatt.

2.5.2 Subset Sum er NP-Komplett

Jeg har tidligere sett litt på Subset sum problemet, og skal nå vise at dette problemet er **NP-komplett** ved å lage en transformasjon fra Partition problemet. Partition problemet er å avgjøre om en gitt en mengde med heltall kan deles opp i 2 delmengder, der summen av tallene er lik. Jeg går ut fra at partition problemet er **NP-komplett.**
Det må vises at alle tilfeller av Partition kan transformeres til et tilfelle av Subset Sum. La A =

[14] Computers and Intractability – Garey/Johnson s.45

$\{x_1,x_2,...,x_n\}$ være en mengde positive heltall som vi ønsker å dele i 2 delmengder med samme sum. Anta at summen av elementene i A er S, slik at hver av delmengdene får summen $S/2$. Vi kan konstruere et tilfelle av Subset Sum med de samme heltallene og mål $G = S/2$. Derfor kan heltallene $x_1,x_2,...,x_n$ deles inn i 2 mengder med samme sum hvis og bare hvis en eller annen delmengde av tallene summerer seg til $S/2$. Vi kan nå gå ut fra at Subset Sum er **NP-komplett.**

2.5.3 Multiprocessor Sceduling er NP-komplett

Multiprocessor Scheduling er følgende problemstilling. Gitt flere mikroprosessorer og prosesser som ikke nødvendigvis har samme tidsforbruk. Prosessene kan ikke deles, men vi kan fordele prosessene som vi vil på prosessorene. Kan alle prosessene være ferdig innen en gitt tidsfrist?

For å vise at Multiprocessor Sceduling er **NP-komplett** kan vi videre prøve å lage en transformasjon fra Subset Sum. Gitt et tilfelle fra Subset sum. La $x_1, x_2, ..., x_n$ være positive heltall med mål G. La S være summen av tallene.

Vi konstruerer så et Scheduling problem med $n + 2$ prosesser som krever henholdsvis $x_1, x_2, ..., x_n, G + 1, S - G + 1$ tid. Den totale prosessortiden som trengs er mao. $2S + 2$. Vi har 2 mikroprosessorer til disposisjon og deadline er $S + 1$. Vi må vise at alle $n + 2$ prosessene kan gjøres ferdig av 2 prosessorer på $S + 1$ tid hvis og bare hvis summen til en eller annen delmengde av $x_1,x_2, ..., x_n$ er G. Anta først at en delmengde U av $x_1, x_2, ..., x_n$ summeres til G. Da kan prosessor 1 tildeles prosessene med tidsforbruk som tilsvarer delmengden U i tillegg til prosessen som krever $S - G + 1$ tid. Resten tildeles prosessor 2 og arbeidet blir ferdig på $S + 1$ tid.

Anta nå at mikroprosessorene gjør jobben ferdig på $S + 1$ tid. Prosessene som tar $G + 1$ og $S - G + 1$ tid kan ikke tildeles samme prosessor siden den da ville være opptatt minst $S + 2$ tid. Tidsforbruket for de andre prosessene som er tildelt prosessoren som utfører $S - G + 1$ prosessen er G, og derfor må summen av en delmengde av $x_1, x_2, ..., x_n$ være G.

2.6 Andre kjente NP-Komplette problemer [15]

2.6.1 Satisfiability problemet

Teorien om **NP-Komplette** problemer begynte i 1971 da Stephen Cook viste at Satisfiability-problemet er komplett for klassen **NP**. Dette betyr at hvis satisfiability-problemet kan løses i polynomiell tid, så kan alle andre problem i **NP** det også, og kan man bevise at et annet **NP-Komplett** problem er eksponensielt, så gjelder det også for satisfiability-problemet.

En boolesk variabel (literal) x er en variabel som kun kan ha verdiene sann eller usann. Booleske variabler kan settes sammen til større uttrykk ved å bruke operasjonene konjunksjonen-og($\wedge$), disjunksjonen-eller($\vee$) og komplementet-ikke (som betegnes ved å sette en strek over variabelen f.eks $\bar{x}$). En måte å sette sammen booleske uttrykk på er å kombinere dem med $\vee$ slik at de blir en **clause**. F.eks $x_1 \vee \bar{x}_2 \vee x_3$. Denne clausen er sann hvis x_1 er sann eller x_2 er usann eller x_3 er sann. Et boolesk uttrykk er på konjunktiv normalform

[15] Infeasible Computation: NP-Complete Problems – Edmund A. Lamagna

(CNF), hvis det er en konjunksjon av grupper av clauser. 2 eksempler på CNF uttrykk er

$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_4) \wedge (\bar{x}_3 \vee x_4) \wedge (\bar{x}_2)$ og

$(x_1 \vee x_2) \wedge (\bar{x}_1) \wedge (\bar{x}_2)$ mens uttrykket

$(x_1 \wedge \bar{x}_2) \vee (x_3)$ ikke er på CNF form.

CNF uttrykk kan bare være sanne hvis hver og en av clausene er sanne. Uttrykk som kan bli sanne for en eller annen tildeling av verdier til $x_1, x_2, \ldots, x_n$, kalles **Satisfiable**, eller kanskje oppfylt på godt norsk. Satisfiability problemet er med andre ord å avgjøre om booleske uttrykk på CNF form kan oppfylles.

I vårt eksempel 1 ser vi at $x_1 = x_4 =$ sann og $x_2 = x_3 =$ usann gir uttrykket sann verdi. Ikke alle booleske uttrykk kan oppfylles, og de kan derfor ikke få sann verdi for noen tildeling av verdier til $x_1, x_2, \ldots, x_n$. Dette fordi de på en eller annen måte inneholder en kontradiksjon. Et kort eksempel på dette er $x_1 \wedge \bar{x}_1$.

For å vise at Satisfiability problemet er **NP-Komplett** må vi først vise at det er i **NP**. Dette er lett siden en ikke deterministisk TM kan gjette seg til hvilke av variablene som skal være sanne eller usanne, og så verifisere at uttrykket er sant i polynomiell tid. Å vise at problemet er komplett for klassen **NP** krever at en deterministisk polynomiell tids algoritme for løsningen impliserer at alle **NP** problem kan løses i deterministisk polynomiell tid. Dette siste er kjent som Cook´s teorem, og det finner man i Garey & Johnson eller Papadimitriou & Steiglitz.

2.6.2 Andre NP-Komplette problemer:

Etter 1971 har antallet problemer som er klassifisert som **NP-Komplette** steget til flere hundre. Noen av dem følger her:

3-Satisfiability: Samme som Satisfiability, men vi kan maks bruke 3 forskjellige variabler.

Clique: En Clique er en mengde av noder der det er en kant mellom ethvert par av noder. Finnes det en clique med størrelse k?

Colorability: Fargelegging av en graf slik at ingen av nabonodene har samme farge. Kan grafen fargelegges med med mindre enn k farger?

Vertex Cover: Er en delmengde av alle nodene i grafen slik at alle kantene har minst et endepunkt i mengden. Har vi en slik delmengde med størrelse k?

Hamiltonian Circuit: Gitt en graf. Er det mulig å konstruere en vei gjennom grafen som går innom alle nodene og ender opp der vi startet, og slik at alle nodene er besøkt nøyaktig 1 gang?

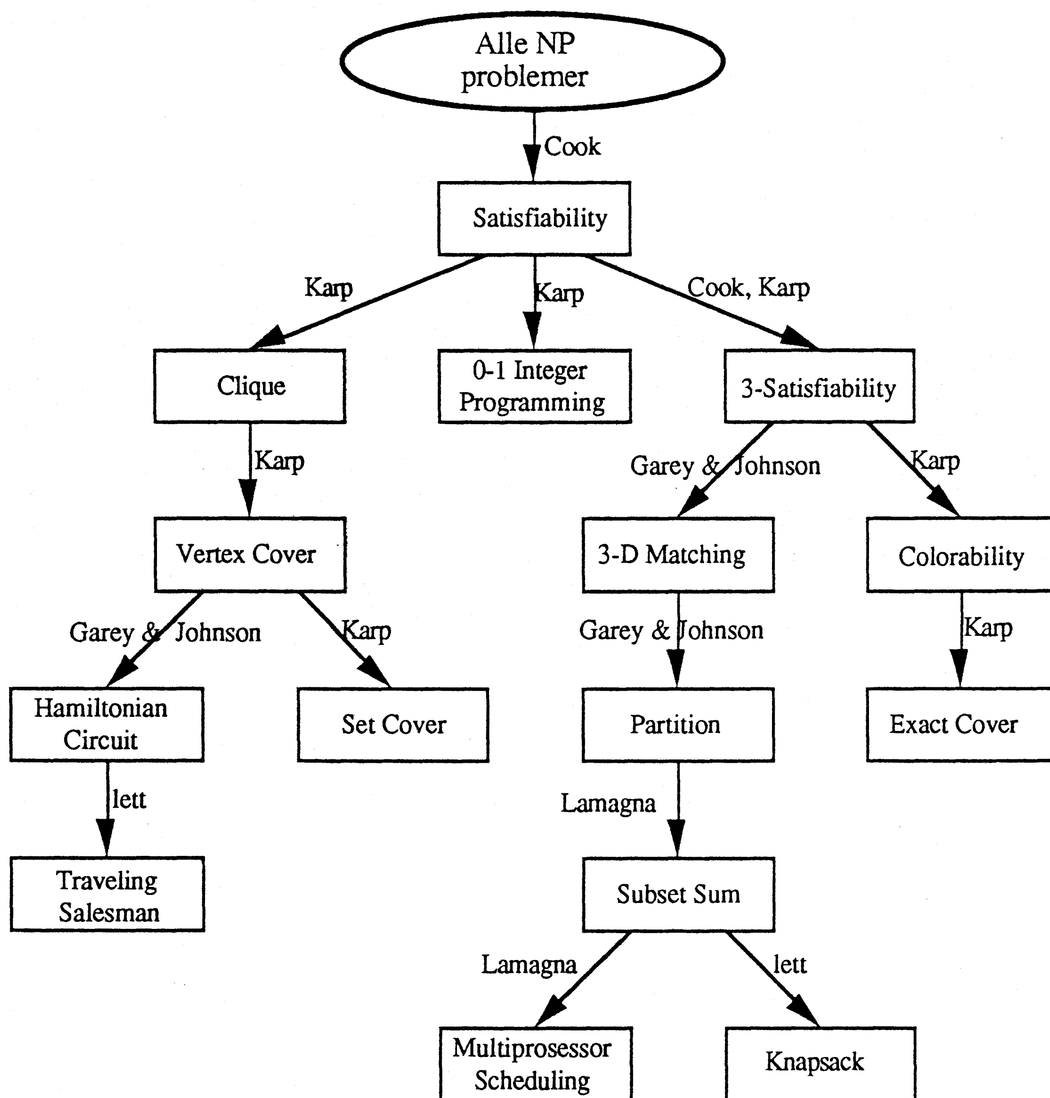
Traveling Salesman: Et særtilfelle av den forrige. Finnes det en vei med lengde mindre et gitt tall?

Set Cover and exact Cover: Et set cover er flere mengder som tilsammen dekker universet og exact cover er flere disjunkte mengder som tilsammen dekker universet. Spørsmålet er: Kan universet dekkes med unionen av k mengder?

3-D Matching: Det klassiske gifteproblemet er et typisk 2-D matching problem. 3-D blir f.eks: Gitt 3 like lange lister med mennesker, arbeidsplasser og bosted og regler for akseptable tildelinger. Kan vi tildele et passende arbeid og bosted for hver person?

0-1 Integer Programming: Produktet mellom en m x n matrise C og en 1 x m søylevektor $\tilde{x}$ er en søylevektor $\tilde{z}$ der $z_i = \sum_{j=1}^{n} c_{ij}x_j$. Spørsmålet er om det eksisterer en vektor $\tilde{x}$ med nuller og enere slik at $C\tilde{x} \geq \tilde{z}$, dvs. at alle elementene i produktet er større enn eller lik tilsvarende element i $\tilde{z}$.

Figuren som følger er en oversikt over hvordan man påviser at problemene nevnt tidligere er **NP-komplette**, med henvisning til litteratur der det er beskrevet. Pilene viser transformasjonsretningen. Utgangspunktet er Cooks teorem fra 1971 som beviser at Satisfiability problemet er **NP-komplett**. For å bevise at nye problemer er det samme, konstruerer man en polynomiell tid transformasjon fra Satisfiability til det nye problemet. Nå har man 2 **NP-komplette** problemer som man kan transformere til evt. andre nye kandidater.



2.7 Kompleksitet og algoritmene i oppgaven

Addisjon og subtraksjon er linært reduserbart til hverandre. Det samme gjelder for multiplikasjon, divisjon og kvadrering [16].

Problemene største felles divisor og Inverser (mod n) er antageligvis linært reduserbart til hverandre, men det har jeg ikke funnet stoff om.

Den diskrete logaritme, faktorisering og primtallstesting kan løses i eksponensiell tid. Derfor er disse algoritmene mest interessante sett fra et kompleksitetsteoretisk (og kryptologisk) synspunkt. Resten av algoritmene kan løses i polynomiell tid.

Det er ikke bevist, men sterkt antatt at både faktorisering og den diskrete logaritme ikke tilhører klassen **P**. Disse tilhører sannsynligvis heller ikke klassen **NP-Komplett**. Den diskrete logaritme ser ut til å ha samme asymptotiske kompleksitet som faktorisering [17].

Primtallstesting derimot er et mere usikkert problem. Det ser ut til å være mye lettere enn faktorisering, men om det tilhører **P** er usikkert, men ikke helt utenkelig. Dersom den generaliserte Riemann-hypotesen er sann, så eksisterer det en deterministisk polynomiell-tids algoritme for å avgjøre om et heltall er primtall eller ikke [18]. Det finnes forøvrig en del stokastiske polynomiell-tids algoritmer for primtallstesting.

Av andre ting som er verdt å nevne er at Pohlig-Hellman og RSA systemene er gode kandidater til å være henholdsvis enveisfunksjoner og enveis felle funksjoner. En **enveisfunksjon** er en invertibel funksjon f der $f(x) = y$ kan beregnes i **P**, mens $f^{-1}(y)$ tilhører **NP** – **P**. En **enveis felle funksjon** fungerer på samme måte, bortsett fra at $f^{-1}(y)$ tilhører **NP** – **P** hvis vi ikke vet noe tilleggsinfomasjon om hvordan fellen er konstruert (i tilfellet RSA er det primtallene p og q). Om disse funksjonene virkelig eksisterer er fortsatt et åpent spørsmål og et aktivt forskningsområde.

---

[16] Algorithmics – Brassard, Bratley s.309
The Design and Analysis of Computer Algorithms - Aho/Hopcroft/Ullman s.280
[17] Discrete Logarithms in Finite Fields and their Cryptographic significance – A.M.Odlyzko
[18] Elementary Number Theory and its Applications – K.Rosen s.178

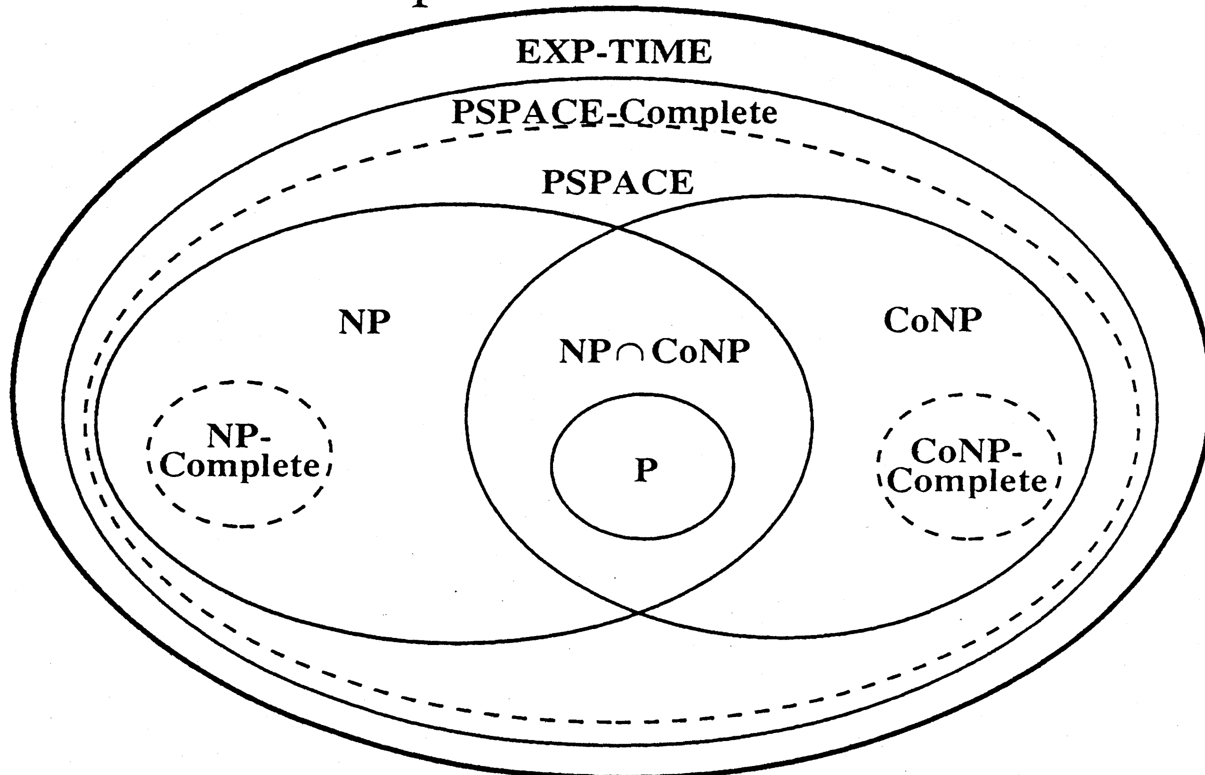
Kompleksitetstabell

Inputlengde →	10	100	1000	10000	100000
Funksjon	Antall maskin-operasjoner				
n	10	100	1000	10000	100000
$n \cdot \log_2 n$	34	665	9966	132878	1660965
n^2	100	10000	10^6	10^8	10^{10}
2^n	1024	$1.27 \cdot 10^{30}$	$1.07 \cdot 10^{301}$	$2.00 \cdot 10^{3010}$	$1.00 \cdot 10^{30103}$
n!	3628800	$9.33 \cdot 10^{157}$	$4.02 \cdot 10^{2567}$	$2.85 \cdot 10^{35659}$	$>10^{400000}$

Betydningen av raskere datamaskiner

Kompleksitets funksjon	Største lengde på input som er mulig å utføre på en dag		
	Med dagens teknologi	Med en datamaskin som er 100x raskere	Med maskin som er 10000x raskere
n	$k_1 = 8.6 \cdot 10^{11}$	$100 \cdot k_1$	$10000 \cdot k_1$
$n \cdot \log_2 n$	$k_2 = 2.5 \cdot 10^{10}$	$84 \cdot k_2$	$7296 \cdot k_2$
n^2	$k_3 = 929516$	$10 \cdot k_3$	$100 \cdot k_3$
2^n	$k_4 = 39$	$7+k_4$	$13+k_4$
n!	$k_5 = 14$	$2+k_5$	$4+k_5$

Kompleksitetsklassene



Litteratur som er brukt til denne delen:

The Design and Analysis of Computer Algorithms
Aho, Hopcroft, Ullman, Addison Wesley 1974

Cipher Systems : The protection of Communications
Henry Beker og Fred Piper, Northwood Books 1982

Algorithmics: Theory and practice
G. Brassard, P. Bratley, Prentice-Hall 1988

Cryptography and data security
D. Denning, Addison-Wesley 1983

Computers and Intractability : A guide to the Theory of NP-Completeness
M.R.Garey og D.S.Johnson, W.H.Freemann & Co. 1979

Combinatorial Algorithms
T.C.Hu, Addison-Wesley 1982

Kryptografi
Ben Johnsen, UiTø 1987

Algorithms : Their Complexity and Efficiency
Lydia I. Kronsjö, Wiley 1979

Infeasible Computations : NP-Complete Problems
Edmund A. Lamagna, Abacus Vol.4 no.3 1987

Discrete Logarithms in Finite Fields and their Cryptographic significance
A.M.Odlyzko, Advances in Crypograpy. EUROCRYPT 84

Combinatorial Optimization : Algorithms and Complexity
C. H. Papadimitriou og K. Steiglitz, Prentice-Hall 1982

Elementary Number Theory and its Applications 2. Utgave
Kenneth H. Rosen. Addison-Wesley 1988

Del 3
RSA´s tidskritiske algoritmer

I denne delen av oppgaven ser jeg på:

- *Langtallsaritmetikk*
- *Skole-Addisjon*
- *Skole-Subtraksjon*
- *Skole-Multiplkasjon*
- *Splitt og Hersk multiplikasjon*
- *Rekursiv FFT multiplikasjon*
- *Iterativ FFT multiplikasjon*
- *Modulær aritmetikk*
- *Schönhage-Strassen multiplikasjon*
- *Skole-Divisjon*
- *Rask Eksponensiering*

3.1 Langtallsaritmetikk

3.1.1 Generelt

Vår vanlige desimale representesjon av heltall er nært knyttet til polynomer. F.eks. så kan et vanlig n-sifret desimalt heltall $(a_{n-1}\ldots a_1a_0)_{10}$ skrives som

$$a_{n-1}10^{n-1}+ \ldots + a_1 10 + a_0.$$

Dvs. verdien til polynomet $\sum_{i=0}^{n-1} a_i x^i$ i $x = 10$.

Denne polynomrepresentasjonen kan generaliseres til alle grunntall $G \geq 2$ slik at et n-sifret heltall $(a_{n-1}\ldots a_1a_0)_G$, $0 \leq a_i < G$, med grunntall G representerer tallet

$$a_{n-1}G^{n-1}+ \ldots + a_1 G + a_0.$$

Dvs. verdien til polynomet $\sum_{i=0}^{n-1} a_i x^i$ i $x = G$.

Skal man regne med store tall på en datamaskin må man ta hensyn til endel ting. Datamaskinen kan som regel bare regne med heltall som er mindre enn maskinens ordlengde W. Vanligvis er den 16 ,32 eller 64 bit. For å regne med heltall > W, må man skaffe seg en programpakke for langtallsaritmetikk. Slike pakker er som regel vanskelig å få tak i. Alternativet er å programmere en selv, noe som forøvrig tar lang tid.

Den mest vanlige måten å representere et stort tall a på er $a = (a_{n-1}a_{n-2}\ldots a_1a_0)_G$, der G er grunntallet slik som beskrevet over. Det er en fordel å bruke så stort grunntall som mulig, med $G \leq W$, siden tidforbruket for elementæraddisjon og multiplikasjon er så godt som uavhengig av størrelsen på tallene.

Ved addisjon må grunntallet være < $W/2$, og ved multiplikasjon < $\sqrt{W}$. Det bør sjekkes i hver

enkelt algoritme at ingen av regneoperasjonene kan gi resultater større enn W. Vil man bruke større grunntall kan man lagre sifrene i floating point variabler. Disse har vanligvis større kapasitet. På Macintosh opptil 96 bit presisjon.
Større maskiner har ofte dobbel presisjon på heltallsaritmetikk, slik at man velge grunntall fritt ≤ ordlengden både for addisjon og multiplikasjon. I denne oppgaven har jeg stort sett gått ut fra maskiner uten dobbel presisjon på heltallsaritmetikk.
Sifrene er lagret i n lagerenheter, et siffer i hver lagerenhet, der n er lengden til tallet. Hvis a har lengde n så er $a < G^n$. Skal man programmere selv er det praktisk å velge $G = 10^n \leq W$, slik at det er lett å konvertere mellom grunntall G og grunntall 10. I tillegg er det mye lettere å rette feil under programmeringen.
Alt av prosedyrer (unntatt de som er skrevet i pascal) har jeg programmert selv. Som programmeringsspråk har jeg brukt C. C er et såkalt "lavt høynivåspråk" som er raskt, samtidig som det er lett å bruke.

Tallene i de fleste (unntatt de i del 3.4 og 3.8) programeksemplene er på formen:

Element nr.	0	1	2		n
Langtall Array	Antall siffer i tallet	Minst signifikante siffer		...	Mest signifikante siffer

I C starter alle arrays med element 0. Der har jeg valgt å legge lengden til tallet. Det er praktisk å ha med lengden hvis man skal regne med tall av forskjellig lengde. Deretter følger det minst signifikante siffer, det nest minste osv.
I de rekursive prosedyrene er lengden på tallet lagt i en egen variabel, siden alle tallene i hver prosedyreinkarnasjon har lik lengde. Det minst signifikante siffer ligger derfor i element 0.

3.1.2 Negative tall [19]

Skal man regne med fortegn forverrer det situasjonen en smule. Her har vi 3 vanlige måter å representere negative (og positive) tall på:

A. Den "vanlige" med tegn:
Sett av et bit eller et element i arrayen til tegnet.

Fordeler: Lett å se hvilket tall vi har foran oss.

Bakdeler: Addisjon og subtraksjon krever to forsjellige prosedyrer. Vanskelig å subtrahere siden vi må finne ut hvor store tallene er i forhold til hverandre. F.eks 12 – 14. Subtraherer vi på vanlig måte får vi 98 med en hengende mente på –1. Det vi kan gjøre er å trekke 98 fra 100 slik at vi får 2 som vi gir negativt fortegn. En annen måte er å teste at 14 er større enn 12 og heller trekke 12 fra 14 og gi svaret minustegn.
Vi får 2 representasjoner av 0 nemlig + og – 0.

[19] Computer Organization and Programming – VAX-11 – S.El-Asfouri, O.Johnson, W.K.King s.278 og The Art of Computer Programming: Seminumerical Algorithms – Donald Knuth s. 186

B. 10er komplements notasjon (eller grunntalls komplements notasjon):
Tallene fra 1 til 444...49 er de positive, og de fra 500...00 til 999...99 er de negative. Et negativt tall representeres dermed som 1000...00 minus den positive delen av tallet. Dette er forøvrig den mest vanlige måte å representere heltall på i en datamaskin, og da brukes selvfølgelig 2er komplementsnotasjon.

Fordeler: Bare en versjon av 0. Addisjon og subtraksjon blir enkelt. 12 – 14 blir 12 + 86 = 98 som tilsvarer –2. 14 – 12 blir 14 + 88 = 102. Her glemmer vi den siste menten. Vi kan i tillegg bruke samme algoritme både til addisjon og subtraksjon.

Bakdeler: Litt vanskelig å se at –2 er det samme som 98. Høyreshift er også litt problematisk. F.eks. –11 = ...99989 shiftet 1 plass til høyre gir ...99998 som er det samme som –2. Vi går ut fra at det settes inn 9-ere på venstresiden av tallet. Generelt vil x shiftet 1 plass til høyre gi $\lfloor x/10 \rfloor$ uansett om x er positiv eller negativ. I tillegg er ikke 10er komplements notasjon symmetrisk om 0. Det største negative tall 500...00 er ikke det additivt inverse av noe positivt tall.

C. 9er komplements notasjon (eller grunntalls – 1 komplements notasjon):
Her blir de positive tallene som før, mens de negative tallene blir forandret. Hver siffer i det negative tallet erstattes med 9 minus det aktuelle siffer. Eks. 14 – 12 blir 14 + 87 = 101. Den siste menten fjernes og legges til sifret til høyre slik at svaret blir 02 eller 2. 12 – 14 blir 12 + 85 = 97 som er det samme som –2.

Fordeler: Addisjon og subtraksjon blir enkelt.

Bakdeler: Litt vanskelig å se at -2 er det samme som 97. Vi får 2 representasjoner av 0, nemlig 999...99 og 000...00.

3.2 Addisjon – Skolealgoritmen

3.2.1 Beskrivelse av algoritmen

Skoleaddisjon er den enkleste av de 4 vanlige regneartene. Her foregår utregningen nøyaktig som med papir og blyant. Gitt to tall $a = (a_{n-1}a_{n-2}\ldots a_1a_0)_G$ og $b = (b_{m-1}b_{m-2}\ldots b_1b_0)_G$ med $a \geq b$, så kan de adderes sammen til $c = (c_nc_{n-1}\ldots c_1c_0)_G$ i O(n) tid og O(n) plass. Siden $a,b < G^n$ er $c < 2 \cdot G^n$. Lengden til c er derfor $\leq n + 1$. For å forenkle ting litt går vi ut fra at b har 0-ere i alle siffer med indeks i, $m \leq i < n$.

Vi adderer på den fra før kjente måten:

	$mente_n$	$mente_{n-1}$	...	$mente_1$	
		a_{n-1}	...	a_1	a_0
+		b_{n-1}	...	b_1	b_0
=	c_n	c_{n-1}	...	c_1	c_0

Formelt kan dette skrives som:

$c_0 = (a_0 + b_0) \bmod G$

$mente_1 = (a_0 + b_0) \text{ div } G$

og

$c_i = (a_i + b_i + m_i) \bmod G$

$mente_{i+1} = (a_i + b_i + mente_i) \text{ div } G$

der $0 < i < n$

og $c_n = m_n$

3.2.2 Et program-eksempel i C

Denne prosedyra trenger 2 positive heltall på standardform (Se 3.1.1), og den returnerer svaret på samme form. Svar-arrayet trenger ikke å nullstilles før bruk. Det er det samme hvilket av tallene som er størst. Vi fyller på med ledende 0-ere slik at det minste tallet får samme lengde som det største.

```
void addisjon ( tall1, tall2, svar )
unsigned long  *tall1, *tall2, *svar;
{
	unsigned long  mente, t;
	int  max, i;

	if  ( tall2[0] > tall1[0] ) max = tall2[0];
	else  max = tall1[0];
	mente = 0;

	for ( i=1; i<=max; i++ )
	{
		t = tall1[i] + tall2[i] + mente;
		svar[i] = t % grunntall;
		mente = t / grunntall;
	}

	if  ( mente > 0 )
	{
		svar[max+1] = 1;
		svar[0] = max+1;
	}
	else  svar[0] = max;
}
```

3.2.3 Kort analyse av algoritmen

Menten er alltid 0 eller 1 siden

$$\begin{aligned} mente_{i+1} &= (a_i + b_i + mente_i) \text{ div } G \\ &\leq [2(G-1)+1] / G \\ &= (2G-1)/G < 2 \end{aligned}$$

Hvor stort grunntall kan vi bruke?

Beregningen av *mente* er den største og kan maksimalt bli $(G - 1) + (G - 1) + 1 = 2 \cdot G - 1$. Vi bør derfor velge $G \leq \frac{W+1}{2}$ for å unngå overflow.

Tidsforbruket er avhengig av hvor mange ganger hovedløkka gjennomløpes. Det blir n ganger. Hver gang med 4 operasjoner som tilsammen gir oss $4n$ operasjoner. Kompleksiteten blir dermed $O(n)$. D.Knuth antyder $10n + 6$ maskinsykler.

Plassbehovet er også beskjedent. Vi har to n-sifrete tall som skal adderes, og vi får et $(n + 1)$-sifret svar, dvs. $3n$ eller $O(n)$.

Algoritmen egner seg ikke spesielt godt for parallellprosessering siden menten forplanter seg utover i beregningen. Farten kan maksimalt økes med en faktor 2.

3.3 Subtraksjon – Skolealgoritmen

3.3.1 Beskrivelse av algoritmen

Subtraksjon er den nest enkleste av de 4 regneartene, og utregningen forgår nesten på samme måte som på papir. Gitt to tall $a = (a_{n-1}a_{n-2}\ldots a_1a_0)_G$ og $b = (b_{m-1}b_{m-2}\ldots b_1b_0)_G$ med $a \geq b$, så kan b trekkes fra a, slik at vi får $c = (c_{n-1}c_{n-2}\ldots c_1c_0)_G$, der c_i kan være 0 for alle $i \geq j$, med $0 \leq j < n$. Dette kan gjøres ved å bruke skolealgoritmen for subtraksjon i $O(n)$ tid og $O(n)$ plass.

Siden $a < G^n$ så er $c < G^n$. Lengden til c er derfor $\leq n$. For å forenkle ting går vi ut fra at b har 0-ere i alle siffer med indeks i, $m \leq i < n$.

Vi subtraherer på den vanlige skolemåten:

	$mente_{n-1}$	...	$mente_1$	
	a_{n-1}	...	a_1	a_0
–	b_{n-1}	...	b_1	b_0
=	c_{n-1}	...	c_1	c_0

Formelt kan dette skrives som:

$c_0 = (a_0 - b_0) \bmod G$

$m_1 = \lfloor (a_0 - b_0) / G \rfloor$

og

$c_i = (a_i - b_i + m_i) \bmod G$

$m_{i+1} = \lfloor (a_i - b_i + m_i) / G \rfloor$

der $0 < i < n$

3.3.2 Et program-eksempel i C

Denne prosedyra trenger to positive heltall tall på standardform, og den returnerer svaret på samme form. Svar-arrayet trenger ikke å nullstilles før bruk. Tall1 må være større enn tall2. I tillegg må vi fylle på med ledende 0-ere slik at tall2 får samme lengde som tall1.

```
void subtraher ( tall1 , tall2 , svar )
unsigned long  *tall1, *tall2, *svar;
{
        int  j, k;
        unsigned  long  t;

        k = 0;
        for ( j = 1 ; j <= tall1[0] ; j++ )
        {
                t = tall1[j] - tall2[j] + k;
                if ( t < 0 )
                {
                        t = ( t + grunntall ) % grunntall;
                        k = -1; svar[j] = t;
                }


                else
                {
                        k = 0;
                        svar[j] = t % grunntall;
                };
        }
        j = tall1[0];
        while ( (svar[j] == 0) && ( j > 0 ) )  j--;
        svar[0] = j;
}
```

3.3.3 Kort analyse av algoritmen

Menten er alltid 0 eller –1 bortsett fra siste mente som alltid er 0, siden $a \geq b$. Dette fordi $a_i - b_i + m_i$ maksimalt kan bli $(G - 1)$ og minimalt $-G$. $\lfloor (G - 1) / G \rfloor = 0$ og $\lfloor (-G) / G \rfloor = -1$

Hvor stor kan den største beregningen bli? Beregningen av *mente* er den største, og kan maksimalt (minimalt) bli $0 - (G - 1) - 1 = -G$. Velger vi $G \leq W/2$ er vi på den sikre siden.

Tidsforbruket er avhengig av hvor mange ganger hovedløkka gjennomløpes. Det blir n ganger. Hver gang med (ca) 4 operasjoner som tilsammen blir $4n$. Kompleksiteten til algoritmen blir $O(n)$. Knuth antyder $12n + 3$ maskinsykler.
Plassbehovet er også beskjedent. Vi starter med to n-sifrete tall og får et n-sifret svar dvs. $3n$ eller $O(n)$.

Algoritmen egner seg ikke spesielt godt for parallellprosessering siden menten forplanter seg utover i beregningen. Farten kan, på samme måte som for addisjon, maksimalt økes med en faktor 2.

3.4 Multiplikasjon – Skolealgoritmen

3.4.1 Beskrivelse av algoritmen

Multiplikasjon er den nest mest tidskrevende av de 4 regneartene, bare divisjon tar lengere tid. Den eneste forskjellen på dataprogrammet og papirmetoden er at delsvarene adderes fortløpende, slik at vi slipper å lagre dem.
Gitt to tall $a = (a_{m-1}a_{m-2}\ldots a_1a_0)_G$ og $b = (b_{n-1}b_{n-2}\ldots b_1b_0)_G$.
Da kan de multipliseres sammen til $c = (c_{m+n-1}c_{m+n-2}\ldots c_1c_0)_G$ ved hjelp av skole-algoritmen i $O(n^2)$ tid og $O(n)$ plass.

Skolealgoritmen for multiplikasjon kan lett deles opp i mindre deler.
Skal vi finne $a \cdot b$ kan vi først multiplisere b_0 med hele a, dvs:
$(a_{m-1}a_{m-2}\ldots a_1a_0)_G \cdot (00\ldots 0b_0)_G$. Deretter multipliserer vi b_1 med hele a:
$(a_{m-1}a_{m-2}\ldots a_1a_0)_G \cdot (00\ldots b_10)_G$. Dette gjør vi for alle b_i med $0 \le i < n$:

$\vdots$

$(a_{m-1}a_{m-2}\ldots a_1a_0)_G \cdot (0b_{n-2}\ldots 00)_G$
$(a_{m-1}a_{m-2}\ldots a_1a_0)_G \cdot (b_{n-1}0..00)_G$
Til slutt adderer vi delsvarene og får $c = a \cdot b$. Delsvarene vil i praksis adderes sammen etterhvert som de regnes ut, slik at vi ikke trenger så stor lagringskapasitet.

Vi skal først se på multiplikasjon mellom et m-sifret tall og et en-sifret tall.
$(a_{m-1}a_{m-2}\ldots a_1a_0)_G \cdot (b_k)_G = (c_mc_{m-1}\ldots c_1c_0)_G$

Dette kan skrives som:

$c_0 = (a_0 \cdot b_k) \bmod G$
$mente_1 = (a_0 \cdot b_k) \operatorname{div} G$
og
$c_i = (a_i \cdot b_k + mente_i) \bmod G$
$mente_{i+1} = (a_i \cdot b_k + mente_i) \operatorname{div} G$
der $0 < i < m$
og $c_m = mente_m$

Å **shifte** [20] et tall k plasser er det samme som å multiplisere tallet med G^k. Det vil si å "flytte" tallet k plasser mot venstre.

Selve multiplikasjonsalgoritmen:
Multipliser a med b_k. Shift delsvaret k plasser til venstre, og addér det til evt. tidligere delsvar. Etter å ha gjort dette for alle b_k, $0 \leq k < n$, har vi funnet $c = a \cdot b$.
Siden det siste delsvaret har maksimal lengde $m + 1$, og det er shiftet $n - 1$ plasser til venstre vi sluttsvaret c ha maksimal lengde $(m + 1) + (n - 1) = m + n$.

3.4.2 Et program-eksempel i C

Denne prosedyra trenger to positive heltall på standardform, og den returnerer svaret på samme form.

```
void multipliser ( tall1, tall2, svar )
unsigned long  *tall1, *tall2, *svar;
{
        int  k, i;
        unsigned long  mente, t;

        for  ( k=1; k < tall1[0] + tall2[0]; k++ )  svar[k]=0;  L1

        for  ( k=1; k <= tall2[0]; k++ )
        {
        |       mente = 0;
        |       for  ( i = 1; i <= tall1[0]; i++ )
        |       {
        |       |       t = tall1[i] * tall2[k] + svar[k+i-1] + mente;
        L2      L3      svar[k+i-1] = t % grunntall;
        |       |       mente = t / grunntall;
        |       }
        |       svar[k+i-1] = mente;
        }
        if  ( mente > 0 )
        svar[0] = tall1[0] + tall2[0];
        else  svar[0] = tall1[0] + tall2[0] - 1;
}
```

[20] Elementary Number Theory and its Applications – K.Rosen s.51

3.4.3 Analyse av algoritmen og C prosedyra

Først ser vi på C prosedyra. Hvor stor kan den største beregningen bli?
Beregningen av mente er den største, og kan maksimalt bli $(G-1)\cdot(G-1)+(G-1)+(G-1)=G^2-1$. Velges $G \leq \sqrt{W}$ er man på den sikre siden mot overflow under enkeltberegningene.

Sett m = lengden til tall1 og n = lengden til tall2.
Da krever L1: $m+n$ operasjoner
L2: L3 n ganger
L3: $3m$ multiplikasjoner (divisjoner), $3m$ addisjoner (grovt regnet),
Resten av programmet: 1 addisjon

TILSAMMEN:
$3mn$ multiplikasjoner
$3mn+m+n$ addisjoner
Dette blir $6n^2$ når $n \approx m$ og n går mot uendelig.
Det er verdt å gjøre oppmerksom på at denne konstanten kun er brukbar som sammenligning mellom prosedyrer som er analysert på samme (grove) måte. D.Knuth antyder antall operasjoner til $28mn+7n+4m+3$.

Så over til algoritmen generelt. Multiplikasjonen av et m-sifret tall og et en-sifret tall er $O(m)$. Dette gjøres n ganger, dvs. vi får $O(nm)$ som er $O(n^2)$ når $m \approx n$. Kompleksiteten til skolealgoritmen blir dermed $O(n^2)$.
Plassbehovet er lite. Det trengs m ord til a, n ord til b og $m+n-1$ ord til svaret. Når $m \approx n$ trengs det $4n$ ord. Dvs. $O(n)$.

Til forskjell fra addisjon og subtraksjon er mulighetene for beregne ting i parallell mye større i denne algoritmen. Her kan f.eks delsvarene beregnes parallelt. Har vi k prosessorer, $k \leq n$ får vi redusert tiden til n^2/k.

3.4.4 Tidsforbruk for C-programmet

Kjørt på en Macintosh IIcx med 32-bits 68030 prosessor og 15,67 MHz, $G = 10^4$

Antall siffer (desimale)	Sekunder
64	0,02
128	0,08
256	0,33
512	1,30
1024	5,20
2048	20,78

3.5 Splitt og Hersk multiplikasjon

3.5.1 Beskrivelse av algoritmen

Splitt og Hersk kommer fra det engelske Divide and Conquer. Det betyr å dele opp et problem i flere mindre underproblemer, som tilsammen kan løses raskere enn det opprinnelige problemet. Er det så mulig å multiplisere to tall raskere enn ved vanlig skolemultiplikasjon? Umiddelbart skulle man kanskje ikke tro det, men det kan vi, og metodene er mange og forskjellige. En mulighet er å dele tallene på midten, slik at vi bare får halvparten så lange tall å arbeide med.

Se på multiplikasjonen av to $2n$-sifrete tall med grunntall G.
$A = (a_{2n-1}a_{2n-2}\ldots a_1a_0)_G$ og
$B = (b_{2n-1}b_{2n-2}\ldots b_1b_0)_G$.
Vi kan skrive $A = G^nA_1 + A_2$ og $B = G^nB_1 + B_2$, der
$A_1 = (a_{2n-1}a_{2n-2}\ldots a_{n+1}a_n)_G$ og $A_2 = (a_{n-1}a_{n-2}\ldots a_1a_0)_G$,
$B_1 = (b_{2n-1}b_{2n-2}\ldots b_{n+1}b_n)_G$ og $B_2 = (b_{n-1}b_{n-2}\ldots b_1b_0)_G$
Da er
(1) $A\cdot B = (G^{2n} + G^n)\cdot A_1\cdot B_1 + G^n(A_1 - A_2)\cdot(B_2 - B_1) + (G^n + 1)\cdot A_2\cdot B_2$.
Slik kan vi forsette å dele tallene i to deler helt til de får lengde 1, slik at mikroprosessoren kan multiplisere dem i en operasjon.

3.5.2 Analyse av algoritmen [21]

Ved å bruke denne multiplikasjonsmetoden på to $2n$-sifrete tall, trenger vi bare å utføre 3 multiplikasjoner av n-sifrete tall, i tillegg til noen addisjoner og shiftinger. La $M(n)$ betegne antall operasjoner som trengs for å multiplisere to n-sifrete tall. Fra (1) får vi at
(2) $M(2n) \le 3M(n) + Cn$,
der C er en konstant. Hver av de 3 n-sifrete multiplikasjonene krever $M(n)$ operasjoner. Antall shiftinger og addisjoner er ikke avhengig av n. Hver av disse krever $O(n)$ operasjoner.
Fra (1) kan vi ved å bruke induksjon vise at

(3) $M(2^k) \le c(3^k - 2^k)$, der $c = \text{maks}(M(2),C)$.

Sett $k = 1$. Vi får da at $M(2) \le c$ som stemmer. Vi antar at $M(2^k) \le c(3^k - 2^k)$
Ved å bruke (2) får vi

$$\begin{aligned} M(2^{k+1}) &\le 3M(2^k) + C2^k \\ &\le 3c(3^k - 2^k) + C2^k \\ &\le c3^{k+1} - c\cdot 3\cdot 2^k + c2^k \\ &\le c(3^{k+1} - 2^{k+1}) \end{aligned}$$

[21] Elementary Number Theory and its Applications – K.Rosen s.57

Vi kan da multiplisere sammen to n-sifrete tall ved å bruke $O(n^{\log_2 3})$ operasjoner.

Bevis: Fra (3) har vi at

$$
\begin{aligned}
M(n) = M(2^{\log_2 n}) &\leq M(2^{[\log_2 n]+1}) \\
&\leq c(3^{[\log_2 n]+1} - 2^{[\log_2 n]+1}) \\
&\leq 3c\cdot 3^{[\log_2 n]} \leq 3c\cdot 3^{\log_2 n} = 3cn^{\log_2 3}
\end{aligned}
$$

siden $3^{\log_2 n} = n^{\log_2 3}$

Dermed er $M(n) = O(n^{\log_2 3})$

Et eksempel på Splitt og hersk Multiplikasjon

La oss multiplisere $A = 4321$ med $B = 5678$.

$A_1 = 43 \quad B_1 = 56$

$A_2 = 21 \quad B_2 = 78$

$A\cdot B = (10^4 + 10^2)\cdot 43\cdot 56 + 10^2(43 - 21)\cdot(78 - 56) + (10^2 + 1)\cdot 21\cdot 78$

Vi får da følgende addisjoner etter at delproduktene er shiftet til rett posisjon.

	2	4	0	8				
			2	4	0	8		
				4	8	4		
			1	6	3	8		
+					1	6	3	8
=	2	4	5	3	4	6	3	8

3.5.3 Et program-eksempel i C

Prosedyra splittoghersk tar imot 2 positive heltall med lengde som er en potens av 2.
I element 0 lagres fortegnet som er 1 for positivt og –1 for negativt.

Denne algoritmen var utgangspunktet:

```
Function mult(X,Y,n:integer):integer;
integer   s, Tar vare på fortegn
          m1,m2,m3, De 3 produktene
          A,B,C,D; Tallene etter at de er delt opp

begin
   if n=1 then return X*Y;
   else
   begin
      s=sign(X)*sign(Y);
      X=abs(X);Y=abs(Y); Gjør tallene positive
      A=venstre halvdel av X;
      B=høyre halvdel av X;
      C=venstre halvdel av Y;
      D=høyre halvdel av Y;
      m1=mult(A,C,n/2);
      m2=mult(A-B,D-C,n/2);
      m3=mult(B,D,n/2);
      return
      (s*(m1*2^n+(m1+m2+m3)*2^(n/2) +m3));
   end;
end.
```

```
void shift(tall , plasser, n)
Multipliserer tall med lengde n med
grunntall^plasser
int *tall, plasser, n;
{
   int i;

   for (i=n;i>0;i--)
   {
      tall[i+plasser]=tall[i];
      tall[i]=0;
   }
   for (i=1;i<=plasser;i++) tall[i]=0;
}
```

```
void trekkfra2(x,y,n)
Trekker y med lengde n fra x
int *x,*y,n;
{
   int j,k;
   int t;

   k=0;
   for (j=1;j<=n;j++)
   {
      t=x[j]-y[j]+k;
      if (t<0)
      {
         t=(t+grunntall)%grunntall;
         k=-1;
         x[j]=t;
      }
      else
      {
         k=0;
         x[j]=t%grunntall;
      };
   }
x kan ha lengde > n
   while (k<0)
   {
      t=x[j]-y[j]+k;
      if (t<0)
      {
         t=(t+grunntall)%grunntall;
         k=-1;
         x[j]=t;
      }
      else
      {
         k=0;
         x[j]=t%grunntall;
      };
      j++;
   }
}
```

```
void trekkfra(a,b,c,n)
Trekker b fra a og legger svaret i c.
Alle tallene har lengde n
int *a,*b,*c,n;
{
   int i,k,j,t;

   i=n;
   while (i>0 && (a[i]==b[i])) i--;
   if (b[i]>a[i])
Hvis b ≥ a trekker vi a fra b
   {
      k=0;

      for (j=1;j<=n;j++)
      {
         t=b[j]-a[j]+k;
         if (t<0)
         {
            t=(t+grunntall)%grunntall;
            k=-1;
            c[j]=t;
         }
         else
         {
            k=0;
            c[j]=t%grunntall;
         };
      }
      c[0]=-1; Svaret blir negativt
   }
   else
Hvis a ≥ b trekker vi b fra a
   {
      k=0;

      for (j=1;j<=n;j++)
      {
         t=a[j]-b[j]+k;
         if (t<0)
         {
            t=(t+grunntall)%grunntall;
            k=-1;
            c[j]=t;
         }
         else
         {
            k=0;
            c[j]=t%grunntall;
         };
      };
      c[0]=1; Svaret blir positivt
   }
}
```

```
void leggtil(x,y,n)
Legger x med lengde n til y
int *x, *y,n;
{
   int mente,t,i,max;

   mente=0;
   for (i=1; i<=n; i++)
   {
      t=x[i]+y[i]+mente;
      y[i]=t%grunntall;
      mente=t/grunntall;
   }
y kan ha lengde > n
   while (mente>0)
   {
      t=y[i]+mente;
      y[i]=t%grunntall;
      mente=t/grunntall;
      i++;
   }
}
```

```
void splittoghersk(tall1,tall2,svar,n)
Selve multiplikasjonsprosedyra: svar=tall1·tall2,
tall1, tall2 har lengde n=2^k
int *tall1,*tall2,*svar,n;
{
int
a1[ARRAYLENGDE / 2],
a2[ARRAYLENGDE / 2],
b1[ARRAYLENGDE / 2],
b2[ARRAYLENGDE / 2],
svar1[ARRAYLENGDE],
svar2[ARRAYLENGDE],
svar3[ARRAYLENGDE],
a1a2[ARRAYLENGDE],
b2b1[ARRAYLENGDE],
s,i,tegn;
long t1,t2,t3;

Nullstille svar array
   for (i=1;i<=2*n;i++) svar[i]=0;
   svar[0]=1;
   tegn=tall1[0]*tall2[0]; Tegnet til svaret
   s=n/2;
   if (s>0)
   {
Oppdeling av tall
      for (i=1;i<=s;i++) a2[i]=tall1[i];
      for (i=s+1;i<=n;i++) a1[i-s]=tall1[i];
      for (i=1;i<=s;i++) b2[i]=tall2[i];
      for (i=s+1;i<=n;i++) b1[i-s]=tall2[i];

Alle fortegn settes positiv
      a1[0]=a2[0]=b1[0]=b2[0]=1;

Rekursivt kall 1 - Beregner a1·b1
      splittoghersk(a1,b1,svar1,s);

Rekursivt kall 2 - Beregner a2·b2
      splittoghersk(a2,b2,svar2,s);
      leggtil(svar2,svar,n);

Beregner a2·b2·grunntall^n
      shift(svar2,s,n);
      leggtil(svar2,svar,s+n);
Beregner a1·b1·grunntall^n
      shift(svar1,s,n);
      leggtil(svar1,svar,s+n);
Beregner a1·b1·grunntall^2n
      shift(svar1,s,s+n);
      leggtil(svar1,svar,2*n);

Rekursivt kall 3 - Beregner (a1-a2)(b2-b1)
      trekkfra(a1,a2,a1a2,s);
      trekkfra(b2,b1,b2b1,s);
      splittoghersk(a1a2,b2b1,svar3,s);

Beregner (a1-a2)(b2-b1)·grunntall^n
         shift(svar3,s,n);

Hvis negativt- trekk ifra ellers legg til
         if(svar3[0]<0)
            trekkfra2(svar,svar3,s+n);
         else
            leggtil(svar3,svar,s+n);

         svar[0]*=tegn;  Setter fortegn
   }
   else
Tallene har bare et siffer
   {
      t1=tall1[1]; t2=tall2[1];
Multiplikasjon på grunn-nivå
      svar[1]=(t1*t2)%grunntall;
      svar[2]=(t1*t2)/grunntall;
      svar[0]=tegn;
   }
}
```

Tidsforbruk

kjørt på Mac IIx med grunntall 10^4

Antall siffer(desimale)	Sekunder
64	0.03
128	0.11
256	0.38
512	1.13
1024	3.45
2048	10.43

3.5.4 Analyse av C-programmet

Denne versjonen av Splitt og Hersk øker lengden til tallene slik at den blir en potens av 2. Deretter deler vi tallene i to deler, og får 3 multiplikasjoner med bare halvparten så lange tall, pluss noen addisjoner, subtraksjoner og shiftinger. Har tallene etter de er delt i to en lengde som er > 1, kalles samme prosedyra rekursivt opp og tallene blir på nytt delt i to.

Multiplikasjoner får vi bare når tallene har lengde 1, og det blir 4 multiplikasjoner for hvert tallpar, tilsammen $4 \cdot 3^{\log_2 n}$ (2 multiplikasjoner, 1 div og 1 mod).

I tillegg har vi addisjoner og shiftinger m.m. når lengden er > 1. Tiden dette tar er (ca.):

Oppretting av variabler	n	
Nullstilling av svar-array	n	(Tildeling av verdier tar mindre tid)
Oppdeling	$2 \cdot \frac{n}{2}$	(Tildeling av verdier tar mindre tid)
Addisjon	n	
Shift	$n + \frac{n}{2}$	
Addisjon	$n + \frac{n}{2}$	
Shift	$n + \frac{n}{2}$	
Addisjon	$n + \frac{n}{2}$	
Shift	$2n$	
Addisjon	$2n$	
Subtraksjon	$2 \cdot \frac{n}{2}$	
Shift	$n + \frac{n}{2}$	
Add/Sub	$n + \frac{n}{2}$	
Tilsammen	$18n$	

Tiden til parameteroverføring og opprettingen av variabler ved prosedyrekallene er grovt estimert. Tiden dette tar er (forhåpentligvis) kort. Hvor mange operasjoner blir dette da?
La $m = \log_2 n$. Vi får:

$$18n + 3 \cdot 18\frac{n}{2} + 3^2 \cdot 18\frac{n}{2^2} + \dots + 3^{m-1} \cdot 18 \cdot \frac{n}{2^{m-1}} = \sum_{i=0}^{m-1} \left(3^i \cdot 18 \cdot \frac{n}{2^i}\right) = 18n \sum_{i=0}^{m-1} \left(\frac{3}{2}\right)^i =$$

$$18n \frac{1 - \left(\frac{3}{2}\right)^m}{1 - \left(\frac{3}{2}\right)} = 18n \left(2\left(\frac{3}{2}\right)^{\log_2 n} - 2\right) = 18n \left(2n^{\log_2(3/2)} - 2\right) = 36 \cdot n^{\log_2 3} + 36\text{n}.$$

Det viser seg ved nummeriske analyser at $36 \cdot n^{\log_2 3}$ er ei bedre tilnærming til $18n \sum_{i=0}^{m-1} \left(\frac{3}{2}\right)^i$ enn $36 \cdot n^{\log_2 3} + 36\text{n}$.

I tillegg kommer $4 \cdot 3^{\log_2 n}$ operasjoner fra tidligere.

Totalt antall operasjoner blir da $36 \cdot n^{\log_2 3} + 4 \cdot n^{\log_2 3}$, som er asymptotisk lik $40 \cdot n^{\log_2 3}$ når $n \to \infty$.
Vi har dermed fått et grovt estimat på konstanten som skjuler seg bak O-notasjonen. Bruker vi samme analyseteknikk for skolemultiplikasjon får vi ca. $6n^2$. Setter vi disse lik hverandre får vi skjæring rundt 100 siffer, noe som stemmer brukbart med måling av tidsforbruket. Jeg gjør oppmerksom på at konstantene kun kan brukes som sammenligning mellom algoritmer som er analysert på samme måten.
Brassard, Bratley antyder at splitt og hersk metoden er en forbedring i forhold til skolealoritmen for tall i underkant av 100 siffer [22]. Aho, Hopcraft,Ullman antyder 500 siffer, noe som virker vel høyt [23]. Det er vel desimalsifre de mener.

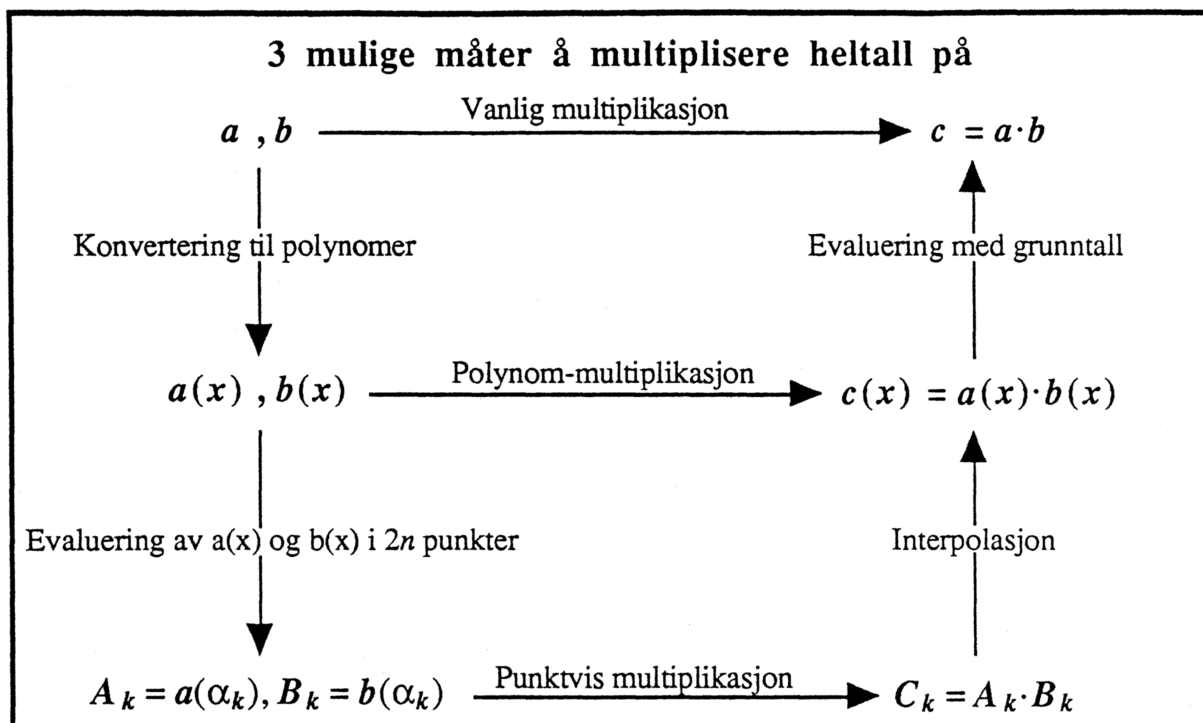
3.6 FFT-Rask Fourier Transformasjon [24]

3.6.1 Motivasjon

Multiplikasjon ved bruk av rask Fourier transformasjon er egentlig en multiplikasjonsmetode for polynomer, men siden heltall og polynomer er nært knyttet, kan vi med små modifikasjoner bruke den på heltall også.
En metode for å multiplisere to n-sifrete heltall $a = (a_{n-1} \ldots a_1 a_0)_G$ og $b = (b_{n-1} \ldots b_1 b_0)_G$ er å bruke skolealgoritmen for multiplikasjon. En annen metode er å konvertere a og b til to polynomer $a(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ og $b(x) = b_{n-1}x^{n-1} + \ldots + b_1 x + b_0$, multiplisere disse på vanlig måte til $c(x)$, og så evaluere $c(x)$ i $x = G$. Begge disse metodene er som kjent $O(n^2)$. En tredje metode er å evaluere polynomene $a(x)$ og $b(x)$ i minst $2n - 1$ forskjellige verdier α, dvs $A_k = a(\alpha_k)$ og $B_k = b(\alpha_k)$. Grunnen til at vi trenger 2n – 1 verdier, er at svaret $c(x)$ har grad $2n - 2$, dvs. $2n - 1$ ledd. Punktene vi nå får kan vi multiplisere, dvs. vi multipliserer punktene A_k og B_k som er evaluert med samme verdi α. Vi får $2n - 1$ punkter $C_k = A_k \cdot B_k$, som vi kan bruke interpolasjon på og få svaret $c(x) = a(x) \cdot b(x)$. For å komme fram til $c = a \cdot b$ evaluerer vi $c(x)$ i $x = G$.
Evalueringen av polynomene er $O(n^2)$, den punktvise multiplikasjonen er $O(n)$ og interpolasjonen er $O(n^2)$, så hva er egentlig vitsen? Alt vi har oppnådd er en vanskelig metode med samme asymptotiske kompleksitet, men med en konstant foran n^2 som er mye større enn med skolemetoden. Men idéen er god og kan forbedres. Et av poengene er, som vi skal se, å velge verdiene der polynomene $a(x)$ og $b(x)$ skal evalueres på en "lur" måte!!

---

[22] Algoritmics – Brassard, Bratley s.14
[23] Data Stuktures and Algorithms – Aho,Hopcroft,Ullman s.309
[24] The Design and Analysis of Computer Algorithms - Aho/Hopcroft/Ullman s.252
og forelesninger vår 89 – Loren Olson



3.6.2 Innledende definisjoner

La **K** være en kropp. Et element ω er en **primitiv** n**-te enhetsrot** i **K** hvis:

$\omega \neq 1$, $\omega^n = 1$, $\omega^i \neq 1$ for $1 \leq i < n$ og $\sum_{i=0}^{n-1} \omega^{ip} = 0$ for $1 \leq p < n$.

Elementene $\omega^0, \omega^1, \ldots, \omega^{n-1}$ er alle n-te enhetsrøtter i **K**.

La A være en $n \times n$ matrise gitt ved $A = \{\omega^{ij}\}_{i,j=0}^{n-1}$.

La oss si at vi er gitt n elementer fra K: $\{a_{n-1}, a_{n-2}, \ldots, a_0\}$.

Av disse kan vi lage en kolonnevektor $\vec{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$.

Den **Diskrete Fouriertransformasjonen** til $\vec{a}$ er vektoren

$F(\vec{a}) = A\vec{a} = \vec{b} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}$ der $b_i = \sum_{k=0}^{n-1} a_k \omega^{ik}$.

Lemma La $\omega \geq 1$ være en primitiv n-te enhetsrot. La $A = \{\omega^{ij}\}_{i,j=0}^{n-1}$.

Da eksisterer A^{-1} og $A^{-1} = \left\{\frac{1}{n}\omega^{-ij}\right\}_{i,j=0}^{n-1}$

Bevis: Vi beregner $(\omega^{ij})\cdot\left(\left(\frac{1}{n}\right)\omega^{-ij}\right)$. Den (i,j)-te koeffisienten er $\frac{1}{n}\sum_{k=0}^{n-1} \omega^{ik}\omega^{-kj}$.

Hvis $i = j$ er denne $= \frac{1}{n}\sum_{k=0}^{n-1} 1 = 1$. Anta $i \neq j$. Da får vi $\frac{1}{n}\sum_{k=0}^{n-1} \left(\omega^{i-j}\right)^k = 0$ og $AA^{-1} = I = A^{-1}A$.

$F^{-1}(\vec{b}) = A^{-1}\vec{b}$. Den i-te komponenten, $0 \leq i < n$, til $A^{-1}\vec{b}$ er $\frac{1}{n}\sum_{k=0}^{n-1} b_k\omega^{-ik}$.

$F^{-1}(\vec{b})$ = den **inverse diskrete Fouriertransformasjonen** til $\vec{b}$.

Det er klart at $F^{-1}(F(\vec{a})) = \vec{a} = F(F^{-1}(\vec{a}))$.

3.6.3 Bruk av Fouriertransformasjon til polynommultiplikasjon

Det er et nært forhold forhold mellom F, F^{-1}, koeffisientene til et polynom og evaluering av polynomet.

Hvis vi har et polynom $a(x) = \sum_{i=0}^{n-1} a_i x^i$ av grad $< n$, så er det entydig bestemt hvis vi enten:

1. Oppgir koeffisientene $a_0, a_1, \ldots, a_{n-1}$, eller
2. Verdien til $a(x)$ i n forskjellige punkter $\{x_0, x_1, \ldots, x_{n-1}\}$

Oppgaven med å beregne a_i-ene gitt $a(x_i)$ kalles for interpolasjon.

Til $\vec{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$ kan vi tilordne polynomet $a(x) = \sum_{i=0}^{n-1} a_i x^i$.

$a(\omega^j) = \sum_{i=0}^{n-1} a_i\omega^{ij} = b_j$ er den j-te koeffisienten til $F(\vec{a})$. Anvender vi den inverse Fouriertransformasjonen $F^{-1}(F(\vec{a}))$ på b_j-ene, får vi igjen koeffisientene til $a(x)$

La $\vec{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$ og $\vec{b} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}$. **Foldingen** (eller konvolusjonen) til $\vec{a}$ og $\vec{b}$ er

$\vec{a} * \vec{b} = \vec{c} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{2n-1} \end{bmatrix}$ der $c_i = \sum_{j=0}^{n-1} a_j b_{i-j}$ (og der $a_k = b_k = 0$ for $k < 0$ og $k \geq n$).

Ser vi på $\vec{a}$ og $\vec{b}$ som polynomer istedet for vektorer, er foldingen det samme som vanlig polynommultiplikasjon.

Eksempel

$$
\begin{aligned}
c_0 &= a_0 b_0 \\
c_1 &= a_0 b_1 + a_1 b_0 \\
c_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\
&\vdots \\
c_{2n-1} &= 0
\end{aligned}
$$

La $a(x) = \sum_{i=0}^{n-1} a_i x^i$ og $b(x) = \sum_{j=0}^{n-1} b_j x^j$. $\quad a(x) \cdot b(x) = \sum_{i=0}^{2n-1} \left[\sum_{j=0}^{i} a_j b_{i-j} x^j \right] \cdot x^i = \sum_{i=0}^{2n-2} c_i x^i$

der c_i-ene er koeffisientene til $\vec{a} * \vec{b}$.

Hvis $a(x)$ og $b(x)$ er representert ved sine koeffisienter, så kan vi finne koeffisientene til produktet ved å folde koeffisientvektorene til $a(x)$ og $b(x)$. Er derimot $a(x)$ og $b(x)$ representert ved verdiene i n punkter α, så får vi verdirepresentasjonen til produktet $c(x)$ ved å gange sammen verdiene til $a(\alpha_k)$ og $b(\alpha_k)$ i samme punkt α_k. Dette gjelder for alle α_k.

Det ser ut som at foldinga av to vektorer $\vec{a}$ og $\vec{b}$ er det samme som den inverse fouriertransformasjonen til det punktvise produktet til fouriertransformasjonen av $\vec{a}$ og $\vec{b}$, eller at $\vec{a} * \vec{b} = F^{-1}(F(\vec{a}) \cdot F(\vec{b}))$.

Vi har imidlertid et problem. Verdirepresentasjonen til produktet mellom $a(x)$ og $b(x)$ krever verdier i minst $2n - 1$ punkter for å kunne representeres entydig. Grunnen til dette er at produktet har grad $2n - 2$. Dette løser vi ved å betrakte $a(x)$ og $b(x)$ som polynomer med lengde $2n$, dvs. vi fyller inn nullere som koeffisienter i alle ledd av grad i, $n \le i < 2n$.

Definisjon (punktvis multipliasjon) $\begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-1} \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix} = \begin{bmatrix} u_0 v_0 \\ u_1 v_1 \\ \vdots \\ u_{n-1} v_{n-1} \end{bmatrix}$

Foldingsteoremet: La $\vec{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ og $\vec{b} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ ha lengde 2n. Da er $\vec{a} * \vec{b} = F^{-1}(F(\vec{a}) \cdot F(\vec{b}))$.

Bevis: La $F(\vec{a}) = \begin{bmatrix} a'_0 \\ a'_1 \\ \vdots \\ a'_{2n-1} \end{bmatrix}$ og $F(\vec{b}) = \begin{bmatrix} b'_0 \\ b'_1 \\ \vdots \\ b'_{2n-1} \end{bmatrix}$

Siden $a_i = b_i = 0$ for $i \ge n$, så er $a'_l = \sum_{j=0}^{n-1} a_j \omega^{lj}$ og $b'_l = \sum_{k=0}^{n-1} a_k \omega^{lk}$ for $0 \le l \le 2n$.

Den l-te koeffisienten i $F(\vec{a}) \cdot F(\vec{b})$ er $a'_l b'_l = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} a_j b_k \omega^{l(j+k)}$

$$\vec{a} * \vec{b} = \vec{c} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{2n-1} \end{bmatrix} \text{ og } F(\vec{a} * \vec{b}) = \begin{bmatrix} c_0' \\ c_1' \\ \vdots \\ c_{2n-1}' \end{bmatrix}$$

$$c_p = \sum_{j=0}^{2n-1} a_j b_{p-j} \quad c_l' = \sum_{p=0}^{2n-1} c_p \omega^{lp} = \sum_{p=0}^{2n-1} \sum_{j=0}^{2n-1} a_j b_{p-j} \omega^{lp} \text{ (Sett } k = p - j)$$

$$= \sum_{j=0}^{2n-1} \sum_{k=-j}^{2n-1-j} a_j b_k \omega^{l(j+k)} = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} a_j b_k \omega^{l(j+k)} = a_i' \cdot b_i' \text{ siden } b_k = 0 \text{ for } k \geq n.$$

Dermed er $F(\vec{a} * \vec{b}) = F(\vec{a}) \cdot F(\vec{b})$. Vi kan nå anvende F^{-1} og få $\vec{a} * \vec{b} = F^{-1}(F(\vec{a}) \cdot F(\vec{b}))$.

3.6.4 En $O(n \cdot \log_2 n)$ FFT algoritme

Ser vi på kompleksiteten til F, så kan vi beregne $F(\vec{a})$ med $O(n^2)$ operasjoner dersom vi opererer direkte med definisjonen av den diskrete Fouriertransformasjonen. Som nevnt i innledningen er ikke dette en forbedring i forhold til skolealgoritmen, tvert imot.
Spørsmålet er nå om dette kan forbedres?
La ω være ei primitiv n-te enhetsrot.

Å beregne $F(\vec{a}) = A\vec{a}$ er det samme som å beregne $a(\omega^i)$-ene der $a(x) = \sum_{j=0}^{n-1} a_j x^j$. Anta at $2 \mid n$ og $n = 2r$.

La $Y = x^2$ og skriv $a(x) = \sum_{j=0}^{r-1} a_{2j} x^{2j} + x \cdot \sum_{j=0}^{r-1} a_{2j+1} x^{2j} = \sum_{j=0}^{r-1} a_{2j} Y^j + x \cdot \sum_{j=0}^{r-1} a_{2j+1} Y^j =$
$b(Y) + x \cdot c(Y)$ der $b(Y) = \sum_{j=0}^{r-1} a_{2j} Y^j$ og $c(Y) = \sum_{j=0}^{r-1} a_{2j+1} Y^j$.

Sett $\alpha_i = (\omega^i)^2 = \omega^{2i}$.
Vi får (*) $a(\omega^i) = b(\alpha_i) + \omega^i \cdot c(\alpha_i)$.
Dermed har vi redusert problemet til å evaluere to polynomer b og c av grad $r - 1$ i punktene $\alpha_i = (\omega^i)^2 = \omega^{2i}$, $0 \leq i < r$, der α er en primitiv r-te enhetsrot.
Merk også at vi får punktene $a(\omega^{i+r}) = b(\alpha_i) - \omega^i \cdot c(\alpha_i)$ nesten gratis siden $\omega^{i+r} = -\omega^i$, for $0 \leq i < r$.
Vi har nå oppnådd å få to polynomer med grad halvparten av det opprinnelige. Arbeidet er følgelig redusert til $T(n) = 2T(n/2) + cn$, der $T(n)$ er tidsforbruket for n punkter og c en konstant. Dette er en liten forbedring, men vi kan gjøre det enda bedre enn dette!!
Anta at $n = 2^k$ og at ω er en primitiv n-te enhetsrot. Da kan vi fortsette å dele opp polynomene videre på samme måte, slik at vi får en rekursiv algoritme.
En av grunnene til at denne metoden virker er at vi velger de n punktene som n-te enhetsrøtter der n er en potens av to. De n n-te enhetsrøttene som vi starter med sørger for at vi til enhver tid har det nøyaktige antall "riktige" punkter å evaluere polynomene med nedover i rekursjonen.

Et eksempel med $n = 8$ og 8.enhetsrot ω. Vi får 8 punkter: $\omega^0,\omega^1,\omega^2,\omega^3,\omega^4,\omega^5,\omega^6,\omega^7$.
Så deles polynomet i to og evalueres med de 4 punktene: $\alpha^0,\alpha^1,\alpha^2,\alpha^3$ der $\alpha = \omega^2$. For å gjøre dette deles de halve polynomene i to og evalueres med de 2 punktene: β^0,β^1 der $\beta = \alpha^2 = \omega^4$. Så deles fjerdedels polynomet vårt opp i to deler og evalueres i punktet $1 = \beta^2 = \alpha^4 = \omega^8$. Dette siste er ikke nødvendig siden vi nå har polynomer som kun består av konstantleddet.

3.6.5 Analyse av algoritmen

La $T(n)$ være tidsforbruket for n punkter. $T(n) = 2T(n/2) + c \cdot n$, der c er en konstant og $c \cdot n$ angir tiden for til å beregne α_i-ene og (*)
Setter vi $n = 2^k$, så har vi

$$\begin{aligned}
T(n) &= T(2^k) \\
&= 2T \cdot (2^{k-1}) + c2^k \\
&= 2[2 \cdot T(2^{k-2}) + c2^{k-1}] + c2^k \\
&= 2^2 \cdot T(2^{k-2}) + c2^k + c2^k \\
&= 2^3 \cdot T(2^{k-3}) + 3c2^k \\
&= 2^k \cdot T(1) + k \cdot c \cdot 2^k \\
&= n \cdot T(1) + c \cdot n \cdot \log_2 n \\
&= O(n \cdot \log_2 n)
\end{aligned}$$

Plassforbruket er $O(n)$. Opprinnelig har tallet (polynomet) lengde n. Så deles det opp til det halve osv. Vi får $n + n/2 + n/4...$ som blir $2n$ eller $O(n)$ når n går mot uendelig.

3.6.6 FFT – Et program-eksempel i (Kvasi) Pascal [25]

Inndata: Et heltall $n = 2^m$

Et polynom $a(x) = \sum_{i=0}^{n-1} a_i x^i$

En primitiv n-te enhetsrot ω

Utdata: En array $\mathbf{A} = \{A_0, A_1, \ldots, A_{n-1}\}$ der $A_k = a(\omega^k)$

procedure FFT (n , $a(x)$, ω , $\mathbf{A}$)
if $n = 1$ **then** $A_0 = a_0$
else
begin

{*Binær splitting*}

$r = n/2$

$b(x) = \sum_{i=0}^{r-1} a_{2i} x^i$

$c(x) = \sum_{i=0}^{r-1} a_{2i+1} x^i$

{*Rekursive kall*}

FFT (r , $b(x)$, ω^2, $\mathbf{B}$)

FFT (r , c(x) , ω^2, $\mathbf{C}$)

{*Sette sammen*}

for $k = 0$ **to** $r - 1$ **do**

begin

$A_k = B_k + \omega^k \cdot C_k$

$A_{k+n} = B_k - \omega^k \cdot C_k$

end

end

På samme måte kan den inverse Fouriertransformasjonen gjøres rask. Ved å betrakte punktene $\{A_0, A_1, \ldots, A_{n-1}\}$ vi fikk i FFT, som koeffisienter i et polynom

$$\mathbf{A}(x) = \sum_{i=0}^{n-1} A_i x^i$$

kan vi kalle opp FFT prosedyren med primitiv n-te enhetsrot ω^{-1} og så multiplisere hver koeffisient med n^{-1}.

Dette er det samme som definisjonen av den inverse Fouriertransformasjonen.

[25] Elements of Algebra and Algebraic Computing – John D Lipson s.297

3.6.7 FFI Rask Fourier Interpolasjon – Et program-eksempel [26]

Forslag til algoritme

Inndata: Et heltall $n = 2^m$

En primitiv n-te enhetsrot ω

punkter $\mathbf{A} = \{A_0, A_1, \ldots, A_{n-1}\}$

Utdata: Et polynom $a(x) = \sum_{i=0}^{n-1} a_i x^i$, der $a(\omega^k) = A_k$, $0 \le k < n$

procedure FFI (n , $\mathbf{A}$, ω , $a(x)$)
begin

$$\mathbf{A}(x) = \sum_{i=0}^{n-1} A_i x^i$$

FFT (n , $\mathbf{A}(x)$, ω^{-1}, $\mathbf{C}$)

$$a(x) = \sum_{i=0}^{n-1} \left(n^{-1} C_i\right) x^i$$

end

Vi kan dermed multiplisere to polynomer av grad n i $O(n \cdot \log_2 n)$ operasjoner, siden FFT har kompleksitet $O(n \cdot \log_2 n)$. Dette gjøres en gang for hvert polynom. Den punktvise multiplikasjonen er $O(n)$ og FFI er $O(n \cdot \log_2 n)$. Dvs. $3 \cdot O(n \cdot \log_2 n) = O(n \cdot \log_2 n)$.

3.7 En Iterativ versjon av FFT[27]

3.7.1 Beskrivelse av algoritmen

I praksis bruker man en iterativ versjon av FFT, siden rekursjon er både tids og plasskrevende. Gitt et $(n - 1)$-te grads polynom $a(x)$. La ω være en primitiv n-te enhetsrot og anta at $n = 2^k$. Vi ønsker å beregne punktene $a(\omega^i)$ dvs. $A_i = a(\omega^i)$ for $0 \le i < n$.

Å evaluere et polynom a(x) i punktet ω^i er det samme som å dele $a(x)$ med $(x - \omega^i)$ og se på resten r. Dette siden $a(\omega^i) = (\omega^i - \omega^i) \cdot q(x) + r = r$

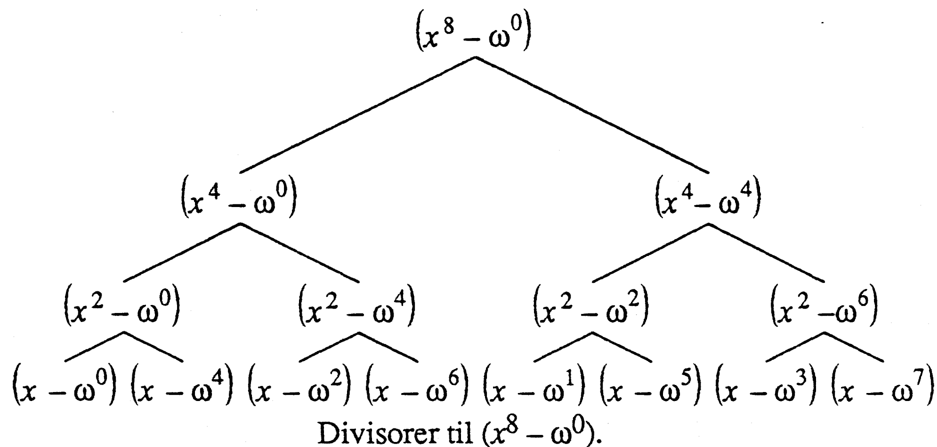
Hvis vi gjør vi dette for alle ω^i, $0 \le i < n$, vil vi igjen bruke $O(n^2)$ operasjoner. Spørsmålet er nå om vi kan forbedre idéen med å dele polynomet $a(x)$ med at eller annet polynom som involverer den primitive n-te enhetsrota ω, for så å se på resten $r(x)$?

Et eksempel: La $n = 8$ og ω være ei primitiv 8. enhetsrot. Se på produktet av lineære faktorer: $(x - \omega^0) \cdot (x - \omega^1) \cdot \ldots \cdot (x - \omega^7) = x^8 - \omega^0$. Alle ledd med eksponenter utenom 0 og 8 kan strykes siden $\omega^8 = 1$. På samme måte vil $(x - \omega^0) \cdot (x - \omega^2) \cdot (x - \omega^4) \cdot (x - \omega^6)$ være lik $(x^4 - \omega^0)$ og $(x - \omega^1) \cdot (x - \omega^3) \cdot (x - \omega^5) \cdot (x - \omega^7)$ bli $(x^4 - \omega^4)$.

---

[26] Elements of Algebra and Algebraic Computing – John D Lipson s.303
[27] Fundamentals of Computer Algorithms. Horowitz/Sahni s.436

Ved å fortsette på samme måten kan vi splitte opp $(x^8 - \omega^0)$ i linære faktorer slik som på figuren under.



Divisorer til $(x^8 - \omega^0)$.

La oss si at vi skal beregne Fouriertransformasjonen til 7. gradspolynomet $a(x)$ i punktene ω^i, $0 \le i \le 7$.

Vi begynner med å dele $a(x)$ med $d(x) = (x - \omega^0)\cdot(x - \omega^1)\cdot \ldots \cdot(x - \omega^7)$. Hvis $a(x) = q(x)\cdot d(x) + r(x)$, så er $a(\omega^i) = r(\omega^i)$ for $0 \le i \le 7$ siden $d(\omega^i) = 0$. Graden til $r(x)$ er mindre enn graden til d(x) som er 8. Siden $a(x)$ har grad < 8 vil $r(x)$ = a(x), og denne første divisjonen kan derfor sløyfes.

Nå kan vi videre dele $r(x)$ med $(x^4 - \omega^0)$ slik at vi får $s(x)$, og dele $r(x)$ med $(x^4 - \omega^4)$ slik at vi får $t(x)$. $a(\omega^i) = r(\omega^i) = s(\omega^i)$ for $i = 0, 2, 4, 6$, og $a(\omega^i) = r(\omega^i) = t(\omega^i)$ for $i = 1, 3, 5, 7$. Graden til $s(x)$ og $t(x)$ er mindre enn 4.

Deretter deler vi $s(x)$ med $(x^2 - \omega^0)$ og $(x^2 - \omega^4)$, slik at vi får rester u(x) og v(x), der $a(\omega^i) = u(\omega^i)$ for $i = 0, 4$, og $a(\omega^j) = v(\omega^j)$ for $i = 2, 6$.

Hver av divisorene har bare 2 ledd $\neq 0$, slik at alle divisjonen går raskt. Ved å fortsette på denne måten vil vi tilslutt ha beregnet alle 8 verdiene $a(x)$ mod $(x - \omega^j)$ for $i = 0, 1, ..., 7$.

3.7.2 Et program-eksempel i (Kvasi) Pascal på Iterativ FFT [28]

Denne algoritmen beregner Fouriertransformasjonen på samme måte som beskrevet over. Ved å utføre divisjonene nedover binærtreet slik som på figuren, vil vi tilslutt ende opp med punktene $A_k = a(\omega^k)$. Rekkefølgen til disse punktene vil bli permutert på samme måte som $(x - \omega^i)$-ene på bunnen av binærtreet. Dette kan vi lett rette på først i algoritmen, ved å permutere koeffisientene på motsatt måte.

Inndata: Et heltall $m = \log_2 n$

Et polynom $a(x) = \sum_{i=0}^{n-1} a_i x^i$

Utdata: array $\mathbf{A} = \{A_0, A_1, \ldots, A_{n-1}\}$ der $A_k = a(\omega^k)$. ω er ei primitiv n-te enhetsrot i kroppen $\mathbf{F}_p$ der p er et **Fourierprimtall**, dvs. på formen $2^i s + 1$ der s er et odde heltall. Mer om dette senere i oppgaven.

[28] Fundamentals of Computer Algorithms. Horowitz/Sahni s.438

```
procedure IkkeRekursivFFT( {a_0, a_1, ..., a_{n-1}} , m )
integer i, j, k, nivå, m, n, ndiv2, pow2, pow2div2, index, r, s, t;
begin
    n = 2^m;
    ndiv2 = n/2;
    j = 1;

>>> Permutering av koeffisientene til a(x) <<<
    for i = 1 to n – 1 do
    begin
        if i < j then
        begin
            t = a[j]; a[j] = a[i]; a[i] = t;
        end;
        k = ndiv2;
        while k < j do
        begin
            j = j – k;
            k = k/2;
        end;
        j = j + k;
    end;

>>> Selve Fouriertransformasjonen <<<
    for nivå = 1 to m do  >>> nivået i binærtreet (rota = 0) <<<
    begin
        pow2 = 2^nivå;
        pow2div2 =pow2/2;
        r = 1;
        s = ω^(2^nivå);
        for j = 1 to pow2div2 do
        begin
            for i = j to n step pow2 do
            begin
                index = i + pow2div2;

                >>> Her foregår divisjonene <<<
                t = ( a[index] * r ) mod Fourierprim;
                a[index] = ( a[i] – t ) mod Fourierprim;
                if ( a[index] < 0) then a[index] = a[index] + Fourierprim;
                a[i] = ( a[i] + t ) mod Fourierprim;
            end;
            r = ( r * s ) mod Fourierprim;  >>> Beregner riktig potens av ω <<<
        end;
    end;
end.
```

3.7.3 Analyse av algoritmen og pascalprosedyra

Siden polynomene på hvert nivå har en enkel form er divisjonene raske, og beregningstiden for koeffisientene er $O(n \cdot \log_2 n)$. Dette er lett å se siden treet har $\log_2 n$ nivåer med $2^{nivå}$ noder på hvert nivå (rota er nivå 0). Polynomet som deles på et gitt nivå har maksimalt $2^{m-nivå}$ ledd, der $n = 2^m$. Tiden for hvert nivå er dermed proposjonal med $2^{nivå} \cdot 2^{m-nivå} = 2^m = n$, og vi får $O(n \cdot \log_2 n)$.

Pascalprosedyra begynner med å permutere koeffisientene til $a(x)$, slik at de står på rett plass når prosedyra er ferdig. Variablen a inneholder selve polynomet, mens $m = \log_2$(polynomets lengde). Anta at $n = 2^m$ og se på de 3 for-løkkene som er inne i hverandre. Den innerste krever ikke mer enn konstant tid per iterasjon, og den utføres ikke mer enn $n/2^{nivå} < 2^{m-nivå+1}$ ganger. Dette impliserer at den totale kjøretiden er begrenset av

$$\sum_{nivå=1}^{m} \sum_{j=1}^{2^{nivå-1}} c \cdot 2^{m-nivå+1} = \sum_{nivå=1}^{m} c \cdot 2^m = c \cdot 2^m \cdot m = O(n \cdot \log_2 n).$$

3.8 : FFT multiplikasjon av heltall $\leq 10^{75.497.472}$ [29]

3.8.1 FFT multiplikasjon av heltall

Ja, så var vi framme med vårt mål med Fouriertransformasjonene, nemlig multiplikasjon av heltall. La $a = (a_{n-1} \ldots a_1 a_0)_G$ og $b = (b_{n-1} \ldots b_1 b_0)_G$ være to (store) heltall med $n = 2^m$ og grunntall G. For tall der lengden ikke er en potens av to, fyller vi på med 0-ere slik at lengden blir en potens av to. Så dobles lengden ved å på nytt å fylle på med 0-ere, slik at tallene får samme lengde som svaret dvs. $2n$. Denne operasjonen vil kun øke kjøretiden med en konstant.

Til a og b kan vi så tilordne polynomene $a(x) = \sum_{i=0}^{2n-1} a_i x^i$ og $b(x) = \sum_{i=0}^{2n-1} b_i x^i$.

Vi velger grunntallet $G <$ ordlengden W til datamaskinen. Det greieste er å velge G som største potens av $10 < W$ (gitt at datamaskinen har dobbel presisjon på heltallsaritmetikk).

Vi har at $a \cdot b = c$, $a(x) \cdot b(x) = c(x)$, $a = a(G)$, $b = b(G)$ og $a \cdot b = a(G) \cdot b(G) = c(G)$.

En koeffisient i $c(x)$, $c_k = \sum_{\substack{i+j=k \\ 0 \leq i,j < n}} a_i b_j$ er $< nG^2$.

Dette betyr at en koeffisient c_k kan være større enn ordlengden til maskinen. Løsningen på dette problemet er å beregne c_k-ene modulo K forskellige primtall, for til slutt å bruke CRT for å beregne c_k-ene nøyaktig. Siden vi også er avhengig av at kroppen vi regner i inneholder primitive $2n$-te enhetsrøtter, må primtallene være på en spesiell måte. Fra kroppteori vet vi at en kropp $\mathbf{F}_p$ inneholder ei primitiv $2n$-te enhetsrot dersom $2n \mid (p-1)$, dvs. at p må være på formen $2^e d + 1$, der d er et positivt odde heltall. Slike primtall kalles Fourierprimtall.

Vi velger nå K forskjellige Fourierprimtall: $p_1, \ldots, p_K$, slik at $G \leq p_i \leq W$.

---

[29] Elements of Algebra and Algebraic Computing – John D Lipson s.308

Hvor mange slike primtall trenger vi?
CRT beregner c_k riktig dersom $c_k < p_1 \cdot p_2 \cdot \ldots \cdot p_K$.
Siden $p_i \geq G$, vil c_k bli riktig beregnet dersom $nG^2 \leq G^K$, dvs $\log_G n + 2 \leq K$. Derfor er $K = 3$ tilstrekkelig for $n < G$.

Så beregner vi $c_{(i)}(x) = a(x) \cdot b(x) \pmod{p_i}$ ved hjelp av FFT, og bruker CRT for å beregne koeffisientene til $c(x)$ nøyaktig.

Siden $c(x)$ har lengde $2n$ må $2^e \geq 2n$ for å garantere at vi har ei primitiv $2n$-te enhetsrot. Hvis E er det største heltallet slik at det eksisterer 3 Fourierprimtall med eksponent e ≥ E, så må $2n \leq 2^E$. Dette holder for $n \leq 2^{E-1}$.

Teorem Hvis vi har en datamaskin med ordlengde W, så kan den multiplisere to heltall med n siffer i grunntall $G < W$ i $O(n \cdot \log_2 n)$ tid dersom:

1. $n \leq G$.
2. $n \leq 2^{E-1}$, gitt at det eksisterer 3 forskjellige Fourierprimtall $p_i = 2^{e_i} \cdot d_i + 1$ med $G \leq p_i \leq W$ der $e_i \geq E$

Vi har sett at å beregne $c_{(i)}(x)$ er $O(n \cdot \log_2 n)$. CRT er $O(n)$, siden vi bare regner modulo 3 forskjellige primtall og evalueringen $c = c(G)$ er $O(n)$.

Et eksempel med ordlengde $W = 2^{31} - 1$ (31 bit + fortegn)
Velger vi $G = 10^9$ må vi ha $n \leq 10^9$. Da eksisterer det 3 Fourierprimtall som følger

$p = 2^e d + 1$	e	α = minste primitive element i $\mathbf{Z}_p$
2013265921	27	31
2113929217	25	5
2130706433	24	3

Med grunntall 10^9 kan vi multiplisere heltall med $\leq 2^{23}$ siffer. Dette tilsvarer 75.497.472 desimale siffer. Det burde holde til de fleste praktiske formål!!

3.8.2 Et eksempel på FFT heltallsmultiplikasjon:

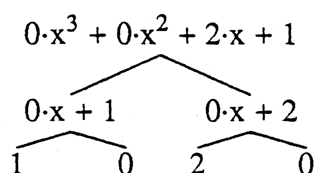
La oss regne over kroppen $\mathbf{Z}_{17}$. $\alpha = 3$ er et primitivt element i $\mathbf{Z}_{17}$ siden $\alpha^{(p-1)/q} \neq 1$ i $\mathbf{Z}_p$ for alle faktorer $q \mid p - 1$. $\omega = \alpha^{(p-1)/n}$ er da ei primitiv n-te enhetsrot i $\mathbf{Z}_p$. Siden $16 = 2^4 \mid p - 1$ kan vi multiplisere tall med opptil 8 siffer i $\mathbf{Z}_{17}$.

La oss multiplisere tallene $a = 21$ med $b = 32$ ved hjelp av Rask Fouriertransformasjon.

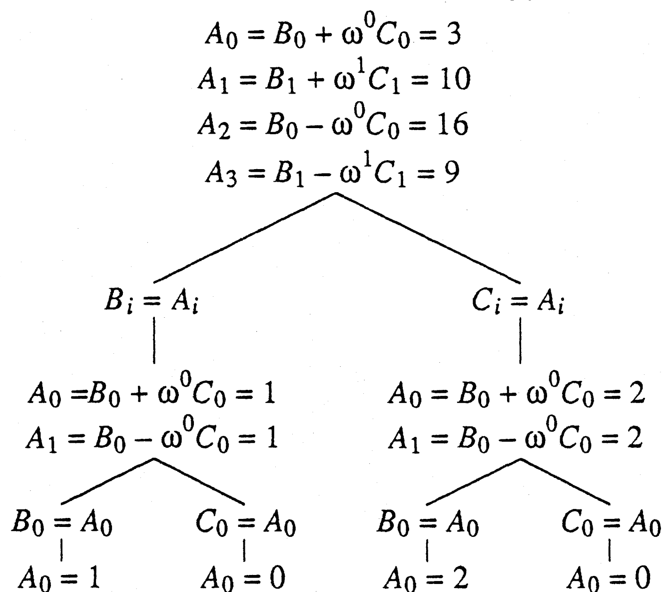
Først konverterer vi tallene om til polynomene $a(x) = 2x + 1$ og $b(x) = 3x + 2$. Siden svaret har lengde ≤ 4 tenger vi en primitiv 4. enhethetsrot. $\omega = 13$ oppfyller dette kravet.

Vi fyller på med nullere slik at lengden til $a(x)$ og $b(x)$ blir den samme som i svaret $c(x)$.

Så splitter vi $a(x)$ opp som på figuren under.



Når det er gjort begynner vi på bladnivå og jobber oss oppover.



Punktene $A_k = a(\omega^k)$ kommer ut i toppen (dvs. rota) av treet.

$A = \{9,16,10,3\}$. På samme måte blir

$B = \{14,16,7,5\}$, og punktmultiplikasjon gir oss $C = \{7,1,2,15\}$.

Så skal vi bruke FFT til interpolasjon. Vi trenger to verdier $n^{-1} = 13$ og $\omega^{-1} = 4$.

Vi ser nå på punktene C som koeffisientene til et polynom. Dette kan vi nå bruke FFT på.

Vi får $c_k = n^{-1} \cdot C((\omega^{-1})^k)$, der c_k er koeffisientene til polynomet $c(x) = a(x) \cdot b(x) = 6x^2 + 7x + 2$. Ved å evaluere $c(x)$ i $x = G = 10$ får vi $672 = 21 \cdot 32$.

3.8.3 Analyse av C-programmet

Analyse av C-programmet basert på rekursive FFT:

På grunn av at Macintosh IIcx er en 32-bits maskin uten dobbel presisjon for heltallsaritmetikk ble det brukt grunntall 10^4. Av den grunn ble de 3 Fourierprimtallene som følger:

$12289 = 3 \cdot 2^{12} + 1$

$18433 = 9 \cdot 2^{11} + 1$

$40961 = 5 \cdot 2^{13} + 1$

Dette fører følgelig til at antall siffer i tallene som skal multipliseres må være $\leq 2^{10}$.

Fra nå av ser vi på tallene som polynomer

Denne versjonen av FFT øker lengden til polynomene slik at den blir en potens av 2 (som beskrevet tidligere). Etter det dobles lengden slik at polynomene får samme lengden som svaret.

Vi skal nå se på FFT for et av disse polynomene.

Prosedyra FFT kalles opp og deler polynomet opp i to nye polynomer med bare halvparten så stor grad som det opprinnelige. Det ene nye polynomet får alle leddene med odde grad, mens det andre får alle med lik grad. Slik fortsetter vi rekursivt å dele opp polynomene helt til de har lengde 1. På vei oppover i rekursjonen settes delsvarene sammen til verdiene vi er på jakt etter.

Hvor mye arbeid utføres i prosedyra? La $N = 2n = 2^m$ = lengden til svaret.

Oppretting av variabler	N	operasjoner
Oppdeleing	N	operasjoner
Sette sammen resultatene	$4N$	operasjoner
Tilsammen	$6N$	operasjoner

Antall operasjoner blir da:

$$6N + 2 \cdot 6 \frac{N}{2} + 2^2 \cdot 6 \frac{N}{2^2} + \ldots + 2^{m-1} \cdot 6 \frac{N}{2^{m-1}} = \sum_{i=0}^{m-1} \left(2^i \cdot 6 \cdot \frac{N}{2^i} \right) = 6N \cdot \log_2 N =$$

$$12n \cdot \log_2(2n) = 12n \cdot \log_2 n + 12n.$$

Dette skal tilsammen utføres 9 ganger. En gang for hvert av polynomene, og en gang for punktene etter punktmultiplikasjonen (FFT Invers). Hver av disse må vi igjen beregne modulo 3 forskjellige primtall. Tilsammen får vi $108n \cdot \log_2 n + 108n$.

I tillegg til dette har vi:

Punktvis multiplikasjon	$6N$	operasjoner
Multiplikasjon med N-invers	$6N$	operasjoner
CRT	$29N$	operasjoner
Evaluering med grunntall	$15N$	operasjoner
Tilsammen	$56N$	operasjoner

Vi får da $108n \cdot \log_2 n + 108n + 56N$ eller $108n \cdot \log_2 n + 220n$. Det betyr at kjøretiden til C-programmet er begrenset av $O(n \cdot \log_2 n)$.
Konstantene er kun brukbar som sammenligning mellom algoritmer som er analysert på samme måte som FFT, dvs. Splitt og Hersk og skolemultiplikasjon. Selv her kan det være en viss unøyaktighet siden to av algoritmene er rekursive, og de forskjellige operasjonene tar forskjellig tid. Jeg har stort sett bare telt opp antall addisjoner og multiplikasjoner som foretas i de forskellige algoritmene.

Med konstantene vi har kommet frem til får vi følgende skjæringspunkter for kjøretidene til de 3 multiplikasjonsalgoritmene:

Skolealgoritmen – Splitt og Hersk	96 siffer
Skolealgoritmen – FFT	170 siffer
Splitt og Hersk – FFT	290 siffer

Analyseresultatene stemmer bra med måleresultatene på s.58

Dette er et godt eksempel på at det er viktig å få med seg den skjulte konstanten hvis man skal finne den "rette" algoritmen for å gitt problem.
Brukes grunntall 10^4 (10^9), så er skolealgoritmen best for tall opp til ca. 380 (850) desimal-siffer, deretter er Splitt og Hersk best opp til 1160 (2600) desimalsiffer og så overtar rekursiv FFT.
Egentlig er skolealgoritmen best for et enda større antall siffer, siden de andre algoritmene må fylle på med 0-ere for at lengden skal bli en potens av 2.
Det burde tilsi at skolealgoritmen er best til de aller fleste praktiske formål.

3.8.4 FFT multiplikasjon : Et program-eksempel i C

Prosedyra FFT tar imot 2 positive heltall med lengde som er en potens av 2. Den regner modulo alle primtallene i motsetning til FFT-Invers som bare regner modulo et primtall om gangen. Programmet er ikke komplett i den forstand at det må skrives inn nye enhetsrøtter hver gang vi skal endre på lengden til tallene som skal multipliseres. Ellers skulle det virke bra.

```
#include <stdio.h>
#include <math.h>
#define BASE 10000

>>>Her settes lengden på tallene med grunntall BASE
#define ARRAYLENGDE 512

>>>Primtallene som brukes
#define PRIME1 12289
#define PRIME2 18433
#define PRIME3 40961


void skrivarray(tall,n)
>>>Skriver ut et tall med lengde n.
Se tillegg.


void lesinnettall(tall,n)
>>>Leser inn et tall i 10-tallsystemet, koverterer det om til grunntall BASE og returnerer n som er lengden.
Se tillegg.


void FFT(NN,a,rot1,rot2,rot3,a1,a2,a3)
long NN,*a,rot1,rot2,rot3,*a1,*a2,*a3;
>>>NN er lengden til polynomet a, rot1-3 er NN-te enhetsrøtter mod PRIME1-3 og a1-3 er a evaluert med respektive enhetsrøtter. Denne prosedyren evaluerer a modulo alle primtallene
{
long
b1[ARRAYLENGDE],b2[ARRAYLENGDE],
b3[ARRAYLENGDE],c1[ARRAYLENGDE],
c2[ARRAYLENGDE],c3[ARRAYLENGDE],
b[ARRAYLENGDE/2],
c[ARRAYLENGDE/2],
k,i,j,n,r1,r2,r3;
>>>Hvis lengde 1 returneres a[0]
if (NN==1) a1[0]=a2[0]=a3[0]=a[0];
else
{
   n=NN/2;

>>>Oppdeling av polynomet
   j=0;
   for (i=0;i<NN;i+=2)
   {
      b[j]=a[i];
      c[j]=a[i+1];
      j++;
   }
>>>Beregne ω²
   r1=rot1*rot1%PRIME1;
   r2=rot2*rot2%PRIME2;
   r3=rot3*rot3%PRIME3;

>>>Rekursive kall
   FFT(n,b,r1,r2,r3,b1,b2,b3);
   FFT(n,c,r1,r2,r3,c1,c2,c3);

>>>Sette sammen resultatene
   r1=r2=r3=1;
   for (k=0;k<n;k++)
   {
      a1[k]=(b1[k]+r1*c1[k])%PRIME1;
      a1[k+n]=(b1[k]-r1*c1[k])%PRIME1;
      if (a1[k+n]<0) a1[k+n]+=PRIME1;
      r1=r1*rot1%PRIME1;

      a2[k]=(b2[k]+r2*c2[k])%PRIME2;
      a2[k+n]=(b2[k]-r2*c2[k])%PRIME2;
      if (a2[k+n]<0) a2[k+n]+=PRIME2;
      r2=r2*rot2%PRIME2;

      a3[k]=(b3[k]+r3*c3[k])%PRIME3;
      a3[k+n]=(b3[k]-r3*c3[k])%PRIME3;
      if (a3[k+n]<0) a3[k+n]+=PRIME3;
      r3=r3*rot3%PRIME3;
   }
}
}
```

```
void FFTInv(NN,a,prim,rot,a1)
long NN,*a,prim,rot,*a1;
>>>NN er lengden til polynomet a. Vi har en NN-
te (enhets)rot mod prim og a1 returnerer
resultatene.
{
long
b1[ARRAYLENGDE],c1[ARRAYLENGDE],
b[ARRAYLENGDE],c[ARRAYLENGDE],
k,i,j,n,r1;

if (NN==1) a1[0]=a[0];

else
{
   n=NN/2;
>>>Oppdeling av polynomet
   j=0;
   for (i=0;i<NN;i+=2)
   {
      b[j]=a[i];
      c[j]=a[i+1];
      j++;
   }
Beregne ω²
   r1=rot*rot%prim;

Rekursive kall
   FFTInv(n,b,prim,r1,b1);
   FFTInv(n,c,prim,r1,c1);
Sette sammen resultatene
   r1=1;
   for (k=0;k<n;k++)
   {

      a1[k]=(b1[k]+r1*c1[k])%prim;
      a1[k+n]=(b1[k]-r1*c1[k])%prim;
      if (a1[k+n]<0) a1[k+n]+=prim;
      r1=r1*rot%prim;
   }
}
}
```

```
void CRT(r0,r1,r2,u)
long r0,r1,r2,*u;
Inn: r0 mod PRIME1...r2 mod PRIME3
Ut:   u≡r0 mod PRIME1
      u≡r1 mod PRIME2
      u≡r2 mod PRIME3
{
   long rho,t,mente,inv,m[3];

   u[0]=r0;
   m[0]=PRIME1;

3=inv(PRIME1*PRIME2)
   rho=((r1-u[0])*3)%PRIME2;
   if (rho<0) rho+=PRIME2;
   u[0]+=rho*m[0];

m=m*PRIME2
   m[0]=3137;
   m[1]=2652;
   m[2]=2;

21813=inv(PRIME1*PRIME2,PRIME3)

rho=((r2-(u[0]%PRIME3))*21813)%PRIME3;
   if (rho<0) rho+=PRIME3;

Beregning av rho·m
   mente=0;
   t=m[0]*rho+mente;
   m[0]=t%BASE;
   mente=t/BASE;

   t=m[1]*rho+mente;
   m[1]=t%BASE;
   mente=t/BASE;

   m[2]=m[2]*rho+mente;

Beregning av u+rho·m
   mente=0;
   t=m[0]+u[0]+mente;
   u[0]=t%BASE;
   mente=t/BASE;

   t=m[1]+mente;
   u[1]=t%BASE;
   mente=t/BASE;

   u[2]=m[2]+mente;
}
```

>>>Hovedprogrammet

```
main()
{
long
a[ARRAYLENGDE],a[ARRAYLENGDE],
a1[ARRAYLENGDE],a2[ARRAYLENGDE],
a3[ARRAYLENGDE],b1[ARRAYLENGDE],
b2[ARRAYLENGDE],b3[ARRAYLENGDE],
svar[ARRAYLENGDE],
rot1,rot2,rot3,
InvRot1,InvRot2,InvRot3,
Ninv1,Ninv2,Ninv3,
i,lengde,ex,mente,t,j,koeff[3];
```

Dette programmet mangler automatisk utregning av enhetsrøtter modulo de respektive primtall. Disse verdiene må derfor skrives inn her avhengig av lengden til tallene. Problemet kan imidlertid lett løses

```
rot1=1479; InvRot1=10810; Ninv1=9217;
rot2=6531; InvRot2=11902; Ninv2=13825;
rot3=14541; InvRot3=26420; Ninv3=30721;
```

Innlesning av tall a og b samt lengde = 2^{ex}

```
    raskinnles(a,b,&lengde,&ex);
```

Rekursive kall

```
    FFT(lengde,a,rot1,rot2,rot3,a1,a2,a3);
    FFT(lengde,b,rot1,rot2,rot3,b1,b2,b3);
```

Punktvis multiplikasjon

```
    for (i=0;i<lengde;i++)
    {
      a1[i]=(a1[i]*b1[i])%PRIME1;
      a2[i]=(a2[i]*b2[i])%PRIME2;
      a3[i]=(a3[i]*b3[i])%PRIME3;
    }
```

Rekursive kall – FFT invers

```
    FFTInv(lengde,a1,PRIME1,InvRot1,b1);
    FFTInv(lengde,a2,PRIME2,InvRot2,b2);
    FFTInv(lengde,a3,PRIME3,InvRot3,b3);
```

Multiplikasjon med den inverse av lengden

```
    for (i=0;i<lengde;i++)
    {
      b1[i]=(b1[i]*Ninv1)%PRIME1;
      b2[i]=(b2[i]*Ninv2)%PRIME2;
      b3[i]=(b3[i]*Ninv3)%PRIME3;
    }
```

Nullstilling av svar array

```
    for (i=0;i<lengde;i++) svar[i]=0;

    for (i=0;i<lengde;i++)
    {
```

Beregner:
koeff ≡ b1[i] mod PRIME1
koeff ≡ b2[i] mod PRIME2
koeff ≡ b3[i] mod PRIME3

```
      CRT(b1[i],b2[i],b3[i],koeff);
```

Evaluering med grunntall

```
      mente=0;j=0;
      do
      {
        t=svar[i+j]+koeff[j]+mente;
        svar[i+j]=t%BASE;
        mente=t/BASE;
        j++;
      }
      while (j<3);
    }
```

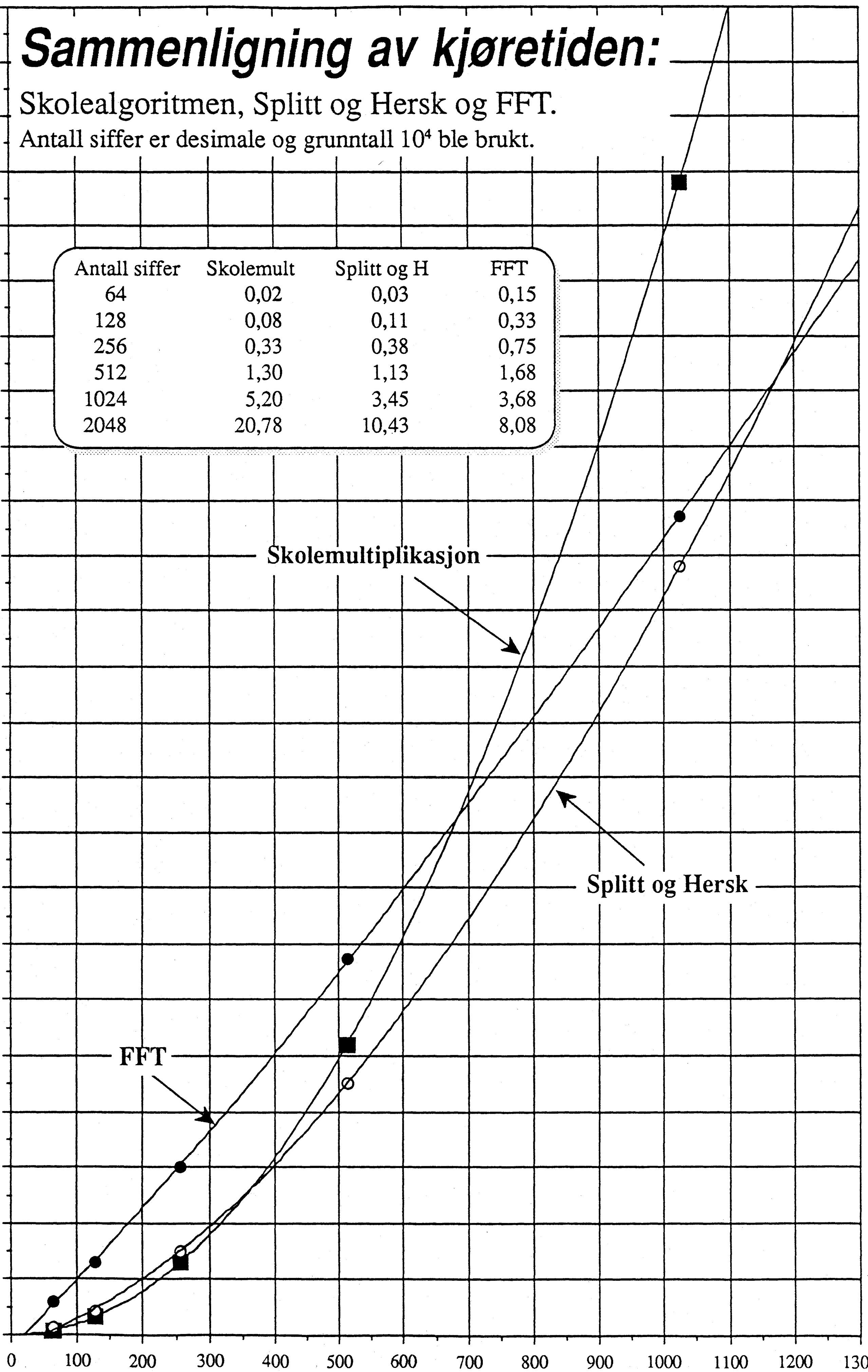
Skriver ut svaret

```
    skrivarray(a1,lengde);
  }
}
```

Tidsforbruk

kjørt på Mac IIx med grunntall 10^4

Antall siffer(desimale)	Sekunder
64	0.15
128	0.33
256	0.75
512	1.68
1024	3.68
2048	8.08


Sammenligning av kjøretiden:
Skolealgoritmen, Splitt og Hersk og FFT.
Antall siffer er desimale og grunntall 10⁴ ble brukt.
Antall siffer | Skolemult | Splitt og H | FFT
64 | 0,02 | 0,03 | 0,15
128 | 0,08 | 0,11 | 0,33
256 | 0,33 | 0,38 | 0,75
512 | 1,30 | 1,13 | 1,68
1024 | 5,20 | 3,45 | 3,68
2048 | 20,78 | 10,43 | 8,08
Skolemultiplikasjon
Splitt og Hersk
FFT
Sekunder
Antall Siffer
6,00
5,75
5,50
5,25
5,00
4,75
4,50
4,25
4,00
3,75
3,50
3,25
3,00
2,75
2,50
2,25
2,00
1,75
1,50
1,25
1,00
0,75
0,50
0,25
0,00
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300

3.9 Modulær Multiplikasjon

3.9.1 Teorien bak Modulær aritmetikk

Det Kinesiske Restledds-teorem [30]:

La $p_0, p_1, \dots, p_{k-1}$ være positive heltall med $\gcd(p_i, p_j) = 1$ for alle $i \neq j$.
Da har settet av kongruensligninger

$$x \equiv x_0 \pmod{p_0}$$
$$x \equiv x_1 \pmod{p_1}$$
$$\vdots$$
$$x \equiv x_{k-1} \pmod{p_{k-1}}$$

en entydig løsning modulo $M = p_0 \cdot p_1 \cdot \dots \cdot p_{k-1}$.

Bevis

Først konstruerer vi ei løsning til ligningssettet.
La $M_i = M/p_i = p_1 \cdot p_2 \cdot \dots \cdot p_{i-1} \cdot p_{i+1} \cdot \dots \cdot p_{k-1}$.
Vi vet at $(M_i, p_i) = 1$, siden $(p_j, p_i) = 1$ så lenge $j \neq i$.
Vi kan derfor finne en invers y_i til M_i mod p_i, slik at $M_i \cdot y_i \equiv 1 \pmod{p_i}$.
Vi konstruerer x som følger: $x = x_0 M_0 y_0 + x_1 M_1 y_1 + \dots + x_{k-1} M_{k-1} y_{k-1} \pmod{M}$
Heltallet x er da løsning på på de k kongruensene.
Vi må vise at $x \equiv x_i \pmod{p_i}$ for $i = 0, 1, \dots, k-1$.
Siden $p_i \mid M_j$ når $j \neq i$ har vi at $M_j \equiv 0 \pmod{p_i}$.
Derfor er alle leddene i summen til x untatt den i-te kongruent med $0 \pmod{p_i}$.
Dermed er $x \equiv x_i M_i y_i \equiv x_i \pmod{p_i}$ siden $M_i \cdot y_i \equiv 1 \pmod{p_i}$.
Vi må også vise at løsningen er entydig (mod M)
La a_0 og a_1 begge være løsninger til ligningssystemet.
Da vil det for hver i, $a_0 \equiv a_1 \equiv x_i \pmod{p_i}$, slik at $p_i \mid (a_0 - a_1)$.
Av det følger at $M \mid (a_0 - a_1)$.
Derfor er $a_0 \equiv a_1 \pmod{M}$ som viser at løsningen er entydig modulo M.

Det kinesiske restledds teoremet (CRT) gir oss en interessant og lettfattelig måte å regne med store tall på en datamaskin. CRT sier at gitt parvise relativt primiske moduli $p_0, p_1, \dots, p_{k-1}$, så kan et positivt heltall n med $n < M = p_0 \cdot p_1 \cdot \dots \cdot p_{k-1}$, bestemmes entydig ved de positive restene modulo p_i for $i = 0, 1, \dots, k-1$.
La oss si at vi skal multiplisere to tall på en datamaskin med ordstørrelse 100. Ta $p_0 = 99$, $p_1 = 98$, $p_2 = 97$ og $p_3 = 95$. Vi kan nå multiplisere tall der produktet ikke overstiger $99 \cdot 98 \cdot 97 \cdot 95$, dvs. 89.403.930. Tallene som skal multipliseres konverterer vi om til 4-tupler av rester modulo p_0, p_1, p_2 og p_3.
Hvis $x \equiv x_i \pmod{m_i}$ og $y \equiv y_i \pmod{p_i}$ så er:

$$x \cdot y \equiv x_i \cdot y_i \pmod{p_i}$$

---

[30] Elementary Number Theory and its Applications – K.Rosen s.127

og ikke nok med det men vi har også at:

$$x + y \equiv x_i + y_i \pmod{p_i}$$
$$x - y \equiv x_i - y_i \pmod{p_i}$$

Et eksempel:

Hvis $x = 122$ og $y = 376$ så får vi at

$x \equiv 23 \pmod{99}$ $y \equiv 79 \pmod{99}$ $x \cdot y \equiv 35 \pmod{99}$

$x \equiv 24 \pmod{98}$ $y \equiv 82 \pmod{98}$ $x \cdot y \equiv 8 \pmod{98}$

$x \equiv 25 \pmod{97}$ $y \equiv 85 \pmod{97}$ $x \cdot y \equiv 88 \pmod{97}$

$x \equiv 27 \pmod{95}$ $y \equiv 91 \pmod{95}$ $x \cdot y \equiv 82 \pmod{95}$

Så bruker vi CRT og får at $x \cdot y \equiv 45872 \pmod{89.403.930}$

Overraskende ser vi at hvis vi har tallene på den "rette" formen, kan vi multiplisere to tall med n siffer ved å bruke $O(n)$ operasjoner. Dette gjelder også for addisjon og subtraksjon. En annen stor fordel er mulighetene for parallellprosesering av en enkelt regneoperasjon. Dette er umulig i f.eks. skolealgoritmen for addisjon, siden menten forplanter seg utover i beregningen. Her kan vi foreta alle enkeltberegninger samtidig. Problemet med modulær representasjon av tall er at det vanskelig å sjekke om et tall er større enn et annet, divisjon er vanskelig og konverteringen til og fra den vanlige måten å representere tall på tar tid.

Teoretisk sett vil det være best å velge p_i-ene som de største heltallene $< W$ som er innbyrdisk primiske. På binære datamaskiner er det ofte fordelaktig å velge p_i-ene på en annen måte [31]. Ved å velge $p_i = 2^{e_i} - 1$ blir de elementære aritmetiske operasjonene enklere, siden det er lett å regne modulo $2^{e_i} - 1$ (eners komplement aritmetikk). For å sjekke at tall på formen $2^{e_i} - 1$ er primiske, kan vi benytte oss av et teorem som sier at $\gcd(2^e - 1, 2^f - 1) = 2^{(e,f)} - 1$.

3.9.2 Konvertering til modulær representasjon[32]

Dette kan gjøres på mange måter. Gitt et positivt heltall x. Den mest opplagte metoden er å beregne:

$$x_0 = x \bmod p_0$$
$$x_1 = x \bmod p_1$$
$$\vdots$$
$$x_{k-1} = x \bmod p_{k-1},$$

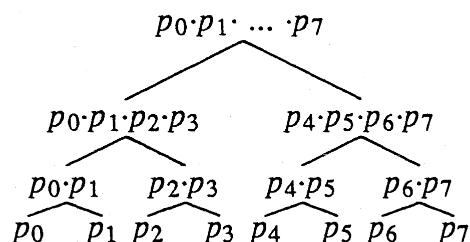
slik at vi får modulær-representasjonen $(x_0, x_1, \ldots, x_{k-1})$

Antar vi at hver p_i har lengde b-bits, så vil x ha en lengde på rundt bk-bits. Divisjonene av x med p_i-ene blir k divisjoner av kb-bits heltall med b-bits heltall.

Hvis vi deler opp hver divisjon slik at vi får k divisjoner av $2b$-bits heltall med b-bits heltall, får denne metoden kompleksiteten $O(k^2 D(b))$, der $D(b)$ er tiden det tar å dele et $2b$-bits heltall på et b-bits heltall.

[31] The Art of Computer Programming: Seminumerical Algorithms – Donald Knuth s.270
[32] The Design and Analysis of Computer Algorithms - Aho/Hopcroft/Ullman s.289

Denne metoden kan forbedres ved å bruke samme prinsippet som i iterative FFT.
Anta at k er en potens av 2.
Istedet for å dele x med hver p_i kan vi først dele x med hver produktene $p_0 \cdot p_1 \cdot \ldots \cdot p_{k/2-1}$ og $p_{k/2} \cdot p_{k/2+1} \cdot \ldots \cdot p_{k-1}$, slik at vi får rester r_1 og r_2.
Resten r_1 deler vi så videre på produktene $p_0 \cdot p_1 \cdot \ldots \cdot p_{k/4-1}$ og $p_{k/4} \cdot p_{k/4+1} \cdot \ldots \cdot p_{k/2-1}$, slik at vi får rester t_1 og t_2. Resten r_2 deler vi på produktene $p_{k/2} \cdot p_{k/2+1} \cdot \ldots \cdot p_{3k/4-1}$ og $p_{3k/4} \cdot p_{3k/4+1} \cdot \ldots \cdot p_{k-1}$, slik at vi får rester t_3 og t_4.
Vi fortsetter på samme måte helt til produktet av alle moduliene er delt opp til de enkelte modulene. Etter å ha delt med disse også, sitter vi igjen med de ønskete rester (mod p_0, $p_1, \ldots, p_{k-1}$). Figuren under viser alle divisorene vi får når $k = 8$.



Denne metoden har kompleksitet O(M(bk) log k), der M(n) er tiden det tar å multiplisere n–bits heltall. Dette får vi siden vi etter log k iterasjoner har delt produktene opp i de enkelte primtallene og divisjonene i hver iterasjon krever maksimalt M(bk)[33] tid.

Når grunntallet er 2 og modulusen p_i er på den spesielle formen $2^{e_i} - 1$, kan vi gjøre det enkelt ved å gruppere binær-representasjonen til x i blokker på e_i-bits.
$x = a_t A^t + a_{t-1} A^{t-1} + \ldots + a_1 A + a_0$ der $A = 2^{e_i}$ og $0 \leq a_k < 2^{e_i}$ for $0 \leq k \leq t$.
Da er $x \equiv a_t + a_{t-1} + \ldots + a_1 + a_0 \pmod{2^{e_i} - 1}$

3.9.3 Konvertering fra modulær representasjon[34]

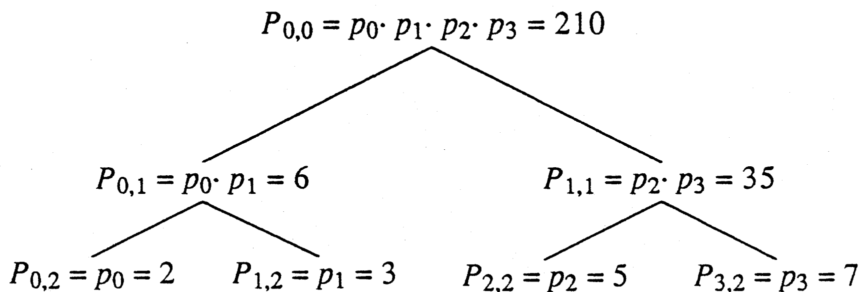
Her bruker vi CRT slik som vist i 3.9.1. Vi har at $M = p_0 \cdot p_1 \cdot \ldots \cdot p_{k-1}$, $M_i = M/p_i$ og y_i som er en invers av $M_i \pmod{p_i}$. Da er $x = x_0 M_0 y_0 + x_1 M_1 y_1 + \ldots + x_{k-1} M_{k-1} y_{k-1}$ (mod M). Siden beregningen av inversene er den samme fra gang til gang, kan vi gå ut fra at de er beregnet på forhånd og er en del av inputdata.
Hva er kompleksiteten til denne metoden? Hvis p_i-ene har lengde b-bits, så har x_i lengde b-bits, y_i lengde b-bits og M_i lengde bk-bits. Anta $b \approx k$. Da krever beregningen $a_i M_i y_i$ (mod M) O(M(bk)) tid. Denne utføres k ganger. Kompleksiteten blir da O($k \cdot$M(bk)).

Ser vi på leddene som gir oss x ser vi at de har mange faktorer felles. $a_i M_i y_i$ har f.eks. $p_0 \cdot p_1 \cdot \ldots \cdot p_{k/2-1}$ som faktor for $i \geq k/2$, og $p_{k/2} \cdot p_{k/2+1} \cdot \ldots \cdot p_{k-1}$ som faktor for $i < k/2$.
Vi kan derfor skrive x som $\left(\sum_{i=0}^{(k/2)-1} x_i M_i' y_i \right) \times \prod_{i=(k/2)+1}^{k-1} p_i + \left(\sum_{i=(k/2)+1}^{k-1} x_i M_i'' y_i \right) \times \prod_{i=0}^{(k/2)-1} p_i$
der M_i' er produktet $p_0 \cdot p_1 \cdot \ldots \cdot p_{(k/2)-1}$ delt på p_i, og M_i'' er produktet $p_{k/2} \cdot p_{k/2+1} \cdot \ldots \cdot p_{k-1}$

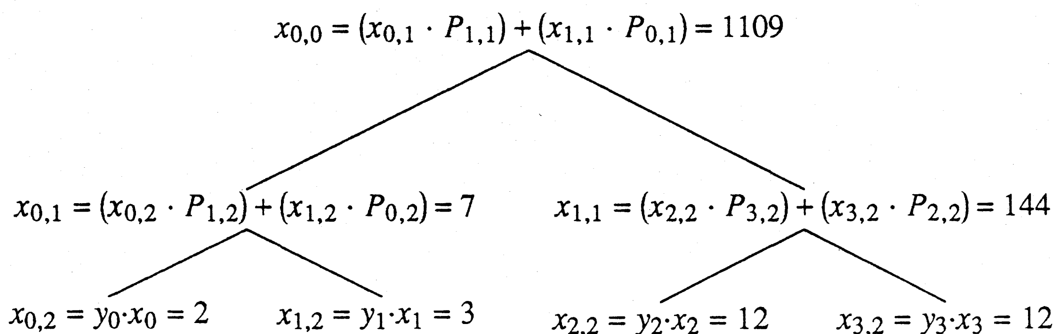
[33] Husk at M(n) og D(n) er lineært reduserbart til hverandre.
[34] The Design and Analysis of Computer Algorithms - Aho/Hopcroft/Ullman s.294

delt på p_i. Vi har nå delt summen som gir oss x i to deler i tillegg til at vi har halvert lengden til M_i. Fortsetter vi på samme måte får vi en splitt og hersk algoritme lik den i 3.9.2, bortsett fra at her er det mest praktisk å begynne utregningen på bladnivå i binærtreet.

La oss se på et eksempel:
La $p_0 = 2, p_1 = 3, p_2 = 5, p_3 = 7$
og $(x_0, x_1, x_2, x_3) = (1,2,4,3)$.
De prekalkulerte inversene er $y_0 = 1, y_1 = 1, y_2 = 3, y_3 = 4$.
Først bygger vi opp et binærtre på samme måte som i 3.9.2 (Se figur under). En variabel $P_{0,2}$ betyr variabel P, forgreining 0 på nivå 2. Rota har nivå 0.



Så bygger vi et nytt tre ved å starte på bladnivå og bygge oss oppover (Se figur under). Bladene får verdiene $y_i \cdot x_i$. Disse går så sammen 2 og 2. Verdien til forgreiningspunktet til 2 blader er verdien til det ene bladet multiplisert med P-verdien til det andre bladet addert med verdien til det andre bladet multiplisert med P-verdien til det første bladet. Slik forsetter vi til vi står igjen med verdien x.



Hva er kompleksiteten til denne metoden? Hvis p_i-ene har lengde b-bits, så har x_i lengde b-bits, y_i lengde b-bits og M_i lengde bk-bits. Anta $b \approx k$. Da krever beregningen på hvert nivå i $O(2^i \cdot M(b/2^i))$ tid, og siden vi har log k nivåer blir kompleksiteten $O(M(bk) \log k)$

Bedre enn dette er det vanskelig å få til. Som en følge av en rask algoritme for største felles divisor som står beskrevet i [35] vil algoritmen ha samme kompleksitet selv om vi ikke prekalkulerer inversene. Kompleksiteten til modulær multiplikasjon ved å bruke metodene som er beskrevet over er dermed $O(M(n) \cdot \log k)$, der $M(n)$ er tiden det tar å multiplisere n-bits

[35] The Design and Analysis of Computer Algorithms - Aho/Hopcroft/Ullman s.310

heltall ved å bruke en eller annen multiplikasjonsalgoritme. Hva som skjer hvis man bruker algoritmen rekursivt har jeg ikke sett på.

Spesielt er modulær aritmetikk bra hvis det er muligheter for parallellprosesering, og ved beregninger som krever mange multiplikasjoner, addisjoner og subtraksjoner uten at man trenger å konvertere frem og tilbake hele tiden. F.eks. og opphøye et stort tall i et annet stort tall.
I hvor stor grad algoritmene for modulær aritmetikk egner seg for å multiplisere tall i forhold til f.eks. skolealgoritmen og FFT har jeg ikke sett på.

3.10 Schönhage - Strassen metoden [36]

Jeg skal nå skissere den raskeste av alle multiplikasjonsalgoritmer asymptotisk sett – Schönhage-Strassen algoritmen. Denne metoden har kompleksitet $O(n \cdot \log_2 n \cdot \log_2 \log_2 n)$. Dette er egentlig dårligere enn FFT, men vi må huske på at FFT er en algoritme av subasymptotisk karakter som bare har gyldighet for tall under en viss størrelse. Algoritmen er nok forøvrig mer av teoretisk interesse enn av praktisk betydning.
Jeg vil ikke hevde å kunne algoritmen til bunns, til det er den litt for "teknisk". Målet mitt har vært å bli bevist idéene som ligger bak.
I del 3.5 så vi på en splitt og hersk algoritme som delte opp tallene i 2 deler, for så å få 3 multiplikasjoner med bare halvparten så lange tall. Denne metoden kan generaliseres ved å dele hvert av tallene opp i b blokker, der hver blokk består av l bits. Vi kan nå se på de b blokkene som koeffisienter i et polynom. For å finne koeffisientene til produktet mellom to polynomer evaluerer vi polynomene i en del "lure" punkter, foretar punktmultiplikasjon og interpolerer. Velger vi punktene som primitive enhetsrøtter kan vi benytte oss av Fouriertransformasjoner og foldingsteoremet. Lar vi b være en funksjon av n, og bruker rekursjon, kan vi multiplisere to n-bits heltall ved å bruke $O(n \cdot \log_2 n \cdot \log_2 \log_2 n)$ operasjoner.
For å gjøre det litt enklere fyller vi på med nullere slik at n blir en potens av 2. Denne algoritmen multipliserer to n-bits tall (mod $2^n + 1$), så for å få riktig svar må vi fylle på med nye nullere, slik at vi egentlig multipliserer to $2n$-bits tall (mod $2^{2n} + 1$). Påfyllingen av nullere vil kun øke kjøretiden med en konstant. All regningen foregår i en kommutativ ring.
Anta at u og v er to n-bits tall som vi skal multiplisere (mod $2^n + 1$) og at $n = 2^k$. La $b = 2^{k/2}$ hvis k er et partall. Ellers er $b = 2^{(k-1)/2}$. La $l = n/b$. Så deler vi opp u og v i b blokker med l bits i hver blokk. Dvs.

$$u = u_{b-1}2^{(b-1)l} + \ldots + u_1 2^l + u_0 \text{ og}$$
$$v = v_{b-1}2^{(b-1)l} + \ldots + v_1 2^l + v_0$$

---

[36] The Design and Analysis of Computer Algorithms - Aho/Hopcroft/Ullman s.270
Algorithms: Their Complexity and Efficiency – L.Kronsjö

Produktet mellom u og v blir da

$$u \cdot v = y_{2b-2}2^{(2b-2)l} + \ldots + y_1 2^l + y_0, \text{ der } y_i = \sum_{j=1}^{b-1} u_j v_{i-j}, \; 0 \le i < 2b.$$

$u_j = v_j = 0$ for $j < 0$ eller $j > b - 1$. $y_{2b-1} = 0$ og er tatt med for symmetriens skyld.

Vi kan også beregne $u \cdot v$ ved å bruke foldingsteoremet. Punktmultiplikasjonen ville da kreve $2b$ multiplikasjoner. Ved å bruke sammenlappete (wrapped) foldinger kan antallet multiplikasjoner reduseres til b. Dette er grunnen til at vi beregner $u \cdot v \pmod{2^n + 1}$.

Definisjon av sammenlappete foldinger:

La $\vec{a} = [a_0, a_1, \ldots, a_{n-1}]^T$ og $\vec{b} = [b_0, b_1, \ldots, b_{n-1}]^T$ være 2 vektorer med lengde n. La ω være en primitiv n-te enhetsrot og la $\psi^2 = \omega$ og anta at n har multiplikative inverser.

Den **positivt sammenlappete foldingen** av $\vec{a}$ og $\vec{b}$ er vektoren $\vec{c} = [c_0, c_1, \ldots, c_{n-1}]^T$ der $c_i = \sum_{j=0}^{i} a_i b_{i-j} + \sum_{j=i+1}^{n-1} a_j b_{n+i-j}$. Vi får da at $\vec{c} = F^{-1}(F(\vec{a}) \cdot F(\vec{b}))$.

Den **negativt sammenlappete foldingen** av $\vec{a}$ og $\vec{b}$ er vektoren $\vec{d} = [d_0, d_1, \ldots, d_{n-1}]^T$ der $d_i = \sum_{j=0}^{i} a_i b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j}$.

La $\vec{A} = [a_0, \psi a_1, \ldots, \psi^{n-1} a_{n-1}]^T$, $\vec{B} = [b_0, \psi b_1, \ldots, \psi^{n-1} b_{n-1}]^T$,

$\vec{D} = [d_0, \psi d_1, \ldots, \psi^{n-1} d_{n-1}]^T$.

Da er $\vec{D} = F^{-1}(F(\vec{A}) \cdot F(\vec{B}))$.

Regner vi modulo $\pmod{2^n + 1}$ får vi at

$u \cdot v \equiv (w_{b-1}2^{(b-1)l} + \ldots w_1 2^l + w_0) \pmod{2^n + 1}$, der $w_i = y_i - y_{b-i}$, $0 \le i < b$.

Produktet av to l-bits tall er mindre enn 2^{2l}.

Siden y_i og y_{b-i} er summen av henholdsvis $i + 1$ og $b - (i + 1)$ slike produkter, må $w_i = y_i - y_{b-i}$ ligge i området $-(b - 1 - i)2^{2l} < w_i < (i + 1)2^{2l}$.

Det betyr at w_i maksimalt kan anta $b2^{2l}$ forskjellige verdier.

For å få det til å gå raskt beregnes w_i-ene først modulo b og så modulo $2^{2l} + 1$.

Vi får da at $w_i' = w_i$ modulo b og $w_i'' = w_i$ modulo $2^{2l}+1$.

Tilslutt beregner vi w_i-ene nøyaktig ved å bruke formelen

$w_i = (2^{21} + 1)\left(\left(w_i' - w_i''\right) \text{ modulo } b\right) + w_i''$.

3.10.1 Nøyaktig beskrivelse av algoritmen

Inn: to n-bits heltall u og v der $n = 2^k$

Ut: Et $(n + 1)$-bits produkt av u og v modulo $(2^n + 1)$

La $b = 2^{k/2}$ hvis k er et partall, ellers la $b = 2^{(k-1)/2}$. La $l = n/b$.

Skriv u som $\sum_{i=0}^{b-1} u_i 2^{li}$ og v som $\sum_{i=0}^{b-1} v_i 2^{li}$, der u_i og v_i er heltall mellom 0 og $2^l - 1$.

Dvs u_i-ene er blokker på l-bits av u og tilsvarende for v.

1. Beregn Fouriertransformasjonen modulo $2^{2l} + 1$ av $[u_0, \psi u_1, ..., \psi^{b-1} u_{b-1}]^T$ og $[v_0, \psi v_1, ..., \psi^{b-1} v_{b-1}]^T$, der $\psi = 2^{2l/b}$ ved å bruke ψ^2 som primitiv b-te enhetsrot.

2. Beregn parvise produkt av Fouriertransformasjonen fra steg 1 modulo $2^{2l} + 1$ ved å bruke algoritmen rekursivt for å beregne hvert enkelt produkt. (Situasjonen der et av tallene er 2^{2l} kan behandles som et lett spesialtilfelle)

3. Beregn den Inverse Fouriertransformasjonen modulo $2^{2l} + 1$ av vektoren av parvise produkter fra steg 2. Resultatet av denne beregningen vil være $[w_0, \psi w_1, ..., \psi^{b-1} w_{b-1}]^T$ modulo $(2^{2l} + 1)$, der w_i er det i-te leddet til den negativt sammenlappete foldingen til $[u_0, u_1, ..., u_{b-1}]^T$ og $[v_0, v_1, ..., v_{b-1}]^T$. Beregn $w_i'' = w_i$ modulo $2^{2l}+1$ ved å multiplisere $\psi^i w_i$ med ψ^{-1} modulo $(2^{2l} + 1)$.

4. Beregn $w_i' = w_i$ modulo b som følger:
   a) La $u_i' = u_i$ modulo b og la $v_i' = v_i$ modulo b for $0 \le i < b$.
   b) Konstruer tallene $\hat{u}$ og $\hat{v}$ ved å legge $2 \cdot \log_2 b$ 0-ere til hver av u_i' og v_i'-ene. Dvs. $\hat{u} = \sum_{i=0}^{b-1} u_i' \cdot 2^{(3 \cdot \log_2 b)i}$ og $\hat{v} = \sum_{i=0}^{b-1} v_i' \cdot 2^{(3 \cdot \log_2 b)i}$.
   c) Beregn produktet $\hat{u} \cdot \hat{v}$ ved å bruke Splitt og hersk multiplikasjon.
   d) Produktet $\hat{u} \cdot \hat{v}$ er $\sum_{i=0}^{2b-1} y_i' \cdot 2^{(3 \cdot \log_2 b)i}$ der $y_i' = \sum_{j=0}^{2b-1} u_j' \cdot v_{i-j}'$. Vi kan få w_i-ene modulo b fra dette produktet ved å evaluere $w_i' = \left(y_i' - y_{i-j}'\right)$ modulo b for $0 \le i < b$.

5. Nå kan vi regne ut w_i-ene nøyaktig ved å bruke formelen: $w_i = (2^{2l} + 1)\left(\left(w_i' - w_i''\right) \text{modulo } b\right) + w_i''$. w_i ligger nå mellom $(b - 1 - i)2^{2l}$ og $(i + 1)2^{2l}$.

6. Beregn $\sum_{i=0}^{b-1} w_i 2^{li}$ modulo $(2^n + 1)$ som er svaret vi ønsker.

3.11 Divisjon – Skolealgoritmen

3.11.1 Litt teori bak algoritmen

Gitt to heltall $a = (a_{m-1}a_{m-2}...a_1a_0)_G$ og $b = (b_{n-1}b_{n-2}...b_1b_0)_G$ med $0 < b \le a$.
Da kan vi finne $q = (q_{m-n}q_{m-n-1}...q_1q_0)_G$ og $r = (r_{n-1}r_{n-2}...r_1r_0)_G$
slik at $a = b \cdot q + r$, der $0 \le r < b$.

Når vi dividerer for hånd vil det alltid bli endel gjetting underveis, og dette kan være vanskelig å programmere. Hvis vi ser på den vanlige skolealgoritmen for divisjon, kan den beskrives på følgende måte:

1. La a = $(a_m a_{m-1}...a_1a_0)_G$, dvs. vi setter inn et ledende siffer $a_m = 0$.
2. Del de $n + 1$ mest signifikante sifrene i a på b slik at $0 \le \left\lfloor \frac{a_m a_{m-1} \ldots a_{m-n}}{b} \right\rfloor = q_{m-n} < G$.
3. Trekk $q_{m-n} \cdot b \cdot G^{m-n}$ fra a
4. $m = m - 1$.
5. Hvis $m \ge n$ går vi tilbake til steg 2.
6. Det som er igjen av a er resten r.

Dette kan vi skissere slik:

$$q_{m-n}q_{m-n-1} \ldots q_1q_0$$

$$b_{n-1}b_{n-2} \ldots b_1b_0 \; \big) \; a_m a_{m-1} \ldots a_{m-n+1} a_{m-n} a_{m-n-1} \ldots a_1 a_0$$

$$- \longleftarrow \; q_{m-n} \cdot b \; \longrightarrow \qquad \downarrow$$

$$= \text{Ny } a\text{:} \longleftarrow \; r \; \longrightarrow a_{m-n-1} \ldots a_1 a_0$$

Problemet blir nå å "gjette" hva q blir.
Den beste metoden er å gjette på q utfra de første sifrene til a og b. Et godt forslag til q er:

$$Q = \min\left(\left\lfloor \frac{a_m G + a_{m-1}}{b_{n-1}} \right\rfloor, G - 1\right)$$

Vi får Q ved å dele de 2 første sifrene i a med første siffer i b, og hvis resultatet er større enn G erstatter vi det med $G - 1$. Det viser seg at Q alltid er en meget bra approksimasjon til q så lenge b_{n-1} er tilstrekkelig stor.

Teorem [37] Gitt approksimasjonen Q av q som over. Da er:
1. $Q \ge q$
2. Hvis $b_{n-1} \ge \frac{G}{2}$, så er $Q \le q + 2$. Det betyr at $q \le Q \le q + 2$.

---

[37] A Concrete Iintroduction to Higher Algebra – Lindsay Childs s.42

Bevis

(1) Siden $q{\cdot}b \leq a$ så er

$$q{\cdot}(b_{n-1}G^{n-1} + b_{n-2}G^{n-2} + \ldots + b_0) \leq (a_nG^n + a_{n-1}G^{n-1} + \ldots + a_0).$$

Derfor er

$$q{\cdot}b_{n-1}{\cdot}G^{n-1} \leq a_nG^n + a_{n-1}G^{n-1} + \ldots + a_0 = a.$$

Siden $a_{n-2}G^{n-2} + a_{n-3}G^{n-3} + \ldots + a_0 < G^{n-1}$,

$$a < a_nG^n + a_{n-1}G^{n-1} + G^{n-1} = (a_nG + a_{n-1} + 1){\cdot}G^{n-1}$$

er $q{\cdot}b_{n-1}G^{n-1} < (a_nG + a_{n-1} + 1){\cdot}G^{n-1}$.

så $q{\cdot}b_{n-1} < a_nG + a_{n-1} + 1$

Siden begge sider er heltall, så er $q{\cdot}b_{n-1} < a_nG + a_{n-1}$

Av dette følger at $q \leq Q$. For $q \leq G - 1$ siden $q{\cdot}b \leq a < G{\cdot}b$ hvis $b_{n-1} > a_n$, da er $Q{\cdot}b_{n-1}$ definert til å være største multiplum av av b_{n-1} som er $\leq a_nG + a_{n-1}$ dermed er $q \leq Q$. Dette beviser punkt 1.

Anta at $b_{n-1} \geq \frac{G}{2}$.

Vi må nå vise at $(Q - 2){\cdot}b \leq a$.

$(Q - 2){\cdot}b = (Q - 2){\cdot}(b_{n-1}G^{n-1} + b_{n-2}G^{n-2} + \ldots + b_0)$

Siden $b_{n-1}G^{n-1} + b_{n-2}G^{n-2} + \ldots + b_0 < G^{n-1}$ får vi at

$(Q - 2)b < (Q - 2)(b_{n-1} + 1)G^{n-1} = [Q{\cdot}b_{n-1} + (Q - 2 - 2b_{n-1})]G^{n-1}$

$\leq (Ga_n + a_{n-1})G^{n-1} + (Q - 2 - 2b_{n-1})G^{n-1}$.

Husk at $b_{n-1} \geq G/2$ og $Q \leq G - 1$.

Derfor er $Q - 2 - 2b_{n-1} < 0$

Dermed er høyresiden $\leq (Ga_n + a_{n-1})G^{n-1}$

$\leq (Ga_n + a_{n-1})G^{n-1} + a_{n-2}G^{n-2} + a_1G + a_0 = a$ som beviser (2).

3.11.2 Selve algoritmen [38].

Gitt to heltall $a = (a_{m-1}a_{m-2}\ldots a_1)_G$ og $b = (b_nb_{n-1}\ldots b_1)_G$ med $0 < n < m$ og $a > b$. Av praktiske grunner lar jeg første indeks starte på 1.

Da kan vi finne q og r slik at $a = b{\cdot}q + r$, med $0 \leq r < b$ og $q = (q_{m-n}q_{m-n-1}\ldots q_1)_G$ og $r = (r_nr_{n-1}\ldots r_1)_G$.

Steg 1 **Normalisering:** Sett $d = \frac{G}{b_n + 1}$. (All divisjon er heltallsdivisjon). Så settes $(a_ma_{m-1}\ldots a_1)_G$ lik $d{\cdot}(a_{m-1}a_{m-2}\ldots a_1)_G$ og sett $b = b{\cdot}d$. Dette er nødvendig for å garantere $b_n \geq G/2$. Merk innføringen av et ekstra siffer a_m.

Steg 2 **Ferdig ?:** Er $m > n$? Hvis JA fortsett, ellers gå til Steg 8. Løkken fra Steg 3 til Steg 7 er divisjon av $(a_ma_{m-1}\ldots a_{m-n})_G$ med $(b_nb_{n-1}\ldots b_1)_G$ for å få q_{m-n}.

[38] Omskriving av The Art of Computer Programming: Seminumerical Algorithms – Donald Knuth s. 257

Steg 3 **Beregning av Q:** Hvis $a_m = b_n$, sett $Q = G - 1$. Ellers sett $Q = \frac{a_m G + a_{m-1}}{b_n}$. Test nå om $b_{n-1}Q > (a_m G + a_{m-1} - Qb_n)\cdot G + a_{m-2}$. Hvis sant, reduser Q med 1 og gjenta testen. Denne testen er kjapp, og finner i de fleste tilfeller Q som er q + 1, og alle tilfeller av Q som er $q + 2$.

Steg 4 **Multipliser og trekk fra:** La $(a_m a_{m-1} \ldots a_{m-n})_G$ være lik $(a_m a_{m-1} \ldots a_{m-n})_G - Q\cdot(b_n b_{n-1} \ldots b_1)_G$. $(a_m a_{m-1} \ldots a_{m-n})_G$ skal være ≥ 0. Hvis det er blitt negativt, glem den siste negative menten og legg til G^{n+1}, dvs. G komplimentet til den egentlige verdien. Den siste menten skal vi ikke skrive ned, bare huske.

Steg 5 Sett $q_{m-n} = Q$. Hvis resultatet fra Steg 4 var negativt gå videre til Steg 6, ellers gå til Steg 7.

Steg 6 **Adder tilbake:** Sannsynligheten for å komme hit er svært liten, bare $2/G$. Tell ned q_{m-n} og adder $(0b_n b_{n-1} \ldots b_1)_G$ til $(a_m a_{m-1} \ldots a_{m-n})_G$. Menten som kommer til høyre for a_m kan glemmes, siden den oppveier menten fra Steg 4.

Steg 7 Tell ned m å gå tilbake til tilbake til Steg 2.

Steg 8 **Avnormalisere:** q er nå $(q_{m-n} q_{m-n-1} \ldots q_1)_G$ og resten r fåes ved å dele det som er igjen av a på d.

3.11.3 Analyse av algoritmen

Beregningen av Q med testen i

while ((b[n-1]*qq) > ((a[m]*grunntall + a[m-1] – qq*b[n])*grunntall + a[m-2])) qq--;

gir den største enkeltberegningen.

Dvs. $(a_m\cdot G + a_{m-1} - Q\cdot b_n)\cdot G + a_{m-2}$.

Denne er størst når $a_m = a_{m-1} = a_{m-2} = b_n = G - 1$.

Den blir da $((G - 1)G + (G - 1) - (G - 1)(G - 1))G + (G - 1) =$

$(G^2 - G + G - 1 - G^2 + 2G - 1)G + (G - 1) =$

$2G^2 - G - 1$.

Dermed må $2G^2 - G - 1 < W$ for at vi ikke skal få overflow under utregningen.Velger vi grunntall på en vanlig 32-bits maskin ≤ 46340, er vi på den sikre siden mot dette.

Denne algoritmen er litt vanskeligere å finne kompleksiteten til i forhold til skolealgoritmen for multiplikasjon. Hovedløkka while $(m > n)$ utføres $m - n$ ganger. Inne i denne har vi diverse løkker som utføres n ganger. Vi får da $(m - n)\cdot n = m\cdot n - n^2$ som er $O(n^2)$ når $m \approx 2n$.

Plassbehovet er ikke uventet $O(n)$.

Mulighetene for parallellprosessering er dårligere her enn i skolealgoritmen for multiplikasjon. Farten kan sannsynligvis ikke økes med mer enn en faktor 2, uten at jeg vil begrunne dette nærmere.

3.11.4 Et program-eksempel i C

Denne prosedyra trenger 2 positive heltall a,b der a ≥ b og b har minst 2 siffer. Tallene må være gitt på standardform og svaret returneres på samme form. Prosedyra kalles opp med a,b og en tom array q. Resten returneres gjennom a og kvotienten gjennom q.

```
void div(a,b,q)
long *a,*b,*q;

{
   long
   m, n, c[ARRAYLENGDE],
   mente, t, qq, r,d;
   int
   i, q_lengde;
```

Normalisering av tallene hvis nødvendig (steg 1)

```
   d=grunntall/(b[b[0]]+1);
   a[0]++;
   a[a[0]]=0;
   if (d>1)
```

Multipliser a med d og b med d

```
   {
      mult(a,d);
      mult(b,d);
   }


   m=a[0];
   n=b[0];
   c[0]=n+1;
   q_lengde = q[0] = m - n;


   while (m>n)
   {
```

Vi gjetter på q (steg 3)

```
      if (a[m]==b[n]) qq=grunntall-1;
      else qq=(a[m]*grunntall+a[m-1])/b[n];
```

Denne testen avslører de fleste qq som er for stor

```
      while ((b[n-1]*qq) >
      ((a[m]*grunntall+a[m-1]-
      qq*b[n])*grunntall + a[m-2])) qq--;
```

Gang qq med b og vi får c (steg 4)

```
      mente=0;
      for (i=1; i<=n; i++)
      {
         t=b[i]*qq+mente;
         c[i]=t % grunntall;
         mente=t/grunntall;
      }
      c[i]=mente;
      c[i+1]=0;
```

Trekk c fra a (steg 4)

```
      mente=0;
      for (i=1; i<=c[0]; i++)
      {
         t = a[m-c[0]+i] - c[i] + mente;
         if (t<0)
         {
            t=t+grunntall;
            mente=-1;
            a[m-c[0]+i]=t;
         }
         else
         {
            mente = 0;
            a[m-c[0]+i]=t;
         };
      }
```

Addere tilbake hvis c er for stor (steg 6)

```
    if (mente ==-1)
    {
        a[n+1]=(a[n+1]+1) % grunntall;
        mente=0;
        b[n+1]=0;
        for (i=1; i<=c[0]-1; i++)
        {
            t = a[m-c[0]+i] + b[i] + mente;
            a[m-c[0]+i]=t%grunntall;
            mente=t/grunntall;
        }
    qq--;
    }
```

Vi har funnet riktig qq og legger den inn q-arrayet

```
    q[m-n]=qq;

    m--;
  }
```

Legger riktig lengde på q og a inn i henholdsvis q[0] og a[0]

```
  if (q[q_lengde]==0) q[0]--;
  i=a[0]=n;
  while ((a[i]==0) && (i>0)) i--;
  a[0]=i;
```

Av-normalisering av rest (steg 8)

```
  if (d>1)
  {
    r=0;
    m=a[0];
    while (m>0)
    {
      t=a[m];
      a[m]=(r*grunntall+a[m])/d;
      r=(r*grunntall+t)%d;

      m--;
    }
    if (a[a[0]]==0) a[0]--;
  }
}
```

3.12 Rask Eksponensiering (modulo n)

3.12.1 Beskrivelse av algoritmen

Kryptering og dekryptering hører til de tidskritiske delene av RSA systemet. Det er derfor viktig at dette skjer raskt. Beregningen av $E_e(P) = P^e \pmod{n}$ utføres med tall på rundt 200 siffer. Det finnes mange måter å beregne $P^e \pmod{n}$. Vi kan f.eks først regne ut P^e slik at vi får et kjempetall på 40000 siffer. Dette enorme tallet dividerer vi med n for å finne resten C. Denne metoden tar lang tid og krever mye plass. Vi må finne på noe raskere.

En av de enkleste og raskeste algoritmene for kryptering/dekryptering er rask eksponensiering (mod n).

La oss si at vi skal beregne $P^e \equiv C \pmod{n}$ med $P = 123$, $e = 107$ og $n = 101$.
$123^{107} \pmod{101}$. Dette er det samme som
$123^{64+32+8+2+1} \pmod{101} =$
$123^{64} \cdot 123^{32} \cdot 123^{8} \cdot 123^{2} \cdot 123^{1} \pmod{101} =$
$123^{64} \pmod{101} \cdot 123^{32} \pmod{101} \cdot 123^{8} \pmod{101} \cdot 123^{2} \pmod{101} \cdot 123^{1} \pmod{101})$
$\pmod{101}$

Vi kan med andre ord utrykke eksponenten e i 2-tallsystemet. I dette tilfelle $e = (1101011)_2$. Så beregner vi P^1, P^2, P^4, P^8,...osv. ved gjentatte kvadreringer og reduksjoner (mod n) helt til P har en eksponent som er $\leq e$ og $> e/2$. Deretter plukker vi ut de P-ene med eksponenter som tilsvarer 1-erne i $e = (1101011)_2$. Disse multipliserer vi sammen og reduserer modulo n

e						
1	P^1	$\equiv$			22	(mod 101)
1	P^2	$\equiv$	22^2	$\equiv$	80	(mod 101)
0	P^4	$\equiv$	80^2	$\equiv$	37	(mod 101)
1	P^8	$\equiv$	37^2	$\equiv$	56	(mod 101)
0	P^{16}	$\equiv$	56^2	$\equiv$	5	(mod 101)
1	P^{32}	$\equiv$	5^2	$\equiv$	25	(mod 101)
1	P^{64}	$\equiv$	25^2	$\equiv$	19	(mod 101)

Nå er det bare å plukke ut de riktige tallene.
Vi får at $19 \cdot 25 \cdot 56 \cdot 80 \cdot 22 \equiv 76 \pmod{101}$. Dvs at $P^{107} \equiv 76 \pmod{n}$.

I praksis multipliserer vi sammen verdiene etterhvert som de regnes ut for å spare lagerplass.

3.12.2 Et program-eksempel i (Kvasi) Pascal

```
Function FastExp ( P , e , n : integer ) : integer;
var y , z : integer;
begin
    y = 1;
    z = P;
    while ( e > 0 ) do
    begin
        if ( e mod 2 == 1 ) then y = y·z mod n;
        e = e div 2;
        z = z^2 mod n;
    end;
    FastExp = y;
end.
```

3.12.3 Analyse av algoritmen

Kjøretiden til algoritmen er avhengig av antallet runder i while-løkka. Anta at både P,e og n har lengde m bit som tilfellet ofte er i RSA. Da går vi m ganger gjennom while-løkka. Inne i den er det en multiplikasjon mod n som utføres i gjennonsnitt halvparten av gangene og en kvadrering mod n. Siden både multiplikasjon, kvadreringer og divisjoner har samme kompleksitet, får vi kompleksiteten til rask eksponensiering til å være: $O(m \cdot 2M(m)) = O(m \cdot M(m))$, der $M(m)$ er tiden det tar å multiplisere to m-bits tall.

3.13 Konklusjon

Konklusjonen skulle være enkel. Av de algoritmene som er tatt med i denne delen viser det seg at de gode gamle måtene å addere, subtrahere, multiplisere og dividere på, er de beste for tall som er aktuelle for RSA-kryptosystemet. Når det gjelder multiplikasjon som jeg har sett mest på, ser det ut som at skolealgoritmen er best opptil 96 siffer uavhengig av tallsystem. Deretter overtar Splitt og hersk som er raskest opptil 290 siffer der den rekursive FFT overtar. FFT vil så være det beste valget helt opp til $\approx 2^{23}$ siffer, hvis vi har tilgang på dobbel presisjon på heltallsmultiplikasjon. Deretter overtar Schönhage-Strassen algoritmen.
Med stor sannsynlighet vil iterativ FFT være litt raskere enn den rekursive, og vil konkurere ut Splitt og hersk for et mindre antall siffer. Brukes grunntall 10^4 vil skolemultiplikasjon være raskest opp til ca. 380 desimalsiffer og brukes grunntall 10^9 raskest opp til ca. 850 desimal-siffer. Dvs. at det ennå er lang tid før man trenger å bry seg med andre algoritmer for multiplikasjon.
Når det gjelder RSA og rask eksponensiering, så er det ikke sikkert det er raskest å først multiplisere to tall, for så å redusere produktet modulo n. Det er mulig å gjøre det i en operasjon. Interesserte kan se i artikkelen Modular Multiplication without Trial Division av Peter L. Montgommery. I den senere tid har også CRT vært brukt i kombinasjon med eksponensiering for å øke hastigheten, men det har jeg ikke satt meg inn i.

Kompleksiteten til multiplikasjonsalgoritmene er som følger (i asymptotisk stigende rekkefølge):

FFT............................ $O(n \cdot \log n)$
Schönhage-Strassen.......... $O(n \cdot \log n \cdot \log\log n)$
Modulær multiplikasjon...... $O(n \cdot \log n \cdot \log n \cdot \log\log n)$
Splitt og Hersk................ $O(n^{\log_2 3})$
Skolealgoritmen.............. $O(n^2)$

Litteratur som er brukt i denne delen:

The Design and Analysis of Computer Algorithms
Aho, Hopcroft, Ullman. Addison Wesley 1974

The Computation of π to 29.360.000 Decimal Digits...
David H. Bailey. Math of Comp Jan.88 Vol.50 No.181

Computer Organization and Programming – VAX-11
Souhail El-Asfouri, O.Johnson, W.K.King. Addison Wesley 1984

Fundamentals of Computer Algorithms
Ellis Horowitz, Sartaj Sahni. Pitman 1978

The Art of Computer Programming : Seminumerical Algorithms
Donald E. Knuth . Addison-Wesley 2.Utgave 1981

Algorithms : Their Complexity and Efficiency
Lydia I. Kronsjö. Wiley 1979

Elements of Algebra and Algebraic Computing
J.D.Lipson. Benjamin/Cummings 1981

Modular Multiplication without Trial Division
Peter L. Montgommery. Math of Comp Apr.85 Vol.44 No.170

Prime Numbers and Computer Methods for Factorization
Hans Riesel, Birkhäuser 1985

Elementary Number Theory and its Applications 2. Utgave
Kenneth H. Rosen. Addison-Wesley 1988

Del 4
RSA – Andre algoritmer

I denne delen av oppgaven ser jeg på:

- *Største felles divisor*
- *Inverser modulo n*
- *Primtallstesting*

4.1 Største Felles divisor – Euclids algoritme

Største felles divisor til to heltall a og b, der ikke begge er 0, er det største heltall som deler både a og b og betegnes $\gcd(a,b)$ eller bare (a,b).

Eksempel $\gcd(12,28) = 4$, $\gcd(35,22) = 1$, $\gcd(31,0) = 31$ og $\gcd(12,6) = 6$

Teorem La a,b,c være heltall med $\gcd(a,b) = d$

1. $\gcd(a/d,b/d) = 1$
2. $\gcd(a + c \cdot b,b) = \gcd(a,b)$
3. a,b partall: $\gcd(a,b) = 2 \cdot \gcd(a/2,b/2)$
4. a partall og b odde: $\gcd(a,b) = \gcd(a/2,b)$

Til både RSA og Pohlig-Hellman trenger vi en algoritme for å avgjøre om to tall er innbyrdisk primiske, dvs. at de ikke har noen felles faktorer. For RSA gjelder dette e og $\phi(n)$) og for Pohlig-Hellman e og $p - 1$. Til dette kan vi bruke en algoritme som beregner største felles divisor. To heltall er innbyrdisk primiske hvis og bare hvis største felles divisor til tallene er lik 1.
Største felles divisor vil kun bli brukt under oppsett av kryptosystemene, og ved eventuelle senere endringer. Algoritmen blir derfor utført sjelden, og er ikke avhengig av å bli beregnet raskt. Den kanskje eldste og enkleste av algoritmene for å finne største felles divisor – Euclids algoritme er som regel mer enn rask nok.

4.1.1 Euclids algoritme

Euclids algoritme bruker gjentatte divisjoner for å finne største felles divisor mellom to tall a og b. Anta at algoritmen krever n divisjoner og at $R_n = a > b = R_{n-1}$ [39].
Algoritmen kan vi definere på følgende måte:

$$\begin{aligned} R_n &= R_{n-1} \cdot q_{n-1} + R_{n-2} \\ R_{n-1} &= R_{n-2} \cdot q_{n-2} + R_{n-3} \\ &\vdots \\ R_2 &= R_1 \cdot q_1 \quad + R_0 \\ R_1 &= R_0 \cdot q_0 \quad + 0 \end{aligned}$$

der $R_n > R_{n-1} > \ldots > R_1 > R_0 \geq 0$ og $q_{n-1},\ldots,q_1 \geq 1$, $q_0 \geq 2$.
Algoritmen stopper når $R_{-1} = 0$.
Da er $R_0 = \gcd(R_0,R_1) = \gcd(R_1,R_2) = \ldots = \gcd(R_{n-1},R_n) = \gcd(a,b)$

4.1.2 Analyse av algoritmen

Tiden algoritmen bruker er avhengig av n, og tiden vi bruker på hver divisjon. Det viser seg at de tallene som gir størst n er to påfølgende Fibonacci tall. Fibonacci tallene er definert ved: $F_0 = 1$, $F_1 = 1$ og $F_k = F_{k-1} + F_{k-2}$ for $k > 2$. Det viser seg at $\alpha = F_k/F_{k-1} = (1 + \sqrt{5})/2 \approx 1{,}618$, når k går mot uendelig. Det betyr at antall divisjoner er begrenset av $\log_\alpha a = 4{,}78 \cdot \log_{10} a$ eller $1{,}44 \cdot \log_2 a$. Algoritmen har dermed kompleksitet $O(n)$ for n-bits tall hvis, vi går ut fra at hver divisjon er $O(1)$.

Skal vi regne med store tall vil divisjonen i hver iterasjon av algoritmen ha kompleksitet > $O(1)$. Siden divisjon har samme kompleksitet som multiplikasjon kan vi derfor si at kompleksiteten til Euklids algoritme er $O(n \cdot M(n))$, der $M(n)$ er tiden det tar å multiplisere n-bits heltall.

Det viser seg imidlertid at i 99,97% av divisjonene er kvotienten $q < 5000$.
Dette siden sannsynligheten for $q = a$ er $\log_2\left(\frac{(a+1)^2}{(a+1)^2 - 1}\right)$ [40].
På en datamaskin med ordlengde 32 bit vil q ha enkel presisjon i bortimot 100% av divisjonene, slik at kompleksiteten til hver divisjon kan antas å være $O(n)$. Kompleksiteten til Euklids algoritme blir da $O(n^2)$, gitt at kvotientene har enkel presisjon.

Er kvotienten til en av divisjonene større en enkel presisjon blir kompleksiteten til denne divisjonen større enn $O(n)$. På den annen side kutter vi betraktelig ned på tallene ved store kvotienter, slik at antall divisjoner blir færre. Spørsmålet er nå om dette oppveier merarbeidet med vanskeligere divisjoner?

Vi har sett at påfølgende Fibonacci gir størst antall divisjoner. Det kan være interessant å se på divisjon på samme måten. Divisjon tar lengst tid når dividenden er dobbelt så lang som divisoren (skolealg.). Se på følgende eksempel:

---

[39] Ben Johnsen – Kryptografi, primtall og Riemanns hypotese
[40] The Art of Computer Programming: Seminumerical Algorithms – Donald Knuth s.352

a	=	**kvotient**	·	b	+	**rest**
11578298334	=	1	·	11578190732	+	107602
11578190732	=	107602	·	107602	+	328
107602	=	328	·	328	+	18
328	=	18	·	18	+	4
18	=	4	·	4	+	2
4	=	2	·	2	+	0

I dette tilfellet er antall divisjoner begrenset av $O(\log_2 n)$ for n-bits tall siden tallene halveres i lengde for hver divisjon. Det totale antall operasjoner er dermed begrenset av $O(\log_2 n \cdot M(n))$. Men siden tallene halveres blir divisjonene fort enklere og vi vi kan presse kompleksiteten enda lengre ned. Vi får:

$$\frac{M(n)}{2^0} + \frac{M(n)}{2^1} + \ldots + \frac{M(n)}{2^{\log_2 n}} = \sum_{i=0}^{\log_2 n} \frac{M(n)}{2^i} = 2 \cdot M(n) = O(M(n)) \text{ operasjoner.}$$

Kompleksiteten til Euklids algoritme ser dermed ut til å være $O(n^2)$, siden antall operasjoner synker for større kvotienter og færre divisjoner. Det er selvfølgelig ikke opplagt at kompleksiteten synker jevnt og trutt mellom $O(n^2)$ og $O(M(n))$, men det ser slik ut.

Jeg skal illustrere dette med et eksempel:
Gitt først to store påfølgende Fibonacci tall med ≈ 100 desimalsiffer.
Å redusere størrelsen på disse med ≈ 4 siffer krever $1{,}618^{\text{divisjoner}} = 10^4$. Vi får antall divisjoner til å ligge rundt 19, der hver divisjon krever rundt 100 operasjoner. Tilsammen i underkant av 2000 operasjoner.

Gitt nå 2 store tall med ≈ 100 siffer slik at divisjonen av det største på det minste gir oss kvotienten $q \approx 10^4$, dvs vi reduserer det største tallet med 4 siffer i en divisjon. En slik divisjon krever bare $m \cdot n - n^2 = 100 \cdot 96 - 96^2 \approx 400$ operasjoner, der $m = 100$ siffer og $n = 96$ siffer slik at q har 4 siffer.

Dette viser at vi vinner mye på store kvotienter, og det styrker troen på at kompleksiteten er størst for påfølgende Fibonacci tall, dvs. $O(n^2)$.

4.1.3 Ei prosedyre i Pascal

Ei enkel prosedyre for Euklids algoritme gitt to positive heltall $a,b > 0$.

```
Function Euclid ( a , b : integer ) : integer;
var r: integer;
begin
    repeat
        r = b;
        b = (a mod b);
        a = r;
    until (b = 0);
    Euclid = r;
end.
```

En av de raskeste algoritmene for å finne største felles divisor står beskrevet i [41]. Kompleksiteten til denne er $O(\log_2 n \cdot M(n))$. Algoritmen er rekursiv, og for hver rekursjon beregner den seg halveis nedover i restsekvensen $R_n, R_{n-1}, ..., R_0$ til Euclids algoritme. Dvs. i første rekursjon finner vi $R_{n/2}$, i andre finner vi $R_{n/4}$ osv. Den er egentlig beregnet for å beregne største felles divisor for polynomer, men kan med små modifikasjoner brukes på heltall også.

4.2 Inverser modulo n

Til både RSA og Pohlig-Hellman trenger vi en algoritme som finner inverser modulo henholdvis $\phi(n)$ og $(p - 1)$. Grunnen til dette er at dekrypteringsnøkkelen d er en invers av e modulo $\phi(n)$ eller $(p - 1)$.

Gitt et heltall a med $\gcd(a,m) = 1$ så kalles en løsning til kongruensen $a \cdot x \equiv 1 \pmod{m}$ for en **invers** modulo m.

Den vanlige måten for å finne inverser på er ved å bruke Euklid´s utvidede algoritme. Denne algoritmen har samme kompleksitet som Euklid´s vanlige algoritme. Forskjellen er kun en liten konstant. Algoritmen finner lineærkombinasjonen $a \cdot x - m \cdot y = g$, der x er den inverse av a og g er største felles divisor mellom a og m. Siden $\gcd(a,m) = 1$ vil g alltid være lik 1.

[41] The Design and Analysis of Computer Algorithms - Aho/Hopcroft/Ullman s.302

```
Procedure UtvidetEuklid ( a , m , x , y , g : integer );
var r : integer;
begin
    x = 1; x1 = 0;
    y = 0; y1 = 1;
    repeat
        q = a/b;

        r = b;
        b = a – q·b;
        a = r;

        r = x1
        x1 = x – q·x1;
        x = r;

        r = y1
        y1 = y – q·y1;
        y = r;
    until (b == 0);
    g = a
end.
```

To andre metoder for å beregne inverser modulo et sammensatt tall m og et primtall q [42]:

Gitt gcd(a,m) = 1, da er $x \equiv a^{\phi(m)-1}$ (mod m) en invers av a (mod m).
Dette siden $a \cdot a^{\phi(m)-1} \equiv a^{\phi(m)} \equiv 1$ (mod m).

Gitt gcd(a,q) = 1, q primtall. Da er $x \equiv a^{q-2}$ (mod q) en invers av a (mod q)
Dette siden $a \cdot a^{q-2} \equiv a^{q-1} \equiv 1$ (mod q).
Denne siste metoden vil ikke være til nytte for oss siden både $\phi(n)$ og $(p - 1)$ er sammensatte tall
Tidsforbruket for disse metodene er lik tiden det tar å kryptere ei klartekstblokk med RSA/Pohlig-Hellman. Algoritmene har dermed samme kompleksitet som rask eksponensiering, dvs. $O(m \cdot M(m))$, der $M(m)$ er tiden det tar å multiplisere to m-bits tall. Den er mao. noe tregere enn Euklid´s utvidede algoritme.

Om selve problemet med å finne inverser har samme kompleksitet som problemet største felles divisor har jeg ikke funnet noe skriftlig om, men det er ikke helt utenkelig at så er tilfelle. En av de raskeste algoritmene for å finne inverser står beskrevet i [43]. Kompleksiteten til denne er $O(\log m \cdot M(m))$.

---

[42] Elementary Number Theory and its Applications – K.Rosen s.169 og 182
[43] The Design and Analysis of Computer Algorithms - Aho/Hopcroft/Ullman s.310

4.3 Primtallstesting (Stokastiske algoritmer)

Algoritmer for primtallstesting er nødvendig for å finne primtall til RSA og Pohlig-Hellman metoden. For å finne (ca.) 100-sifrete primtall til RSA trekker man ut tilfeldige odde heltall i intervallet, f.eks fra 10^{98} til 10^{102}. Så kjører man en primtallstest på hver av de tilfeldig tallene for å finne ut om de er primtall eller ikke. Så snart vi har funnet to primtall er vi ferdige. I gjennomsnitt vil man på denne måten finne et primtall etter rundt 115 forsøk. Dette siden sannsynligheten for at et tilfeldig valgt odde 100-sifret heltall er et primtall er $\approx 2/\log_e 10^{100}$ ifølge primtallsteoremet [44]. For Pohlig-Hellman metoden bør primtallet p ha omtrent 200 siffer. Et slikt tall vil man i gjennomsnitt finne etter 230 forsøk.
For å finne kryptologisk sikre primtall, må primtallene konstrueres på en spesiell måte. Primtallstestingen blir imidlertid uforandret, slik at vi kan bruke de samme algoritmene for primtallstesting. Hvordan man konstruerer kryptologisk sikre primtall vil bli tatt opp i konklusjonen til del 5.

Jeg vil først ta for meg to enkle stokastiske algoritmer. Deretter vil jeg se litt på den asymptotiskt raskeste deterministiske algoritmen – Rumely Adleman algoritmen.
Disse algoritmene vil først og fremst bli skissert, ikke spesielt godt analysert.

4.3.1 Rabin's Sannsynlighets Primtallstest [45]

La n være et positivt heltall med $n - 1 = 2^s t$, der s er et positivt heltall ≥ 0 og t er et odde positivt heltall. Vi sier at n passerer **Miller's test** for basen b hvis $b^t \equiv 1 \pmod{n}$ eller $b^{2^j t} \equiv -1 \pmod{n}$ for en eller annen j med $0 \leq j \leq s - 1$.

Teorem Hvis n er primtall og b er et positivt heltall med $n \nmid b$, så passerer n Miller's test for basen b.

Hvis n er sammensatt og passerer Miller's test for basen b, kaller vi n et **sterkt pseudoprimtall** til basen b.

Selv om sterke pseudoprimtall er sjeldne, er det likevel uendelig mange av dem.

Teorem Hvis n er et odde sammensatt positivt heltall, så passerer n Miller's test for maksimalt $(n - 1)/4$ baser b med $1 \leq b \leq n - 1$.

Dette kan vi nå sette sammen til en primtallstest (egentlig en sammensatt tall test). La n være et positivt heltall. Velg k forskjellige tilfeldige positive heltall $< n$, og utfør Miller's test med disse k tallene som baser.

[44] Elementary Number Theory and its Applications – K.Rosen s.233
[45] Elementary Number Theory and its Applications – K.Rosen s.177

Dette gjør vi ved å beregne følgende:

$y = b^t \pmod{n}$. Hvis $y = \pm 1$ fortsett, ellers er n sammensatt.
$y = b^{2t} \pmod{n}$. Hvis $y = -1$ fortsett, ellers er n sammensatt.
$y = b^{4t} \pmod{n}$. Hvis $y = -1$ fortsett, ellers er n sammensatt.
$\vdots$
$y = b^{2^j t} \pmod{n}$. Hvis $y = -1$ har n passert Miller´s test for basen b, ellers er n sammensatt.

Et eksempel på Rabin´s primtallstest:
$n = 1373653$, $n - 1 = 2^2 \cdot 343413$.
Vi trekker en "tilfeldig" base 3 og tester:
$3^{343413} \equiv 1 \pmod{1373653}$
$3^{343413 \cdot 2} \equiv 1 \pmod{1373653}$, dvs. at 1373653 har passert Miller´s test for basen 3.
Vi trekker en ny tilfeldig base 13 og tester:
$13^{343413} \equiv 979048 \pmod{1373653}$ og vi vet dermed at n er sammensatt.

Hvis n er sammensatt, så er sannsynligheten for at n passerer Miller´s test for alle k basene $(1/4)^k$. Lar vi $k = 100$, dvs. vi trekker ut 100 baser tilfeldig mellom 1 og n og utfører testen på hver av basene, så er sannsynligheten for at et sammensatt tall passerer alle testene mindre enn 10^{-60}. Denne sannsynligheten er utrolig liten. Så liten at det er større sannsynlighet for at datamaskinen har regnet feil enn at et sammensatt tall har passert alle 100 basene. Selv om Rabin´s primtallstest ikke beviser at et tall er primtall, så er det ikke langt ifra.
Kjøretiden til Rabin´s primtallstest er $O(\log_2 n)^3$, siden vi utfører maksimalt $\log_2 n$ eksponensieringer som igjen krever $(\log_2 b)^2$ operasjoner.
Er den generaliserte Riemann Hypotesen sann, slik som mange matematikere tror, så finnes det for alle positive sammensatte heltall n en base b med $b < 70 \cdot (\log_2 n)^2$, slik at n ikke passerer Miller´s test for basen b. Vi kan da bevise om n er et primtall ved å bruke $O((\log_2 n)^5)$ bit operasjoner.

4.3.2 Solovay-Strassen´s Sannsynlighets Primtallstest [46]

La p være et odde primtall og a et heltall med $p \nmid a$.
Legendre symbolet $\left(\frac{a}{p}\right)$ er definert ved $\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{hvis a er en kvadratisk rest (mod } p) \\ -1 & \text{hvis a ikke er en kvadratisk rest (mod } p) \end{cases}$

Euler´s kriterium: La p være et odde primtall, og la a være et positivt heltall slik at $p \nmid a$.
Da er $\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}$.

---

[46] Elementary Number Theory and its Applications – K.Rosen s.341

La n være et positivt heltall med primtallsoppspaltning $n = p_1^{t_1} p_2^{t_2} \ldots p_m^{t_m}$, og la a være et heltall relativt primisk med n.
Da er **Jacobisymbolet** $\left(\frac{a}{n}\right)$ definert ved $\left(\frac{a}{n}\right) = \left(\frac{a}{p_1^{t_1} p_2^{t_2} \ldots p_m^{t_m}}\right) = \left(\frac{a}{p_1}\right)^{t_1} \left(\frac{a}{p_2}\right)^{t_2} \ldots \left(\frac{a}{p_m}\right)^{t_m}$ der symbolene på høyre siden av likheten er Legendresymboler.
La a og b være relativt primisk med $a > b$. Da kan Jacobisymbolet beregnes ved å bruke $O((\log_2 b)^3)$ operasjoner.

Euler´s kriterium kan brukes til å sjekke om et tall n er primtall. La b være et positivt heltall slik at $\gcd(b,n) = 1$ og se på kongruensen $b^{(n-1)/2} \equiv \left(\frac{b}{n}\right) \pmod{n}$. Hvis denne ikke stemmer er n sammensatt.

Et odde sammensatt positivt heltall n som tilfredstiller kongruensen $b^{(n-1)/2} \equiv \left(\frac{b}{n}\right) \pmod{n}$, der b er et positivt heltall, kalles et **Euler pseudoprimtall** til basen b.

Teorem Hvis n er et sterkt pseudoprimtall til basen b, så er n et Euler pseudoprimtall til samme base. Det omvendte er nødvendigvis ikke tilfelle.

Av dette kan vi lage en primtallstest (dette er egentlig også en sammensatt tall test).
La n være et positivt heltall. Velg tilfeldig k positive heltall $b_1, b_2, \ldots, b_k$ i området $1,2,\ldots,n-1$. For hver av disse tallene finn ut om $b_j^{(n-1)/2} \equiv \left(\frac{b_j}{n}\right) (mod\ n)$
Hvis en av kongruensene ikke stemmer er n sammensatt, og hvis n er sammesatt så er sannsynligheten for at alle kongruensene stemmer $\leq (1/2)^k$.

Siden alle sterke pseudoprimtall til basen b er et Euler pseudoprimtall til samme base, passerer flere sammensatte heltall denne testen enn Miller´s test. Dette kan vi kompensere ved å utføre testen flere ganger. Likevel krever begge testene $O((\log_2 n)^3)$ bit operasjoner.

På grunn av enkelheten til de to siste algoritmene er de raske nok til å finne primtall store nok både til RSA og Pohlig-Hellman innen rimelig tid.

4.4 Rumely-Adlemans primtallsteste algoritme [47]

I 1980 oppfant Leonard Adleman og Robert Rumely en ny primtallstest som kan trekke ut mer informasjon fra pseudoprimtallstestene, enn bare at tallet passerer eller ikke. Denne ekstra informasjon kan brukes til å bevise primalitet. Antall mulige divisorer blir kuttet ned til et minimum slik at vi kan undersøke dem enkeltvis.
Primtallstesten har en kompleksitet som er "nesten" polynomiell. Kjøretiden er begrenset av $O((\log n)^{c \cdot \log\log\log n})$ for et positivt heltall n og en konstant c. I tillegg vil den bevise at n er et primtall hvis så er tilfelle, ikke bare gi en viss sannsynlighet for det. Algoritmen skal også være rimelig enkel å implementere på en datamaskin.
Siden endel av primtallstesten bygger på matematikk som forfatteren er ukjent med, vil jeg kun prøve å gi en beskrivelse av idéene bak algoritmen.

Velg de minste positive heltall $s,t > 0$ som tilfredstiller følgende krav:
1. t er kvadratfritt.
3. $s > \sqrt{n}$ der $s = \prod_{\substack{(q-1)|t \\ q \text{ primtall}}} q$

I et program som utfører algoritmen, må vi først bestemme oss for hvor store tall vi vil forsøke oss på. Så finner vi s og t slik at betingelsene er oppfylt for alle tall opp til grensen vår. Skal vi se på heltall $\leq 10^{213}$ kan vi f.eks. velge $t = 55440$ slik at $s \approx 4{,}920 \cdot 10^{106}$.

Primfaktorene til t kalles de **Initiale** primtallene, mens primfaktorene til s, dvs. primtallene q med $q - 1 \mid t$ kalles de **Euklidske** primtallene. Sjekk at ikke n er delelig med verken de initsielle eller de Euklidske primtallene. Hvis n er sammensatt så har n en primfaktor $r \leq \sqrt{n}$. Siden produktet av de Euklidske primtallene er større enn $\sqrt{n}$, kan vi finne r ved å bruke CRT hvis vi vet hver av restene $r \pmod{q}$.
La g_q betegne ei primitiv rot (mod q), og $\mathrm{Ind}_q(r)$ er minste positive heltall slik at $r \equiv g_q^{\mathrm{Ind}_q(r)} \pmod{q}$.
Det betyr at å vite resten $r \pmod{q}$ er ekvivalent med å vite $\mathrm{Ind}_q(r)$.
Vi har at $1 \leq \mathrm{Ind}_q(r) \leq q - 1$, der $q - 1$ er et kvadratfritt produkt av initiale primtall som deler t.
Det betyr at vi kan finne $\mathrm{Ind}_q(r)$ ved å bruke CRT, hvis vi vet hver av restene $\mathrm{Ind}_q(r) \pmod{p}$, for hvert primtall $p \mid (q - 1)$. Hvis $\mathrm{Ind}_q(r)$ kan beregnes for hvert par av primtall p,q med $p \mid (q - 1) \mid t$, så kan vi beregne r den hypotetiske primfaktoren til n.

Vi skal nå forfølge denne idéen for spesialtilfellet $p = 2$
La q være et odde Euklidsk primtall. $\mathrm{Ind}_q(r) \pmod{2}$ kan bare ta på seg to verdier nemlig 0 eller 1. Hvis $\mathrm{Ind}_q(r) \pmod{2}$ er 0, så er r et kvadrat mod q, ellers ikke. Det betyr at $\left(\frac{r}{q}\right) = (-1)^{\mathrm{Ind}_q(r)}$ for hvert odde Euklidsk primtall. Hvis vi kunne beregne $\left(\frac{r}{q}\right)$, så ville vi kunne beregne $\mathrm{Ind}_q(r) \pmod{2}$.

[47] Recent Developments in Primality Testing – Carl Pomerance s.97

Men hvordan kan vi gjøre det uten å kjenne r? Det kan vi ikke! Men nøkkelidéen er at vi kan finne relasjoner mellom de forskjellige Legendre symbolene, slik at hvis bare et av dem blir gjettet, så kan alle de andre beregnes ut fra det. Som Pomerance uttrykker det: det er denne "eksplosjonen av kunnskap" som er forklaringen bak hurtigheten til algoritmen.

På grunn av loven om kvadratisk resiprositet kan vi "snu" Legendre symbolene. Vi har at $q - 1$ er kvadratfritt $\Rightarrow 4 \nmid (q-1) \Rightarrow -q \equiv 1 \pmod 4 \Rightarrow \left(\frac{r}{q}\right) = \left(\frac{-q}{r}\right)$.

Så definerer vi "mock restsymbolet" som følger:

$$\left\langle\frac{-q}{n}\right\rangle = \begin{cases} \pm 1 \equiv (-q)^{\frac{(n-1)}{2}} \pmod n, \text{ hvis kongruensen er oppfylt} \\ \text{ellers udefinert} \end{cases}$$

Hvis dette symbolet er udefinert, så er n sammensatt, og vi kan avbryte algoritmen. Hvis $\left\langle\frac{-q}{n}\right\rangle$ er definert, så har n tilfredstilt en pseudoprimtallstest med grunntall $-q$.

Anta nå at $\left\langle\frac{-q}{n}\right\rangle$ er definert for alle odde Euklidske primtall.

La oss videre anta at minst et av $\left\langle\frac{-q}{n}\right\rangle$-ene er -1, f.eks $\left\langle\frac{-q_0}{n}\right\rangle = -1$. Denne antagelsen byr sjelden på problemer i praksis, selv om teorien ikke gir oss noen grunn for at q_0 skulle eksistere.

Da kan vi beregne tall $m_q = 0{,}1$ for hver odde q slik at $\left\langle\frac{-q_0}{n}\right\rangle^{m_q} = \left\langle\frac{-q}{n}\right\rangle$.

Poenget er nå at dette holder dersom vi erstatter n med r, dvs $\left\langle\frac{-q_0}{r}\right\rangle^{m_q} = \left\langle\frac{-q}{r}\right\rangle$.

Vi får også at $\left(\frac{-q_0}{r}\right)^{m_q} = \left(\frac{-q}{r}\right)$.

Vi har fremdeles ikke klart å beregne $\left(\frac{-q}{r}\right)$ men vi har beregnet tallene m_q.

Da er det bare to muligheter: Hvis $\left(\frac{-q_0}{r}\right) = 1$ så er $\left(\frac{-q}{r}\right) = 1$ for alle q.

Hvis $\left(\frac{-q}{r}\right) = -1$, så er $\left(\frac{-q}{r}\right) = (-1)^{m_q}$ for alle q. Teoretisk sett kan det skje at $\left\langle\frac{-q}{r}\right\rangle = 1$ for alle q, men det finnes en metode for å komme rundt dette.

For et initialt primtall $p \geq 3$, må vi generalisere Legendresymbolet og den kvadratiske resiprositetsloven. Denne generaliseringen er potensrestsymbolet og de høyere resiprositetslovene. Med denne matematikken kan vi fortsette på samme måte som for $p = 2$.

Etter at vi har gjort dette for hvert initialt primtall, så prøver vi de forskjellige mulige verdier av de forskjellige potensrestsymbolene. Hver gang får vi noen få mulige divisorer av n, som vi så undersøker. Finner vi en divisor, så er n sammensatt, ellers er n et primtall.

Dette er fremgangsmåten som ble brukt av Adleman, Pomerance og Rumely i deres versjon av algoritmen. Senere er det kommet flere andre versjoner av algoritmen. Den videre matematikken har jeg ikke satt meg inn i, og vil derfor nøye meg med å henvise interesserte videre til artikkelen av Carl Pomerance.

Litteratur som er brukt til denne delen

On Distinguishing Prime Numbers from Composite Numbers
Adelman, Pomerance og Rumely, Annals of Math 117 1983

The Design and Analysis of Computer Algorithms
Aho, Hopcroft, Ullman, Addison Wesley 1974

Kryptografi, primtall og Riemanns hypotese
Ben Johnsen, Normat nr.1 1986

Implementation of a New Primality Test
H. Cohen og A. K. Lenstra, Math of Comp Vol.48 no.177 Jan.87

Primality Testing and Jacobi Sums
H. Cohen og H. W. Lenstra Jr., Math of Comp Vol.42 no.165 Jan.84

The Art of Computer Programming : Seminumerical Algorithms
Donald E. Knuth, Addison-Wesley 2.Utgave 1981

Primality and Cryptography
Evangelos Kranakis, Wiley 1986

Elements of Algebra and Algebraic Computing
J.D.Lipson, Benjamin/Cummings 1981

Recent Developments in Primality Testing
Carl Pomerance, Math. Intelligenser 3 1981

Prime Numbers and Computer Methods for Factorization
Hans Riesel, Birkhäuser 1985

Elementary Number Theory and its Applications
Kenneth H. Rosen, Addison-Wesley 2. Utgave 1988

Del 5
Algoritmene som sikkerheten til RSA avhenger av

Denne delen omhandler:

- *Generelt om faktorisering*
- *Pollard's rho algoritme*
- *Pollard's p-1 algoritme*
- *Elliptisk kurve faktorisering*
- *Generelt om kvadratiske rest metoder*
- *Delbrøkoppspalting*
- *Kvadratisk sil*
- *Den diskrete Logaritme*

5.1 Innledning

Sikkerheten til RSA avhenger først og fremst av kompleksiteten til de to algoritmene faktorisering og den diskrete logaritme. La oss se litt på forholdet mellom det å faktorisere store tall og det å knekke koden:

Gitt $n = p \cdot q$, der p og q er primtall og e er krypteringsnøkkelen. Anta $\gcd(e, \phi(n)) = 1$ og $d = \mathrm{Inv}(e \bmod \phi(n))$. For å finne d må man kjenne $\phi(n) = \phi(pq) = (p-1)\cdot(q-1)$ som forutsetter at man kjenner faktoriseringen til n. Kjenner vi $\phi(n)$ og n men ikke faktoriseringen til n, kan vi beregne den ved å sette opp følgende ligninger [48]:

$p + q = n - \phi(n) + 1$ og $p - q = \sqrt{(p+q)^2 - 4n}$.

Vi får da at $p = \dfrac{(p+q)+(p-q)}{2}$ og $q = \dfrac{(p+q)-(p-q)}{2}$.

Det er dermed like vanskelig å finne $\phi(n)$ som å faktorisere n. Det er imidlertid ikke bevist at det å knekke koden er ekvivalent med å faktorisere n.

Hvis man kjenner d, men ikke $\phi(n)$, kan n faktoriseres siden $e \cdot d - 1$ er et multiplum av $\phi(n)$. Det finnes raske algoritmer for å faktorisere n gitt et multiplum av $\phi(n)$.

En annen måte å knekke koden på er å finne dekrypteringsnøkkelen d ved å beregne den diskrete logaritme. Det finnes forskjellige algoritmer for å finne diskrete logaritmer til en gitt base modulo et primtall p, dvs beregne x slik at $b^x \equiv a \pmod{p}$. Den raskeste algoritmen trenger omtrent $\exp\left(\sqrt{\log p \operatorname{loglog} p}\right)$ bit operasjoner, dvs. omtrent det samme antall operasjoner som trengs for å faktorisere et tall av samme størrelse. I RSA-systemet er moduloen n et sammensatt tall. Om det har noen innvirkning på kompleksiteten til den diskrete logaritme har jeg ikke funnet stoff om. Sikkerheten til Pohlig-Hellman metoden baserer seg utelukkende på at det er vanskelig å beregne diskrete logaritmer for store tall.

---

[48] Elementary Number Theory and its Applications – K.Rosen s.233

5.2 Faktorisering generelt

La oss si at vi skal faktorisere et 100 sifret tall n, som er produktet av 2 ukjente primtall p og q, med $p < q$. Vi har da at $p < \sqrt{n} < q$. Oppgaven blir nå å finne primtallet p blandt alle tallene mindre enn $\sqrt{n}$. Det betyr at vi må prøve å skille ut primfaktoren p fra $\approx 10^{50}$ andre tall. Vi bør derfor finne på noe bedre enn forsøksdivisjon.

Et tankeeksperiment: Vi konstruerer en faktoriseringsalgoritme etter følgende prinsipp: Del alle tallene mellom 2 og $\sqrt{n}$ i 2 like store mengder. Konstruer en funksjon Faktor som uten å bruke "nevneverdig" tid returnerer verdien JA hvis primtallet p er i mengden og NEI hvis det ikke er det. Så bruker vi funksjonen på den ene av de to mengdene. Får vi JA deler vi denne mengden opp i 2 like store deler som vi igjen kjører testen på. Får vi NEI deler vi den andre mengden opp osv... Med en slik algoritme kan vi faktorisere n ved å bruke $\log_2\sqrt{n}$ iterasjoner, som for et 100 sifret tall blir rundt 166.

I virkeligheten derimot, er det ingen som har den minste anelse om hvordan man kan konstruere en slik funksjon. Det nærmeste man hittil har kommet er ved å bruke $\exp\left(\sqrt{\log n \operatorname{loglog} n}\right)$ operasjoner. For et 100 sifret tall blir det rundt $k \cdot 10^{15}$ operasjoner, der k er en konstant. Matematikere har i flere hundre år prøvd å finne effektive faktoriseringsalgoritmer uten helt å lykkes. Likevel har det skjedd store fremskritt i den siste tiden på grunn av utviklingen av raske datamaskiner, kombinert med nye og bedre algoritmer. Interessen for faktorisering steg betraktelig etter at Rivest, Shamir og Adleman "fant opp" RSA-kryptosystemet i 1978. Sikkerheten til dette systemet baserer seg bl.a. på at det er vanskelig å faktorisere store tall.

Faktoriseringsalgoritmer deles ofte inn i to grupper. Vi har de algoritmene som egner seg best til å faktorisere tilfeldige tall, og de som egner seg best til å faktorisere tall på en eller annen spesiell form, dvs. konstruerte tall.
Hurtigheten til den første gruppen er ofte avhengige av størrelsen på faktorene, mens i den andre gruppen er det mer tilfeldig i hvilken rekkefølge faktorene blir oppdaget i. Man kan sette en datamaskin på å faktorisere et partall, og etter timer med beregninger vil den skrive ut tallet 2 som faktor. Lenstras Elliptisk kurve metode er et eksempel fra den første gruppen, mens Kvadratisk sil er fra den andre gruppen.
Når det gjelder størrelsen på tall som man pr. i dag (1.1-90) kan faktorisere, så ligger den på litt i overkant av 100 desimalsiffer [49]. Tidsforbruket for slike faktoriseringer er omtrent 1 år med en 20 mip prosessor. I praksis faktoriseres så store tall på distribuerte datasystemer. I april 89 faktoriserte Lenstra og Manasse et 106 sifret tall. Det gjorde de ved å bruke 80 Firefly arbeidstasjoner, i tillegg til å "låne" PC-er og andre maskiner over hele verden via elektronisk post. Arbeidet som ble utført er estimert til tilsvarende 100 år på en 1 mip prosessor. Det kan nevnes som sammenligning at en Macintosh SE har en regnekraft på nesten 1 mip.

---

[49] Factoring – Carl Pomerance s.28

5.3 Pollard´s Rho algoritme

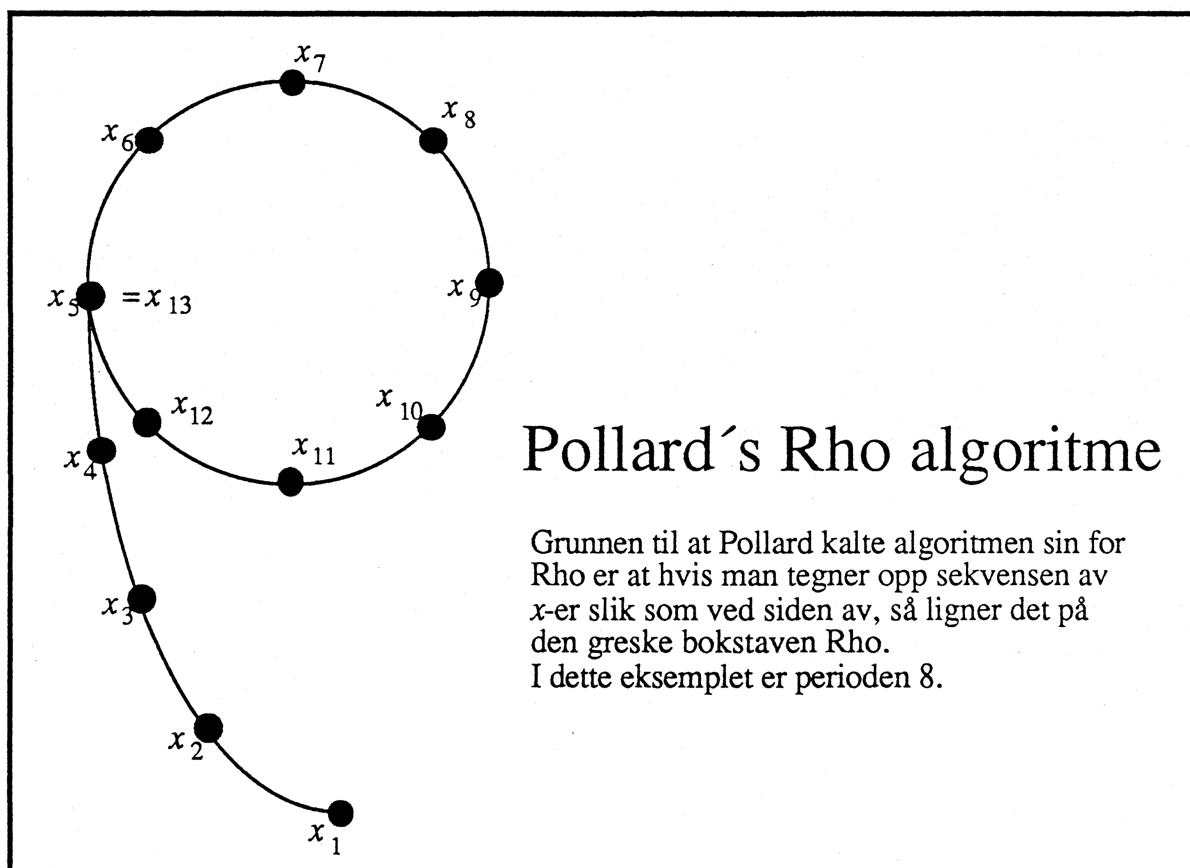
5.3.1 Beskrivelse av algoritmen [50]

Ideen bak denne metoden er analogt til det klassiske bursdagsproblemet. Hvor mange personer må man samle i et rom for at det skal være mer enn 50% sjanse for at minst to av dem har bursdag på samme dag. Målet med Rho algoritmen er å finne to forskjellige heltall x_i og x_k som er i samme restklasse modulo den ukjente primfaktoren p. Finner vi to slike tall kan beregne $\gcd(x_i - x_k, n)$ og håpe på ei ikketriviell løsning.

Det første vi gjør er å konstruere en funksjon f som er lett å beregne fra $\mathbf{Z}/n\mathbf{Z}$ til seg selv. Her kan vi velge et enkelt polynom med heltallskoeffisienter slik som $f(x) = x^2 + 1$. Så velger vi en startverdi a og beregner suksessive iterasjoner av f: $x_0 = a$, $x_1 = f(x_0)$, $x_2 = f(f(x_0))$, $x_3 = f(f(f(x_0)))$ osv. Dvs. $x_{i+1} = f(x_i)$, $i = 0,1,2,...$

Vi kan nå foreta sammenligninger mellom de forskjellige x_i-ene, og håpe at vi finner to som er i forskjellige restklasser modulo n, men som er i samme restklasse modulo en eller annen ikketriviell divisor til n. Med det samme vi finner slike x_i og x_k, har vi at $\gcd(x_i - x_k, n)$ er lik en ekte divisor til n, og vi er ferdig.

Det er viktig å velge et polynom f som avbilder $\mathbf{Z}/n\mathbf{Z}$ til seg selv på en "tilfeldig" måte. $f(x)$ må ikke være et linært polynom og avbildningen bør ikke være injektiv.



Pollard´s Rho algoritme

Grunnen til at Pollard kalte algoritmen sin for Rho er at hvis man tegner opp sekvensen av x-er slik som ved siden av, så ligner det på den greske bokstaven Rho.
I dette eksemplet er perioden 8.

Skal vi beregne største felles divisor av $(x_i - x_k)$ og n for $k = 0, ..., i - 1$, blir algoritmen raskt langsom når i vokser. Det er imidlertid nok å beregne største felles divisor en gang for

[50] A Course in Number Theory and Cryptography – Neal Koblitz s.

hver k. Dette virker, siden så snart det eksisterer en k_0 og i_0 med $x_{k_0} \equiv x_{i_0} \pmod p$ for $p \mid n$, så vil $x_i \equiv x_k \pmod p$ for alle par av indekser i,k som har den samme differansen $i - k = i_0 - k_0$. Dette ser vi hvis vi setter $i = i_0 + m$ og $k = k_0 + m$ og bruker polynomet f på begge sider av kongruensen $x_{i_0} \equiv x_{k_0} \pmod p$ m ganger.

Vi kan nå beregne x_i for alle i som følger. Hvis i er et $(h + 1)$-bits tall, la k være det største h-bits tall dvs. $k = 2^h - 1$. Vi sammenligner nå x_i med x_k, dvs. vi beregner $\gcd(x_i - x_k, n)$ og håper på en ikketriviell faktor. I motsatt fall prøver vi $i + 1$. Fordelen med denne metoden er at vi bare beregner en største felles divisor for hver i, mens bakdelen er at vi sannsynligvis ikke oppdager første gangen vi får en i_0 der $\gcd(x_{i_0} - x_{k_0}, n) = p$, $1 < p < n$. Med denne metoden vil vi f.eks oppdage at $x_1 \equiv x_6 \pmod p$ når vi sammeligner x_{13} med x_7.

En litt annen måte å gjøre sammenligningen på er ved å lage to sekvenser som har samme funksjon og startverdi, men der den ene går dobbelt så fort som den andre.

Vi får: $x_0 = a$, og $x_1 = f(x_0)$, $x_2 = f(x_1)$...

$y_0 = a$, og $y_1 = f(f(y_0))$, $y_2 = f(f(y_1))$... Dvs. $y_i = x_{2i}$.

Vi kan nå beregne $\gcd(y_i - x_i, n)$ for alle i, og håpe på ekte divisorer av n.

Når vil vi oppdage at vi har en x_j og en x_k som er kongruent (mod p), $j < k$?

Siden x_i hele tiden sammenlignes med x_{2i} for $i > 0$, vil vi finne p første gang x får indeks $(j + m)$ slik at $2(j + m) = (k + m + b\lambda)$ der $m,b \geq 0$ og minimale. λ er sykellengden, dvs. $k - j$ når $x_j \equiv x_k \pmod p$.

Hvis $x_1 \equiv x_6 \pmod p$ vil det oppdages når vi finner at $x_{1+4} \equiv x_{6+4} \pmod p$, og $x_{17} \equiv x_{19} \pmod p$ vil oppdages når $x_{17+1} \equiv x_{19+1+8\cdot 2} \pmod p$.

Fordelen med denne versjonen er at vi slipper å lagre mer enn to verdier, mens ulempen er at vi må beregne x_i-ene to ganger.

For å få den til å gå raskere kan vi akkumulere opp $(y_i - x_i)$-ene, og så beregne største felles divisor en gang iblant, f.eks etter K iterasjoner. K velges ofte en plass mellom 10 og 100.

Vi får da $Q_m \equiv \prod_{i=1+K\cdot m}^{K+K\cdot m} (y_i - x_i) \pmod n$, og vi kan beregne $\gcd(Q_m, n)$ for $m \geq 0$.

Når vi beregner største felles divisor til Q_m og n kan vi få 3 forskjellige resultater.

1:	Fortsett
$1 < p < n$:	Vi har funnet en ekte faktor og er ferdig.
0:	Her har vi to muligheter:

1. Vi kan ha passert ei løsning og må gå tilbake.
2. Polynomet f og/eller startverdien a bør skiftes ut.

Et viktig spørsmål nå er hvor mange iterasjoner algoritmen må utføre før vi har to iterasjoner x_i og x_k slik at $\gcd(x_i - x_k, n) = p$, $1 < p < n$.

5.3.2 Et eksempel på Pollard's Rho algoritme:

$n = 11009$. Vi velger startverdi $a = 2$ og et polynom $f(x) = x^2 + 1$ og får sekvensen:

$x_0 = 2$	$y_0 = 2$		
$x_1 = 5$	$y_1 = 26$	$\gcd(x_1 - y_1, n) = 1$	
$x_2 = 26$	$y_2 = 6961$	$\gcd(x_2 - y_2, n) = 1$	
$x_3 = 677$	$y_3 = 5842$	$\gcd(x_3 - y_3, n) = 1$	
	$\vdots$		
$x_8 = 299$	$y_8 = 4389$	$\gcd(x_8 - y_8, n) = 1$	
$x_9 = 1330$	$y_9 = 5370$	$\gcd(x_9 - y_9, n) = 101$	Vi har funnet en ekte faktor!!
$x_{10} = 7461$	$y_{10} = 574$	$\gcd(x_{10} - y_{10}, n) = 1$	
	$\vdots$		
$x_{15} = 5589$	$y_{15} = 8095$	$\gcd(x_{15} - y_{15}, n) = 1$	
$x_{16} = 4389$	$y_{16} = 1991$	$\gcd(x_{16} - y_{16}, n) = 109$	
$x_{17} = 8581$	$y_{17} = 4389$	$\gcd(x_{17} - y_{17}, n) = 1$	
$x_{18} = 5370$	$y_{18} = 5370$	$\gcd(x_{18} - y_{18}, n) = 11009$	

Vi ser at i 9. iterasjon finner vi en faktor $p \mid n$. Resten av sekvensen er tatt med for å vise problemene som kan oppstå ved å akkumulere $(x_i - y_i)$-ene. La oss si at vi samler opp 8 stykker, for så å beregne største felles divisor. I eksemplet får vi 1 til svar etter iterasjon 1 – 8 og 0 til svar etter 9 – 16. Løsningen er å gå tilbake til iterasjon 9, for så å beregne største felles divisor for hver iterasjon derfra.

5.3.3 Kjøretid

Hvor lang må en tilfeldig sekvens av heltall (mod p) være for at minst to av dem skal være kongruente (mod p) med en gitt sannsynlighet? La q være lengden til en slik sekvens. Sannsynligheten for at ingen av tallene er kongruente er [51]:

$$\left(1 - \frac{1}{p}\right) \cdot \left(1 - \frac{2}{p}\right) \cdot \ldots \cdot \left(1 - \frac{q-1}{p}\right) = S.$$

Den venstre siden av uttrykket er $\approx \left(1 - \frac{q}{2p}\right)^{q-1} \approx e^{-q(q-1)/2p}$ ifølge H.Riesel.

For å få dette til å være lik S må $\frac{q(q-1)}{2p} = ln\left(\frac{1}{S}\right) \Rightarrow q = \frac{1 + \sqrt{1 + 8p \cdot ln\left(\frac{1}{S}\right)}}{2} =$
$\sqrt{\frac{1}{4} + 2p \cdot ln\left(\frac{1}{S}\right)} + \frac{1}{2} \approx \sqrt{2p \cdot ln\left(\frac{1}{S}\right)} = \sqrt{2ln\left(\frac{1}{S}\right)} \cdot \sqrt{p}.$

Det vi er interessert i er imidlertid hvor lang sekvensen må være for at minst to av dem skal være kongruente (mod p) med en gitt sannsynlighet. Denne sannsynligheten blir $(1 - S)$, og lengden til en slik sekvens blir dermed $\sqrt{2ln\left(\frac{1}{S}\right)} \cdot \sqrt{p}$.

Dvs. vi trenger $1{,}177 \cdot \sqrt{p}$ iterasjoner av algoritmen for at vi med sannsynlighet $\geq \frac{1}{2}$ finner to kongruente verdier (mod p).

[51] Prime Numbers and Computer Methods for Factorization – H.Riesel s.177

Antall operasjoner for å finne en primfaktor $p \mid n$ ved å bruke Pollard´s Rho metode kan med andre ord kun beregnes med en viss sannsynlighet $1 - S$. Vi kan likevel estimere det totale arbeid, som er begrenset av antallet iterasjoner og beregningen av største felles divisor som vi må gjøre ved jevne mellomrom. Det blir $O(\sqrt{p} \cdot GCD(n))$, der $GCD(n)$ er tiden det tar å beregne største felles divisor for tall med størrelse opp mot n.

I forhold til RSA er det ikke mulig å "beskytte" seg mot Rho algoritmen på annen måte enn å velge primtallene p og q tilstrekkelige store. Med dagens størrelse på primtallene burde ikke Rho algoritmen være en trussel mot RSA systemet. Det er ikke spesielt nyttig å satse på flaksen heller, siden sannsynligheten er f.eks $< 10^{-6}$ for å finne p ved å bruke $0{,}0014 \cdot \sqrt{p}$ iterasjoner.

5.4 Pollard´s $p - 1$ algoritme

5.4.1 Beskrivelse av algoritmen

Bruker vi forsøksdivisjon som faktoriseringsmetode, blir antall operasjoner omtrent like mange som det er primtall mindre enn $\sqrt{n}$. I Pollard´s rho algoritme kan antallet operasjoner kun forutsies med en viss sannsynlighet. Pollard´s $p - 1$ algoritme er ganske annerledes. Her avhenger ikke antall operasjoner av størrelsen på p, men av størrelsen på faktorene til $p - 1$. Antall operasjoner er som regel proposjonalt med antall primtall mindre enn den største primfaktoren til $p - 1$.

Idéen bak Pollard´s $p - 1$ algoritme er Fermat´s lille teorem som sier at hvis $\gcd(a,p) = 1$ så er $a^{p-1} \equiv 1 \pmod{p}$. Av dette følger at hvis $k = (p - 1)d$ er et multiplum av $p - 1$, så er $a^k \equiv a^{(p-1)d} \equiv (a^{(p-1)})^d \equiv 1^d \equiv 1 \pmod{p}$. For å faktorisere et heltall n som har en ukjent primfaktor p, så er alt vi trenger å gjøre å finne et multiplum k av $p - 1$ og beregne $\gcd(a^k - 1, n)$. Spørsmålet blir nå hvordan vi kan finne en k med $(p - 1)$ som divisor.

Den mest vanlige måten å lage k på, er ved å multiplisere sammen tilstrekkelig mange primfaktorer og håpe på at $p - 1$ er en divisor i k. Dette kan f.eks. gjøres ved å ta alle positive heltall mindre enn en viss grense B, og så plukke ut en primfaktor fra alle tall som er primtallspotenser. Resultatet blir ei liste med primtall: $p_0, p_1, \ldots p_t$. Eksponenten k blir da produktet av alle tallene i lista, dvs. $2 \cdot 3 \cdot 2 \cdot 5 \cdot 7 \cdot 2 \cdot 3 \cdot 11 \cdot 13 \cdot 2 \cdot 17 \cdot 19 \cdot 23 \cdot 5 \cdot 3 \cdot 29 \cdot 31 \cdot 2$ for $B = 33$. For å faktorisere store tall må B selvsagt velges mye større.

Vi kan nå beregne $a^k \pmod{n}$ og $\gcd(a^k \pmod{n} - 1, n)$ og håpe på en ekte faktor. I motsatt fall øker vi B og prøver på nytt.

Siden eksponenten k raskt blir et enormt tall, er det bedre å bruke følgende variant:

$b_0 = a$

$b_1 = b_0^{p_0} \bmod n$, $\gcd(b_1 - 1, n)$

$b_2 = b_1^{p_1} \bmod n$, $\gcd(b_2 - 1, n)$

$b_3 = b_2^{p_2} \bmod n$, $\gcd(b_3 - 1, n)$

osv...

der eksponentene $p_0, p_1, ...$ er primfaktorene fra produktet k. Dette gjør vi til vi finner en ikketriviell divisor av n når vi beregner største felles divisor.

For å spare tid gjør vi det i praksis på følgende måter [52]. Beregn $b_{i+1} \equiv b_i^{p_i}$ (mod n), der p_i er den i-te faktoren til k. Start sekvensen med $b_0 = a$. Her kan vi gjøre det samme som i Pollard's Rho algoritme, nemlig å akkumulere opp $(b_i - 1)$-ene slik at vi ikke trenger å beregne største felles divisor hver gang. Dette kan gjøres ved å beregne:

$$Q_m \equiv \prod_{i=1+K\cdot m}^{K+K\cdot m} (b_i - 1) \pmod n$$

og så beregne gcd(Q_m,n) for $m \geq 0$ for å sjekke om en faktor p har dukket opp. På samme måte som i Rho algoritmen kan vi velge K en plass mellom 10 og 100.

Den fjerde og kanskje enkleste varianten av Pollard's algoritme er å beregne $b_{i+1} \equiv b_i^{p_i}$ (mod n) og så undersøke gcd($b_i - 1$, n) hver gang $K \mid i$.

Av og til oppdages også store faktorer med Pollard's $p - 1$ metode, f.eks.
$p = 121450506296081$ til $10^{95} + 1$, der $p - 1$ er $2^4 \cdot 5 \cdot 13 \cdot 19^2 \cdot 15773 \cdot 20509$, og
$q = 2670091735108484737$ til $3^{136} + 1$, der $q - 1$ er $2^7 \cdot 3^2 \cdot 7^2 \cdot 17^2 \cdot 19 \cdot 569 \cdot 631 \cdot 23993$.
Siden alle primfaktorene til henholdsvis $p - 1$ og $q - 1$ er små, vil faktorene p og q oppdages raskt ved bruk av Pollard's $p - 1$ algoritme.

5.4.2 Et eksempel på Pollard's $p - 1$ algoritme

$n = 11009$. Vi velger startverdi $a = 2$.

$b_0 = 2$

$b_1 = 2^2$	(mod 11009) = 4	gcd(4 – 1,11009) = 1
$b_2 = 4^3$	(mod 11009) = 64	gcd(64 – 1,11009) = 1
$b_3 = 64^2$	(mod 11009) = 4096	gcd(4096 – 1,11009) = 1
$b_4 = 4096^5$	(mod 11009) = 8874	gcd(8874 – 1,11009) = 1
$b_5 = 8874^7$	(mod 11009) = 8983	gcd(8983 – 1,11009) = 1
$b_6 = 8983^2$	(mod 11009) = 9328	gcd(9328 – 1,11009) = 1
$b_7 = 9328^3$	(mod 11009) = 4034	gcd(4034 – 1,11009) = 109 Vi har funnet en faktor!!
	⋮	
$b_{13} = 3925^{23}$	(mod 11009) = 2508	gcd(2508 – 1,11009) = 109
$b_{14} = 2508^5$	(mod 11009) = 1	gcd(1 – 1,11009) = 11009

Vi ser at i 6. iterasjon finner vi en faktor $p \mid n$. Resten av sekvensen er på samme måte som i Rho algoritmen, tatt med for å vise problemene som kan oppstå ved ikke å beregne gcd($b_i - 1$,n) for hver i. La oss si at vi beregner største felles divisor hver 15 iterasjon. I

[52] Prime Numbers and Computer Methods for Factorization – H.Riesel s.173

eksemplet får vi 11009 til svar etter 15. iterasjon. Løsningen på problemet er å gå tilbake og beregne største felles divisor oftere.

5.4.3 Kjøretid

Hvor raskt vil Pollard´s $p-1$ algoritme oppdage en ekte faktor?

Anta at $n = \prod_i p_i^{\alpha_i}$, og at $p_i - 1 = \prod_j q_{ij}^{\beta_{ij}}$.

La q^β være den største primtallspotensen i faktoriseringen til $p_i - 1$. Da vil faktoren p_i bli oppdaget så snart q^β er passert i listen med primfaktorer. Dette betyr at faktoren $p_i \mid n$, der verdien av q^β er den minste for alle faktorer $p \mid n$, oppdages først. Kjøretiden er dermed avhengig av størrelsen på primfaktorene til $p-1$.

Dette er viktig å merke seg når man plukker ut primtall til RSA metoden. I forhold til Pollard´s $p-1$ algoritme bør primtallene være på formen $p = 2p' + 1$, der p' er et primtall, slik at $p-1$ inneholder en stor primfaktor.

5.5 Elliptiske kurve Faktorisering (ECM) [53]

Elliptisk kurve faktorisering ble utviklet av H.W.Lenstra i 1985. Metoden blir regnet som en av de beste for å faktorisere generelle tall. Den er også en av de mest lovende algoritmer for konstruerte tall.

Elliptisk kurve faktorisering er en videreutvikling av prinsippene til Pollard´s $p-1$ algoritme.

Gitt $n = p \cdot q$. Anta $\gcd(a,n) = 1$. Fermats lille teorem sier at $a^{p-1} \equiv 1 \pmod p$.

Hvis $(p-1) \mid k$, så må $a^k \equiv 1 \pmod p$. $p \mid (a^k - 1)$ og $p \mid n \Rightarrow p \mid \gcd(a^k - 1, n)$.

Ofte er $p = \gcd(a^k - 1, n)$.

Hvis $(p-1)$ bare består av små primtallsfaktorer, kan vi lage k ved å multiplisere sammen mange små primtallsfaktorer og håpe på at $(p-1) \mid k$. Bakdelen med denne metoden er at hvis $p-1$ er produktet av bl.a. en stor primfaktor, vil faktoriseringen ta svært lang tid. Pollard´s $p-1$ algoritme er helt og fullt avhengig av egenskapene til gruppa $(\mathbf{Z}/p\mathbf{Z})^*$.

Det er her ECM har sin store styrke. I ECM har vi et utall av grupper med forskjellige egenskaper å velge blandt. Fungerer ikke ei gruppe, så velger vi ei ny. Poenget er å bruke elliptiske kurver, siden punktene på disse kurvene danner grupper under en spesiell form for punktaddisjon. I motsetning til Pollard´s $p-1$ metode, der antall elementer i gruppa er $p-1$, ligger antallet punkter på den elliptiske kurven (mod p) en eller annen plass i et intervall rundt p. Det betyr at vi har en brukbar sjanse til å finne en elliptisk kurve der antall punkter på kurven er et tall som ikke er produktet av for store primfaktorer. Finner vi en slik kurve er det som regel lett å faktorisere n.

Siden ECM bygger på nye idéer må vi innføre litt ny teori. Endel av teorien krever matematisk innsikt på høyt nivå, og jeg har derfor ikke som mål å kunne bevise alle påstandene jeg kommer med.

---

[53] Hvis ikke annet er nevnt er alt frem til Del 5.5.5 om faktorisering tatt fra boka Elliptic Curves av Dale Husemöller og Loren Olson´s, forelesninger vår 89

5.5.1 Innledende definisjoner

La **K** være en kropp og $n > 0$ et heltall.

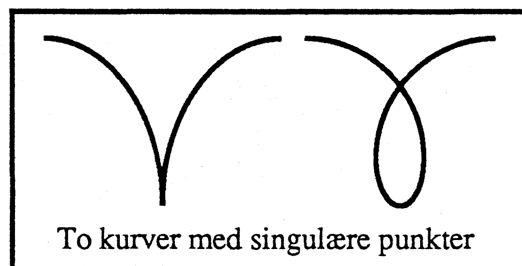
Et **Projektivt n-dimensjonalt rom** $\mathcal{P}^n = \mathcal{P}^n(\mathbf{K})$ består av alle $n + 1$ tupler $(x_0, x_1, \ldots, x_n)$ med $x_i \in \mathbf{K}$ og der ikke alle x_i skal være 0. To punkter x,y i $\mathcal{P}^n$ sies å være ekvivalente, $(x_0, x_1, \ldots, x_n) \sim (y_0, y_1, \ldots, y_n)$, hvis det eksisterer en $\lambda \in \mathbf{K}^*$ slik at $x_i = \lambda \cdot y_i$ for alle i. Dette betyr at punktene i et projektivt rom kan sees på som alle rette linjer gjennom origo.

En **projektiv** kurve i $\mathcal{P}^2$ er $V(f) = \{(x,y,z) \in \mathcal{P}^2 \mid f(x,y,z) = 0\}$ der $f \in \mathbf{K}[x,y,z]$ er ikke-konstant og homogent.

La $f(x,y,z) \in \mathbf{K}[x,y,z]$ være ikkekonstant og homogent. La $P = (x,y,z)$ være et punkt på kurven, dvs $f(P) = 0$.

P er et **Singulært punkt** dersom $\frac{\partial f}{\partial x}(P) = \frac{\partial f}{\partial y}(P) = \frac{\partial y}{\partial z}(P) = 0$.

Det betyr at vi har problemer med tangentlinja i punktet P. Det kan tenke seg at $V(P) = \emptyset$. I vårt vanlige affine rom over de reelle tallene $\mathcal{A}^2(\mathbf{R})$, så får vi at $V(x^2 + y^2 = -1) = \emptyset$. Ser vi på dette over de komplekse tall **C**, så får vi punkter.


To kurver med singulære punkter

En **Elliptisk kurve** E over **K** er $V(f)$ i $\mathcal{P}^2$ der f er homogen av grad 3, samt et punkt $e \in \mathcal{P}(\mathbf{K})$ i $V(f)$, slik at f ikke har noen singulære punkter over **K**.

Til elliptisk kurve faktorisering brukes kurver på formen:

$y^2 = x^3 + a_4 x + a_6$ der kar(**K**) $\neq 2,3$.

Slike ligninger kalles for **Weierstrass ligninger**.

Vi trenger noen konstanter:

$\Delta = -16(4a_4^3 - 27a_6^2)$

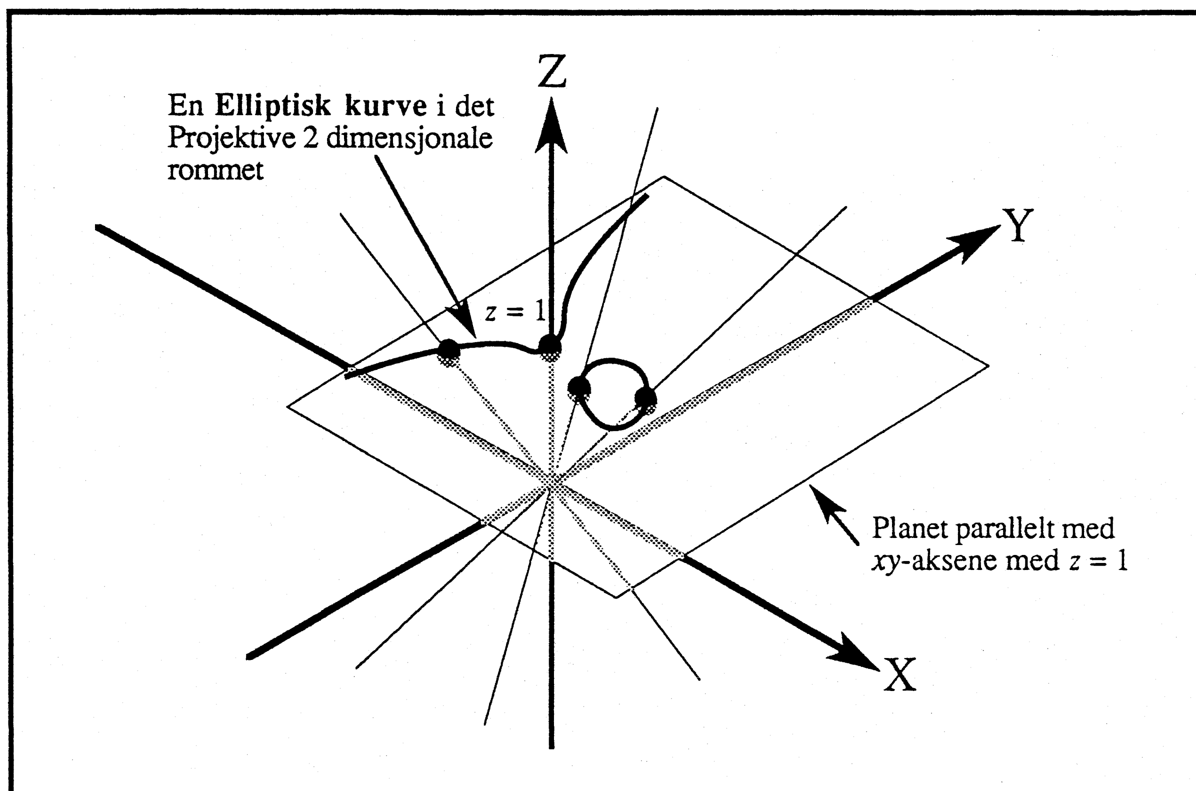
$j = \frac{(-48a_4)^3}{\Delta}$

Δ kalles **diskriminanten** og j kalles **j-invarianten**

Teorem La E være en kurve gitt ved en Weierstrass ligning. Da er E ikkesingulær $\Leftrightarrow \Delta \neq 0$.

De færreste av oss liker å tenke på rom der punktene er ekvivalensklasser. Men et projektivt rom er akkurat som det vanlige affine rommet, bortsett fra at punktene i dette rommet kan sees

på som alle mulige rette linjer som går igjennom origo. Vi går hele tiden ut fra det projektive 2 dimensjonale rommet som de Elliptiske kurvene holder til i. Dette rommet kalles forøvrig det projektive planet, selv om det til forveksling for oss ser ut som et vanlig 3 dimensjonalt rom. Det kan visualiseres som et 3 dimensjonalt rom med de nevnte linjer gjennom origo der det er satt inn et plan som er parallelt med xy-planet. Det er vanlig å velge planet $z = 1$. Vi kan nå identifisere nesten alle ekvivalensklassene entydig i det punktet de skjærer planet, nemlig i $(x,y,1)$. (Se figuren under) Resten av ekvivalensklassene som er på formen $(x,y,0)$ kalles for linja i det "uendelig fjerne". Når vi senere vil finne ut at punktene på elliptiske kurver har gruppestruktur, er denne linja nært knyttet til identitetselementet i gruppa.



5.5.2 Elliptiske kurver har gruppestruktur

Bezouts Teorem [54] La F og G være 2 projektive kurver i $\mathcal{P}^2$ av grad m og n. Da består $F \cap G$ av $m \cdot n$ punkter hvis vi teller multiplisiteten riktig.

Korollar La E være en elliptisk kurve og L ei linje i $\mathcal{P}^2$. Da snitter de hverandre i 3 punkter med multiplisitet (En tangent har multiplisitet 2 og et vendepunkt har multiplisitet 3).

E har gruppestruktur og vi kan gi en beskrivelse av gruppeloven på E

La E være gitt ved $y^2 = x^3 + a_4x + a_6$. Da gjelder:

$e = (0,1,0)$ er det additive identitetselementet.

Hvis $P_0 = (x_0,y_0,1)$ så er $-P_0 = (x_0,-y_0,1)$

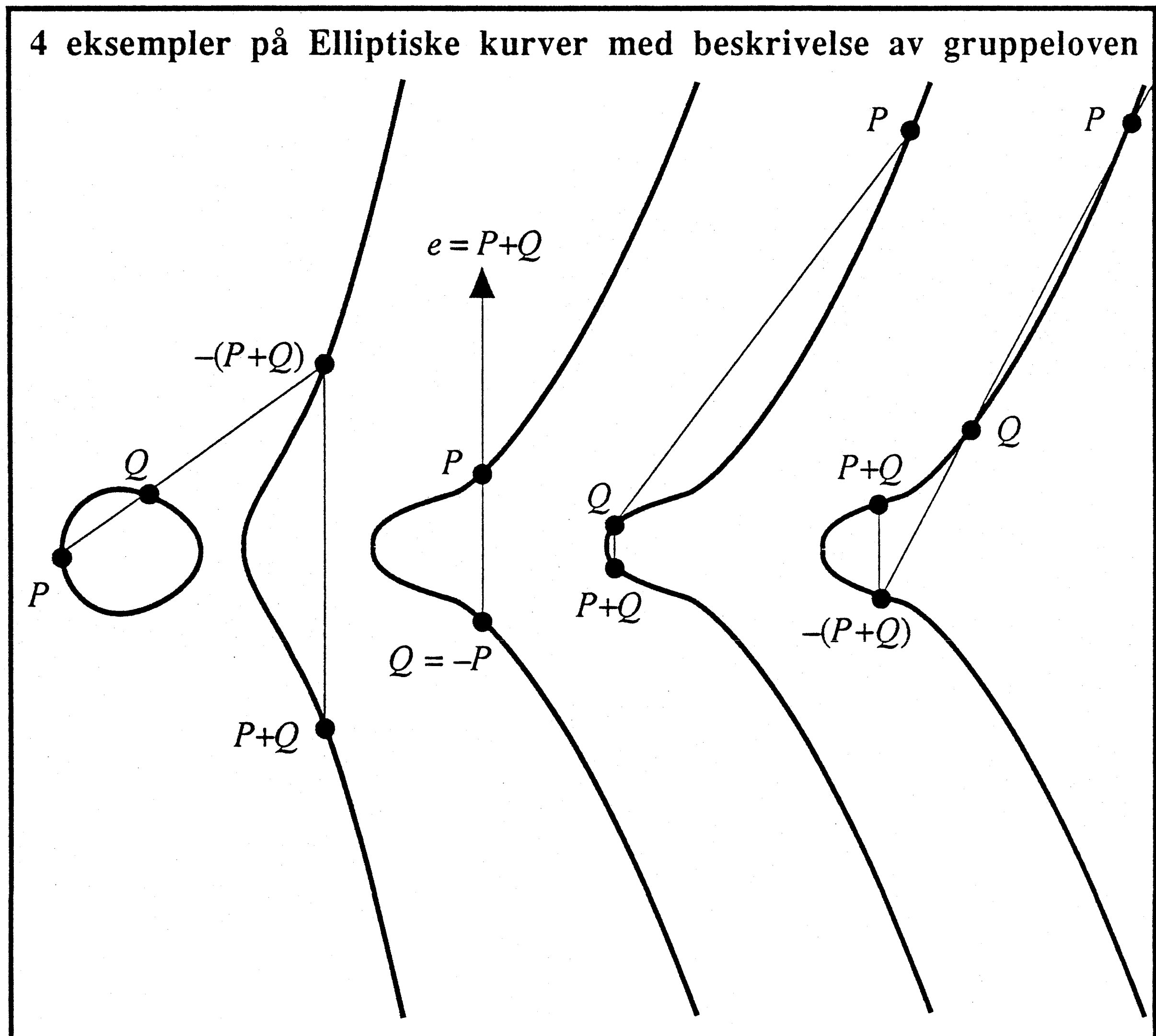
La $P_1 + P_2 = P_3$ med $P_i = (x_i,y_i,1)$

[54] Introduction to Elliptic Curves and Modular Forms – Neal koblitz s.32

Hvis $x_1 = x_2$ og $y_1 = -y_2$ så er $P_1 + P_2 = e$.

Hvis $x_1 = x_2$ og $y_1 = y_2$, dvs. vi vil finne $2P$,
da er $x_3 = \left(\frac{3x_1^2 + a_4}{2y_1}\right)^2 - 2x_1$ og $y_3 = -y_1 + \left(\frac{3x_1^2 + a_4}{2y_1}\right) \cdot (x_1 - x_3)$.

Hvis $x_1 \neq x_2$ la $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$ og $\nu = \frac{y_1 x_2 - y_2 x_1}{x_2 - x_1}$.
Da er $x_3 = \lambda^2 - x_1 - x_2$ og $y_3 = -\lambda x_3 - \nu$.



Figur: Eksempeler på forskjellige elliptiske kurver med gruppeloven skissert. For addisjon av 2 punkter P og Q, trekk ei linje gjennom punktene og merk av det punktet der linja skjærer kurven for 3. gang (Husk: ei linje og en elliptisk kurve snitter hverandre i 3 punkter med multiplisitet). Gå så vertikalt opp eller ned til vi på nytt snitter kurven og vi har funnet punktet $P + Q$. Spesialtilfeller får vi bl.a når P og Q har samme x verdi, hvis linja som snitter P er en tangent i det punktet, og ved beregning av $2P$ hvis P er et vendepunkt.

5.5.3 Homomorfier og Isomorfier av Elliptiske kurver

Punktene på to elliptiske kurver med forskjellige Weierstrass ligninger kan danne isomorfe grupper selv om Weierstrass ligningen er entydig for en elliptisk kurve. Det kan derfor være interessant å se på hva som må til for at punktene på to elliptiske kurver skal danne isomorfe grupper.

Vi kan lage følgende transformasjon fra en elliptisk kurve til en annen:

$f : x = u^2 \cdot \hat{x}$ og $y = u^3 \cdot \hat{y}$, $u \in \mathbf{K}$

Vi får da at $u^{12} \cdot \hat{\Delta} = \Delta$ og at $\hat{j} = j$.

f er da en homomorfi av elliptisk kurver (pr. definisjon).

Isomorfiklassene av elliptiske kurver over **K** er parametrisert av j-invarianten.

1. $E \cong \hat{E} \Leftrightarrow j = \hat{j}$ forutsatt at transformasjonen finnes.
2. Gitt $j \in \mathbf{K}$, så eksisterer det en elliptisk kurve med j-invariant j.

Vi får undersøke hva som må til for at transformasjonen som er definert over eksisterer.

Beskrivelse av alle isomorfiklasser av elliptisk kurver over K.

La E og $\hat{E}$ være to elliptiske kurver definert ved:

$E : y^2 = x^3 + a_4 x + a_6$ og $\hat{E} : y^2 = x^3 + \hat{a_4} x + \hat{a_6}$

Da er

$a_4 = u^4 \cdot \hat{a_4}$ og $a_6 = u^6 \cdot \hat{a_6}$. Anta $j = \hat{j}$.

Anta at $j \neq 0{,}1728$ (dvs. $a_4 \neq 0$, $a_6 \neq 0$).

Vi har at $u^2 = \dfrac{a_6 \cdot \hat{a_4}}{\hat{a_6} \cdot a_4}$. $E \cong \hat{E} \Leftrightarrow \dfrac{a_6 \cdot \hat{a_4}}{\hat{a_6} \cdot a_4} \in (\mathbf{K}^*)^2$.

Dette betyr at E er isomorf med $\hat{E}$ hvis og bare hvis $j = \hat{j}$ og $\dfrac{a_6 \cdot \hat{a_4}}{\hat{a_6} \cdot a_4}$ er kvadratet av et eller annet element i **K**. Det betyr også at enhver elliptisk kurve E er isomorf med en annen kurve $\hat{E}$ der a_4 og a_6 er kvadratfrie heltall.

Anta $j = 1728$ (dvs. $a_6 = 0$): $u^4 \cdot \hat{a_4} = a_4$. $E \cong \hat{E} \Leftrightarrow \dfrac{a_4}{\hat{a_4}} \in (\mathbf{K}^*)^4$

Da er E isomorf med en kurve $y^2 = x^3 + a_4 x$ der a_4 er et 4. potensfritt heltall.

Anta $j = 0$ (dvs. $a_4 = 0$).

Da er $u^6 \cdot \hat{a_6} = a_6$. $E \cong \hat{E} \Leftrightarrow \dfrac{a_6}{\hat{a_6}} \in (\mathbf{K}^*)^6$

Da er E isomorf med en kurve $y^2 = x^3 + a_6$ der a_6 er et heltall som er 6. potensfritt.

Vi skal se etterhvert at for å faktorisere heltall er det viktig at kurvene som brukes ikke er isomorfe siden vi kun er interessert i grupper med forskjellig struktur.

5.5.4 Antall punkter på en Elliptisk kurve over $\mathbf{F}_p$

La oss se på elliptiske kurver over endelige kropper. La p være et primtall. La $\mathbf{F}_p = \mathbf{Z}/p\mathbf{Z}$

Da har:

$\mathcal{A}^1(\mathbf{F}_p) = \{(a_1) \mid a_1 \in \mathbf{F}_p\}$ p elementer,

$\mathcal{A}^2(\mathbf{F}_p) = \{(a_1,a_2) \mid a_1,a_2 \in \mathbf{F}_p\}$ p^2 elementer,

$\mathcal{P}^1(\mathbf{F}_p) = \{(a_0,a_1) \mid (a_0,a_1) \neq (0,0)\}$ $(p^2-1)/(p-1) = p+1$ elementer og

$\mathcal{P}^2(\mathbf{F}_p) = \{(a_0,a_1,a_2) \mid (a_0,a_1,a_2) \neq (0,0,0)\}$ $(p^3-1)/(p-1) = p^2+p+1$ elementer.

La E være en elliptisk kurve over $\mathbf{F}_p$ gitt ved en Weierstrass ligning. Da er E en kurve i $\mathcal{P}^2(\mathbf{F}_p)$. La $E(\mathbf{F}_p)$ betegne den abelske gruppen av punkter på den elliptiske kurven over $\mathbf{F}_p$. Da har $E(\mathbf{F}_p)$ har helt sikkert bare et endelig antall elementer.

La E være en elliptisk kurve over $\mathbf{F}_p$. La $a_p = 1 + p - \#E(\mathbf{F}_p)$ være **Sporet til Frobenius.** Sporet til Frobenius er med andre ord differansen mellom p og antall punkter på den elliptiske kurven over $\mathbf{F}_p$.

Det betyr at $\#E(\mathbf{F}_p) = 1 + p - a_p$.

Teorem (Hasse): La p være et primtall og E en elliptisk kurve over $\mathbf{F}_p$. Da er $|a_p| < 2\sqrt{p}$

Teorem Anta at E er en elliptisk kurve over $\mathbf{F}_p$.

Da er $a_p = -\sum_{t \bmod p} \left(\frac{t^3 + a_4 t + a_6}{p}\right)$ (Legendre symbolet)

Hvis $y^2 = x^3 + a_4 x + a_6$, $a_4, a_6 > 0$, $p \geq 5$ og $M = \frac{p-1}{2}$.

Da er $a_p \equiv \sum_{\substack{2h+3i=M \\ h,i\geq 0}} \frac{M!}{i!\cdot h!\cdot(M-h-i)!} \cdot a_4^h \cdot a_6^i \pmod p$

$y^2 = x^3 + a_4 x$, $a_4 \neq 0$, $p \geq 5$

Hvis $p \not\equiv 1 \pmod 4$ så er $a_p = 0$ og $\#E(\mathbf{Z}/p\mathbf{Z}) = 1 + p$

Hvis $p \equiv 1 \pmod 4$ og $p = 4n + 1$ så er $a_p \equiv \binom{2n}{n} a_4^n \pmod p$

$y^2 = x^3 + a_6$, $a_6 \neq 0$, $p \geq 5$

Hvis $p \not\equiv 1 \pmod 6$ så er $a_p = 0$ og $\#E(\mathbf{Z}/p\mathbf{Z}) = 1 + p$

Hvis $p \equiv 1 \pmod 6$ og $p = 6n + 1$ så er $a_p \equiv \binom{3n}{n} a_6^n \pmod p$

Gitt $a_6 = 0$. Hvor mange forskjellige verdier $(\bmod p)$ kan a_4^n ta på seg?

$a \rightarrow a^n$ er en homomorfi $\mathbf{F}_p^* \rightarrow \mathbf{F}_p^*$ fra ei syklisk gruppe av orden $(p-1)$ inn i seg selv. Kjernen har n elementer og bildet har 4 elementer, dvs. a_4^n kan ta på seg 4 verdier $(\bmod p)$ når a_4 varierer gjennom $\mathbf{F}_p^*$.

Setter vi $a_6 = 0$ kan vi få grupper med maksimalt 5 forskjellige ordener. Setter vi $a_4 = 0$ får vi grupper med maksimalt 7 forskjellige ordener.
Dette er viktig å legge merke til når man skal velge elliptiske kurver til faktorisering. Dersom vi trenger et rikt utvalg av grupper der $\#E(\mathbf{Z}/p\mathbf{Z})$ varierer mest mulig bør både a_4 og a_6 være forskjellig fra null.
Siden $|a_p| < 2\sqrt{p}$ burde vi ha gode muligheter til å velge en elliptisk kurve der $\#E(\mathbf{Z}/p\mathbf{Z})$ er produktet av bare "små" primfaktorer. Når det gjelder valg av kurver ellers, så er det beste å velge dem tilfeldig siden a_p er så vanskelig å regne ut for store tall. Pr. i dag finnes det ikke noen måte å finne gode kurver på som er bedre enn å velge tilfeldig. Men det bør sjekkes at ikke noen av kurvene som brukes danner isomorfe grupper, hvis det er mulig.

5.5.5 Elliptisk kurve Faktorisering

Gitt $n > 3$ et sammensatt tall. Målet er å finne $p \mid n$. Vi velger en elliptisk kurve E over $\mathbf{Z}/n\mathbf{Z}$ og et punkt $P \in E(\mathbf{Z}/n\mathbf{Z})$. Merk at $\mathbf{Z}/n\mathbf{Z}$ er en ring siden n er sammensatt. For å være sikker på at kurven ikke har singulære punkter må vi sjekke at $\Delta \neq 0$. Så beregner vi $P_k = k \cdot P$ på samme måte som i Pollard's $p - 1$ algoritme, bortsett fra at multiplikasjonen i $p - 1$ metoden nå er erstattet med den spesielle punktaddisjonen. k er den samme som i $p - 1$ algoritmen mens P har rollen som a. Vi håper nå på at $P_k = e = (0,1,0)$ når vi regner over $\mathbf{Z}/p\mathbf{Z}$ for et primtall $p \mid n$.

Når vi beregner P_k er det noen tall som dukker opp i nevnerne i formlene for $P_1 + P_2$. Disse må være invertible (mod n) siden vi utfører divisjonen ved å gange telleren med den inverse av nevneren. Vi regner som om $\mathbf{Z}/n\mathbf{Z}$ er en kropp. Poenget er at hvis vi får problemer med nulldivisorer har vi funnet $p \mid n$.

Vi trenger følgende proposisjon:
La E være en elliptisk kurve over $\mathbf{Z}/n\mathbf{Z}$ med $y^2 = x^3 + a_4x + a_6$ der $a_4, a_6 \in \mathbf{Z}/n\mathbf{Z})$ og $(\Delta, n) = (4a_4^3 + 27a_6^2, n) = 1$. La e_p være identitetselementet i $E(\mathbf{F}_p)$.
La $P_1 = (x_1,y_1)$ og $P_2 = (x_2,y_2)$ være punkter i $E(\mathbf{Z}/n\mathbf{Z})$. Anta $P_1 \neq -P_2$.

1. Hvis $P_1 = P_2$, så er $\gcd(y_1,n) > 1 \Leftrightarrow P_1 + P_2 = e_p$ i $E(\mathbf{F}_p)$ for et primtall $p \mid n$.
2. Hvis $P_1 \neq P_2$, så er $\gcd(x_2 - x_1,n) > 1 \Leftrightarrow P_1 + P_2 = e_p$ i $E(\mathbf{F}_p)$ for et primtall $p \mid n$.

Hvis nevnerne ikke er primisk med n har vi funnet et eller annet multiplum k_1P (en av delsummene til kP) som for $p \mid n$ har egenskapen k_1P (mod p) = e mod p. Det betyr at punktet P på $E(\mathbf{Z}/p\mathbf{Z})$ har en orden som deler k_1. Vi har nå funnet en faktor $p \mid n$.

Med elliptisk kurve metoden kan vi forsøke oss på flere kurver. Teoremet til Hasse sier oss at $|a_p| \leq 2\sqrt{p}$. Det vil si at $\#E(\mathbf{Z}/p\mathbf{Z})$ ligger rundt $p + 1$. Hvis mange heltall som er "nær" $p + 1$ bare har små primtallsfaktorer, har vi gode sjanser til å finne p.
Når vi skal bruke ECM til å faktorisere store tall kan dette gjøres på flere måter. Vi kan f.eks

bruke en kurve helt til vi finner en faktor p. Det kan ta lang tid hvis vi har vært uheldig med valg av kurve. Det beste er nok å stoppe når vi er kommet til en grense som vi på forhånd har bestemt, for så å prøve en ny kurve. Dette gjør vi helt til vi finner en faktor. I praksis velger man gjerne mange kurver som kjøres i parallell helt til en av kurvene finner en faktor.

Sammenligning av ECM og Pollard´s p - 1:

Metode	*Når får vi klaff*	*Gruppe*	*#Elementer*	*#Grupper*
Pollard´s $p-1$	$\#(\mathbf{Z}/p\mathbf{Z})$ har små primfaktorer	$(\mathbf{Z}/p\mathbf{Z})^*$	$p-1$	1
ECM	$\#E(\mathbf{Z}/p\mathbf{Z})$ har små primfaktorer	$E(\mathbf{Z}/p\mathbf{Z})$	$p+1-a_p$	Mange

5.5.6 Kjøretid [55]

For å bestemme kjøretiden for et primtall p med grense B (som er valgt optimalt), må man bestemme sannsynligheten for at en tilfeldig elliptisk kurve modulo p har en orden N som ikke er delelig med et primtall $> B$. Ordenen N til alle elliptiske kurver modulo p er som kjent fordelt rundt $p + 1$, i intervallet $p + 1 - 2\sqrt{p} < N < p + 1 + 2\sqrt{p}$. Tettheten er størst rundt $p + 1$.

Det betyr at sannsynligheten er omtrent lik sjansen for at et tilfeldig valgt heltall mellom 0 og p ikke er delelig med primtall større enn B. Denne sannsynligheten er ifølge Koblitz omtrent u^{-u}, der $u = \log p/\log B$. Han estimerer kjøretiden til $O(e^{C\sqrt{r \log r}})$ der r er antall bit i n. Mer presist, anta at n er et positivt heltall som ikke er en potens av et primtall og ikke delelig med 2 eller 3. Anta videre at det eksisterer en regel for hvordan heltallene som ikke er delelige med primtall større enn B fordeler seg i et kort intervall rundt p. Da har Lenstra bevist følgende sannsynlighets tidsestimat for antall operasjoner som kreves for å finne en ikketriviell divisor av n:

$$e^{\sqrt{(2+\varepsilon) \log p \log \log p}}$$

der p er den minste primfaktoren til n og ε går mot 0 for store p. Siden $p < \sqrt{n}$ følger det at

$$e^{\sqrt{(1+\varepsilon) \log p \log \log p}}.$$

I forhold til RSA er det på samme måte som i Pollard´s $p - 1$ metode viktig å velge primtallene slik at $p = 2p' + 1$ der p' er et primtall. p' kan i tillegg gjerne velges som $2p'' + 1$ for å gjøre faktoriseringen ekstra vanskelig.

2 eksempler på Elliptisk kurve faktorisering med $n = 323$

Vi velger kurven $y^2 = x^3 + x + 25$ og et punkt $P_0 = (0,5)$ og får følgende sekvens:

$P_1 = 2P_0 = (42,120)$

$P_2 = 3P_1 = P_1 + 2P_1 = (42,120) + (231,44) = (202,298)$

$P_3 = 2P_2 = (88,215)$

$P_4 = 5P_3 = 2P_3 + 2P_3 + P_3 = (126,89) + (126,89) + (88,215) = (69,158) + (88,215) = ??$

Her får vi 57/19 som stigningstall til linja mellom $4P_3$ og P_3. Det er ikke mulig å finne noen invers til 19 (mod 323) siden gcd(19,323) = 19, og vi har dermed faktorisert 323.

Vi prøver en ny kurve $y^2 = x^3 + x + 1$ og et punkt $P_0 = (0,1)$ og får følgende sekvens:

[55] A Course in Number Theory and Cryptography – Neal Koblitz s.176

$P_1 = 2P_0 = (81,120)$
$P_2 = 3P_1 = P_1 + 2P_1 = (81,120) + (9,165) = (299,294)$
$P_3 = 2P_2 = (10,29)$
$P_4 = 5P_3 = 2P_3 + 2P_3 + P_3 = (27,22) + (27,22) + (10,29) = (27,301) + (10,29) = ??$
Her får vi 272/17 som stigningstall til linja mellom $4P_3$ og P_3. Det er ikke mulig å finne noen invers til 17 (mod 323) siden gcd(17,323) = 17, og vi har dermed faktorisert 323 på nytt. Legg merke til at denne gangen fant vi faktoren 17 først.

5.6 Generelt om kvadratiske rest algoritmer

Jeg skal nå se litt på 2 faktoriseringsalgoritmer som ikke nødvendigvis oppdager små faktorer raskere enn store. Begge algoritmene baserer seg på bruk av kvadratiske rester modulo n. Den raskeste av dem, Kvadratisk sil algoritmen er for øyeblikket den største trusselen mot RSA systemet siden den har faktorisert tall med mer enn 100 desimalsiffer. Delbrøkoppspaltningsalgoritmen er en forløper til kvadratisk sil og brukes derfor sjelden. Algoritmene ble egentlig beskrevet første gang i 1926 av Maurice Kraitchik, men han hadde ingen muligheter til å realisere dem, siden de er lite egnet uten bruk av datamaskin. Jeg kommer ikke til å gå i dybden i teorien til algoritmene.

Et heltall q er en **kvadratrest** (mod n) hvis det eksisterer et heltall x slik at $x^2 \equiv q$ (mod n). x er da kvadratrota til q (mod n).
Faktoriseringsalgoritmene baserer seg på at hvis vi kan finne to forskjellige kvadratrøtter x og y til q (mod n), så kan vi faktorisere n. Dette kan vi siden $x^2 \equiv y^2$ (mod n). n er da en divisor i $(x^2 - y^2) = (x - y)\cdot(x + y)$. Hvis $n = p\cdot q$ har vi fire muligheter:

$p\cdot q \mid (x - y)$
$p\cdot q \mid (x + y)$
$p \mid (x - y)$ og $q \mid (x + y)$
$q \mid (x - y)$ og $p \mid (x + y)$

De to første tilfellene gir oss ingen informasjon av interesse. De to siste tilfellene derimot gir oss p eller q ved å beregne gcd($x - y$, n), siden p eller q deler $x - y$ og n. Vi kan dermed faktorisere n med sannsynlighet 1/2. Gjør vi dette 10 ganger vil vi med sannsynlighet 0,999 finne to forskjellige kvadratrøtter som gir en ikketriviell faktor.
Måten dette gjøres på i praksis er å generere et stort antall kvadratiske rester q_k og for hver k et heltall a_k slik at $a_k^2 \equiv q_k$ (mod n).

Velg så en mengde primtall $B = \{p_1, p_2,..., p_T\}$ som vi kaller primbasen. B kan velges som de T minste primtallene som er er en mulig divisor til en kvadratrest modulo n. Dvs. vi finner de T minste primtallene slik at n er en mulig kvadratrest modulo hver av primtallene p_i. Omtrent halvparten av primtallene har denne egenskapen. Faktoriser så alle, dvs. minst $T + 1$ av de kvadratiske restene q_i fullstendig over primbasen B.
Så konstruerer vi en 0/1-matrise ved å kalle q-ene som faktoriseres fullstendig over primbasen

for $q_1, q_2, \ldots, q_T, q_{T+1}$, og plasser tallet 1 i den i-te raden og j-te kolonnen hver gang det j–te primtallet i primbasen deler q_i med en odde potens. Deretter utfører vi gaussisk eliminasjon på matrisa. Målet er å finne en samling q-er der summen i hver kolonne er et partall. Denne samlingen q-er kan kalles $q_1, q_2, \ldots, q_m$, og vi lar også de korresponderende a-er få samme indeks.
Vi har da at $x^2 = q_1 \cdot q_2 \cdot \ldots \cdot q_m \equiv a_1^2 \cdot a_2^2 \cdot \ldots \cdot a_m^2 = y^2 \pmod{n}$

5.7 Faktorisering ved bruk av delbrøkoppspaltning

5.7.1 Algoritme for delbrøkoppspaltning

Helt siden Gauss har man visst at det er lett å generere små kvadratiske rester (mod n) ved å beregne delbrøkoppspaltningen til $\sqrt{n}$. Dette kan gjøres ved å bruke følgende algoritme [56]:

1. Sett $\Delta = \lfloor \sqrt{n} \rfloor$, $a_{-1} = 1$, $P_0 = 0$, $q_0 = 1$ og $\alpha_0 = a_0 = \Delta$.
2. For $k > 0$, beregn

$$\alpha_k = \lfloor (P_k + \Delta)/q_k \rfloor,$$
$$P_{k+1} = \alpha_k \cdot q_k - P_k,$$
$$q_{k+1} = (n - P_{k+1}^2)/q_k,$$
$$a_{k+1} = q_{k+1} \cdot a_k + a_{k-1} \pmod{n}$$

Med denne får vi at $a_k^2 \equiv (-1)^{k+1} q_{k+1} \pmod{n}$. I tillegg får vi at $q_k < 2\sqrt{n}$ for alle k. Det vil si at hvis vi skal faktorisere et 60-sifret tall n, så er q-ene som skal faktoriseres over primbasen en mengde 30-sifrete tall. Dvs. vi kan faktorisere et 60-sifret tall ved å faktorisere tilstrekkelig mange 30-sifrete tall over B. Det burde være mulig å faktorisere nok av disse tallene siden de er bare halvparten så lange som n. I tillegg er disse tallene generelle, ikke laget for å være vanskelige å faktorisere.

5.7.2 Selve faktoriseringsalgoritmen

1. Bruk algoritmen som er skissert over for å generere et stort antall av par (q,a) med $\pm q \equiv a^2 \pmod{n}$

2. Faktoriser q-ene over primbasen B med T primtall helt til noen flere enn T av q-ene er fullstendig faktorisert over primtallene.

3. Sett opp en matrise med nullere og enere der hver rad korresponderer med et faktorisert tall q og enerne indikerer primtallene i B som dividerer q med odde potens.

4. Bruk Gaussisk eliminasjon for å finne ei samling rader (q-er) der tallet på enere i hver kollonne er et partall. For hver slik rad utfør steg 5 og 6.

---

[56] Computational Methods for Factoring Large Integers – Marvin Wunderlich

5. Multipliser sammen alle q-ene tilsvarende samlingen med rader fra steg 4 slik at vi får et stort kvadrattall x som vi beregner kvadratrota av. Multipliser så sammen alle tilsvarende verdier for a og vi får tallet y. Både x og y kan reduseres modulo n.

6. Beregn $p = \gcd(n, x - y)$. Hvis $1 < p < n$ har vi funnet en ekte faktor og vi er ferdig. Ellers prøver vi en samling nye rader fra steg 4.

7. Hvis ingen av radene fører frem, gå tilbake til steg 1 og generer noen flere par av tall (q,a).

5.7.3 Et eksempel

La oss faktorisere $n = 1147$ med delbrøkoppspaltningsalgoritmen. Vi får følgende sekvens:

$a_{-1} = 1$ $q_0 = 1$
$a_0 = 33$ $q_1 = 58$
$a_1 = 34$ $q_2 = 9$
$a_2 = 237$ $q_3 = 34$
$a_3 = 271$ $q_4 = 33$
$a_4 = 508$ $q_5 = 11$
$a_5 = 517$ $q_6 = 38$
$a_6 = 1025$ $q_7 = 27$
$a_7 = 395$ $q_8 = 33$

La $B = \{2,3,5,7,11\}$. Vi plukker så ut q_i-ene 27,33 og 11 som vi faktoriserer over B.
Matrisen blir:

q_i	2	3	5	7	11
27	0	1	0	0	0
33	0	1	0	0	1
11	0	0	0	0	1

Summen av de 3 radene gir oss bare nullere (mod 2) og vi har dermed funnet en $x^2 = 27 \cdot 33 \cdot 11$ slik at $x = 99 \pmod n$. y blir $1025 \cdot 271 \cdot 508 \pmod n = 25$.
Vi kan nå beregne $\gcd(99 - 25, 1147) = 37$ og $n = 31 \cdot 37$.
Primtallene 5 og 7 i primbasen kan egentlig sløyfes, siden de umulig kan være en divisor til en kvadratrest (mod 1147).
Dette eksemplet er selvfølgelig litt sært, men det er vel ikke uten grunn at Kraitchik la algoritmen på hylla i 1926. En datamaskin er midt i blinken for denne metoden!! Brillhart/Morrison var de første til å faktorisere store tall ved å bruke denne metoden, og den revolusjonerte faktorisering. Før den kom klarte man å faktorisere tall opp til 25-30 siffer, og etter (dvs 1970) faktoriserte de et tall på rundt 40 siffer. Det største tallet som er faktorisert med denne metoden er på 64 siffer.

Kjøretiden er begrenset av $O(e^{C \cdot \sqrt{\log n \log \log n}})$ uten at jeg skal si noe mer om den. Interesserte kan lese [57].

---

[57] A Course in Number Theory and Cryptography – Neal Koblitz s.141

5.8 Kvadratisk sil metoden [58]

Carl Pomerance gjenoppdaget Kvadratisk Sil metoden (KSM) i 1983 etter at den hadde vært "glemt" i nesten 60 år.

Denne algoritmen har klare fordeler fremfor faktorisering ved bruk av delbrøkoppspaltning (DB). Hvis primbasen i DB består av T primtall må vi produsere $T+1$ faktoriserte q-er for å faktorisere n. Hver eneste q må deles med hver av de T primtallene i faktorbasen. Mesteparten av beregningstiden i DB går med til å dele q-ene på primtall p_i, $0 < i \le T$, i B og få en rest ulik null, som sier oss at p_i ikke er en divisor i q. Tiden går mao. stort sett med til å beregne negativ informasjon. Det ville derfor være et stort fremskritt hvis man kunne konstruere en "funksjon" som på forhånd fortalte oss hvilke p_i-er som er slik at $p_i \mid q$. Det er her KSM har sin styrke. Prisen for dette er at de kvadratiske restene q blir litt større.

I KSM defineres de kvadratiske restene ved $q(x) = (\lfloor\sqrt{n}\rfloor + x)^2 - n$, der x varierer over heltallene mellom $-\frac{1}{2}W$ og $+\frac{1}{2}W$, der W er valgt på en eller annen optimal måte.

Det følger da at $a(x)^2 = (\lfloor\sqrt{n}\rfloor + x)^2 \equiv q(x) \pmod n$ og at $q(x) < 2W\sqrt{n} + \frac{1}{4}W^2$.

Husk at i DB så er $q < 2\sqrt{n}$. I KSM er størrelsen på q-ene litt større pga. W. Det fine er imidlertid at hvis p_i er et av primtallene i primbasen, og p_i er en faktor i $q(x) = (\lfloor\sqrt{n}\rfloor + x)^2 - n$, så er det også en faktor i $q(x + p_i) = (\lfloor\sqrt{n}\rfloor + x + p_i)^2 - n = q(x) + 2p_i(\lfloor\sqrt{n}\rfloor + x) + p_i^2$.

Det viser seg at for hver odde p_i i B kan vi finne to heltall x_0 og x_1 slik at når $p_i \mid q(x)$ så vil x enten være kongruent med x_0 eller x_1 mod p_i. Det får vi siden det er to løsninger på kongruensen $q(x) \equiv 0 \pmod{p_i}$. Dette gir oss den etterlengtete orakelfunksjonen.

I stedet for forsøksdivisjon av alle q-ene med alle p_i-ene i primbasen beregner vi $q(x)$ for hver x i sekvensen $-\frac{1}{2}W, \ldots, \frac{1}{2}W$. Så prøver vi forsøksdivisjon av suksessive q-er med hver p_i helt til vi finner de to første x-ene, kalt x_{0p_i} og x_{1p_i}, der p_i deler $q(x)$. Hver av disse divisjonene må lykkes og ingen andre forsøksdivisjoner trengs. Utfør så divisjonen av q med p_i bare for $x_{0p_i} + k \cdot p_i$ og $x_{1p_i} + k \cdot p_i$ helt til k er slik at $k \cdot p_i > W$.

Nå som vi vet at vi vil få divisjoner, kan vi bruke logaritmen til q-ene istedet for q-ene, og så subtrahere logaritmen til p_i-ene fra $\log(q)$ i posisjonene $x_0 + kp_i$ og $x_1 + kp_i$ for hvert primtall p_i i primbasen. Når verdien til $\log(q)$ er ca. null vet vi at den korresponderende q vil faktoriseres fullstendig over primbasen. Med å sile ut mener vi altså å trekke ifra logaritmen til p, derav navnet til metoden.

[58] Computational Methods for Factoring Large Integers – Marvin Wunderlich

KSM algoritmen er forskjellig ifra DB i steg 1 og 2 og disse følger nå:

1. Plasser i W lagringsplasser verdiene $\log(q)$ der q er definert over.

2a. For hver p_i i primbasen finn den minste x_0 og x_1 slik at $p_i \mid q(x_m)$, $m = 0,1$. Så subtraherer vi for alle k med $k \cdot p_i < W$, $\log p_i$ fra $\log(q(x_m + k \cdot p_i))$, $m = 0,1$.

2b. For hver x der $\log q(x)$ er "nær" null, faktoriser heltallsverdien til $q(x)$ over primbasen.

Kjøretiden til Kvadratisk sil metoden er antatt å være $\exp\left(\sqrt{\log n \operatorname{loglog} n}\right)$ [59]. Dette er endel bedre enn delbrøkoppspaltningsmetoden. Forskjellen er konstanten i eksponenten. Mer om kjøretiden har jeg ikke tenkt å si.
Når det gjelder forholdet til RSA er som sagt Kvadratisk sil metoden den største trusselen mot dette systemet. Det finnes ingen måter å konstruere tallet n på slik at det blir vanskeligere å faktorisere n enn hvilket som helst annet tall med samme størrelse. Det eneste man kan gjøre er å lage n tilstrekkelig stor.

5.9 Den diskrete logaritme

5.9.1 Generelt

Hvis G er en endelig (multiplikativ) gruppe, b et element i G og y et element i G som er en potens av b, da er **den diskrete logaritmen** til y med base b elementet x slik at $b^x = y$.

Grunnen til at det er interessant å se på den diskrete logaritme er at sikkerheten til RSA-systemet er avhengig av de to algoritmene faktorisering og den diskrete logaritme. Vi kan knekke koden ved å faktorisere n slik at vi finner $\phi(n)$, og dermed dekrypteringsnøkkelen d. En annen måte er å beregne den diskrete logaritmen. Dvs. finne d, gitt P, C og n, slik at $C^d \equiv P \pmod{n}$. Sikkerheten til Pohlig-Hellman metoden baserer seg også på at det er vanskelig å beregne diskrete logaritmer. Her må vi, gitt P,C og p finne en e slik at $P^e \equiv C \pmod{p}$, eller finne d slik at $C^d \equiv P \pmod{p}$.
Når vi regner over de reelle tall er ikke oppgaven med å finne $\log_b x$ spesielt vanskeligere enn å finne b^x, men går vi over til endelige grupper / kropper forandrer situasjonen seg dramatisk. Rask eksponensiering gjør det mulig å beregne $b^x = y$ for store x ganske raskt, mens er vi gitt elementene b og y, og skal finne x er det nærmest "umulig" for store tall.

5.9.2 Algoritme for å finne diskrete logaritmer i endelige kropper

Vi ønsker å beregne x slik at $b^x \equiv y \pmod{q}$. Anta at alle primfaktorene til $q - 1$ er små. Da eksisterer det en algoritme for å finne den diskrete logaritme til et element $y \in \mathbf{F}_q^*$ med basen b. For å gjøre det litt enklere antar vi at b er en generator i $\mathbf{F}_q^*$.

---

[59] Factoring – Carl Pomerance s.2

Algoritmen som følger er laget av Silver, Pohlig og Hellman [60].

For hvert primtall p som er en divisor i $q - 1$, beregner vi $r_{p,j} = b^{j(q-1)/p}$ for $j = 0, 1, 2, \dots, p - 1$. Denne tabellen med $\{r_{p,j}\}$-er gjør oss klar til å beregne den diskrete logaritmen til alle $y \in \mathbf{F}_q^*$.

Målet er nå å finne en x, $0 \le x < q - 1$, slik at $b^x = y$.
Hvis $q - 1 = \prod_{\text{alle } p \mid q-1} p^\alpha$ er primtallsoppspaltningen til $q - 1$, så er det kun nødvendig å finne $x \pmod{p^\alpha}$ for hver p som deler $q - 1$. Av dette er x entydig bestemt ved å bruke CRT. Vi skal nå vise for et gitt primtall $p \mid q - 1$ hvordan vi finner $x \bmod p^\alpha$.

Anta at $x \equiv x_0 + x_1 p + \dots + x_{\alpha-1} p^{\alpha-1} \pmod{p^\alpha}$ med $0 \le x_i < p$.
Å bestemme x er da ekvivalent med å bestemme $x_0, x_1, \dots, x_{\alpha-1}$.

$$y = b^x = b^{x_0} \cdot b^{x_1 p} \cdot b^{x_2 p^2} \cdot \dots \cdot b^{x_{\alpha-1} p^{\alpha-1}}.$$

For å finne x_0 ønsker vi å bli kvitt alt på høyre side unntatt b^{x_0}. Det gjør vi ved å opphøye alt i $(q - 1)/p$.

$$y^{(q-1)/p} = \left(b^x\right)^{(q-1)/p} = \left(b^{x_0} \cdot b^{x_1 p} \cdot \dots \cdot b^{x_{\alpha-1} p^{\alpha-1}}\right)^{(q-1)/p}.$$

Dette gir oss

$$y^{(q-1)/p} = \left(b^{(q-1)/p}\right)^{x_0} = r_{p,x_0}$$

Så sammenligner vi $y^{(q-1)/p}$ med $\{r_{p,j}\}_{0 \le j < p}$, og setter x_0 lik verdien til j der $y^{(q-1)/p} = r_{p,j}$.

For å finne x_1 erstatter vi y med $y_1 = y/b^{x_0}$.

$$y_1 = y\left(b^{-x_0}\right) = \left(b^{x_1 p} \cdot b^{x_2 p^2} \cdot \dots \cdot b^{x_{\alpha-1} p^{\alpha-1}}\right)$$

$$\left(y_1\right)^{(q-1)/p^2} = \left(b^{(q-1)/p}\right)^{x_1} = r_{p,\, x_1}$$

Nå kan vi sammenligne $y_1^{(q-1)/p^2}$ med $\{r_{p,j}\}$ og sette x_1 lik verdien til j der $y_1^{(q-1)/p^2} = r_{p,j}$.
Slik fortsetter vi til vi har funnet alle x_i-ene:

$$y_\alpha = y_{\alpha-1} b^{-x_{\alpha-1} p^{\alpha-1}}$$

$$\left(y_\alpha\right)^{(q-1)/p^{\alpha+1}} = r_{p,x_\alpha}$$

Når vi er ferdig med dette vil vi ha funnet $x \bmod p^\alpha$.
For å finne x modulo q må prosedyren over utføres for hver $p^\alpha \mid q - 1$, slik at vi kan vi bruke CRT for å finne x.
Algoritmen virker bra når primtallene som er divisorer i $q - 1$ er små. Beregningen av tabellen

[60] A Course in Number Theory and Cryptography – Neal Koblitz s.98
Kryptografi – Ben Johnsen s.35

$\{r_{p,j}\}$ og sammenligningen av $y_i^{(q-1)/p^{i+1}}$-ene med tabellen vil ta lang tid hvis $q-1$ er delelig med et stort primtall.

Kjøretiden for denne algoritmen er ifølge McCurley [61] $O\left(\sum_{i=1}^{\alpha} c_i\left(log\ q + \sqrt{p_i}\ log\ p_i\right)\right)$.

I forhold til Pohlig-Hellman algoritmen er det dermed viktig å velge p slik at $p-1$ har minst en stor primfaktor.

For de pr. idag raskeste algoritmene for å beregne den diskrete logaritme er kjøretiden begrenset av [62] $\exp\left(\sqrt{\log q \text{ loglog } q}\right)$.

5.9.3 Et eksempel på den diskrete logaritme

La oss beregne $3^x \equiv 7 \pmod{17}$

$q = 17,\ q-1 = 16 = 2^4,\ b = 3,\ y = 7.$

Vi må finne $x \equiv (x_0 + 2x_1 + 2^2x_2 + 2^3x_3) \bmod 2^4$.

$r_{p,j}$-ene blir :

	j	0	1
$r_{p,j} =$	3^{8j}	1	-1

For å finne x_0 beregner vi $y^{(q-1)/p} = 7^8 = -1$ og ser etter i tabellen av $r_{p,j}$-er og finner $x_0 = 1$.

$y_1 = y\left(b^{-x_0}\right) = 7 \cdot 3^{-1} = 8 \pmod{17}$

$(y_1)^{(q-1)/p^2} = 8^4 = -1 \Rightarrow x_1 = 1$

$y_2 = y_1\left(b^{-x_1 p}\right) = 8 \cdot 3^{-2} = -1 \pmod{17}$

$(y_2)^{(q-1)/p^3} = (-1)^2 = 1 \Rightarrow x_2 = 0$

$y_3 = y_2\left(b^{-x_2 p^2}\right) = -1 \cdot 3^0 = -1 \pmod{17}$

$(y_3)^{(q-1)/p^4} = -1 \Rightarrow x_3 = 1$

Som gir oss at $x \equiv (1 + 2 \cdot 1 + 2^3 \cdot 1) = 11 \pmod{17}$. Vi har da at $3^{11} \equiv 7 \pmod{17}$.

---

[61] The Discrete Logarithm Problem – Kevin S. McCurley s.11
[62] Cryptography and Data Security – Denning s.103

5.10 Konklusjon

Sikkerheten til RSA-kryptosystemet baserer seg på bl.a. at det er vanskelig å faktorisere store heltall som er produktet av bare to primfaktorer p og q. I forhold til Pollard's $p - 1$, elliptisk kurve og endel andre algoritmer som oppdager små faktorer raskere enn store, bør vi velge primtallene på en spesiell måte. Spesielt gjelder formen på $p - 1$ og $p + 1$.

Primtallene bør være på formen $p = 2p' + 1$ og $q = 2q' + 1$ hvor p' og q' er odde primtall. $p' - 1$ og $q' - 1$ bør også ha en stor primfaktor. Det samme gjelder for $p + 1$ of $q + 1$. $\gcd(p - 1, q - 1)$ må være liten, og det bør være noen få siffers forskjell på lengden til p og q. Velges primtallene etter disse kriteriene gjør man livet surest mulig for kryptoanalytikeren [63].

I tillegg til disse kriteriene må primtallene velges tilstrekkelig store. I dag faktoriseres det tall på rundt 100 siffer [64] med kvadratisk sil metoden. Problemet i forhold til KS er at man ikke kan konstruere primtall som faktoriseringen vanskelig. Eneste muligheten er å velge p og q store nok. Vanlige størrelser på p og q er (pr. 1.3-90) fra 75 siffer og oppover. Ofte anbefales 100 siffer.

Spranget fra faktorisering av 100 sifrete tall til 200 sifrete tall er enormt, og er neppe gjort over natta. Det gjør RSA til et meget sikkert kryptosystem. I tillegg har krypteringshastigheten steget ganske betraktelig i den siste tiden. Tar vi med alle de innbygde fordelene med RSA, fører det til at fremtiden egentlig ser ganske lys ut for RSA algoritmen.

Kompleksiteten til den diskrete Logaritme er også med på å gjøre både RSA og Pohlig-Hellman systemene til meget sikre kryptosystemer.

---

[63] Cryptography and Data Security – Denning s.106
[64] Factoring – Carl Pomerance s.28

Litteratur som er brukt til del 5

Cryptography and data security
D. Denning, Addison-Wesley 1983

Factorization and Primality Tests
John D. Dixon, American Math Monthly Jun/Jul 84

The Art of Computer Programming : Seminumerical Algorithms
Donald E. Knuth, Addison-Wesley 2.Utgave 1981

A Course in Number Theory and Cryptography
Neal Koblitz, Springer-Verlag 1987

Introduction to Elliptic Curves and Modular Forms
Neal Koblitz, Springer-Verlag 1984

Factoring Integers with Elliptic Curves 2 artikler
H. W. Lenstra

The Discrete Logarithm Problem
Kevin McCurley, Artikkel Juni 1989

Speeding the Pollard and Elliptic Curve Methods of Factorization
Peter L. Montgommery, Math of Computations Vol.48 no.177 Jan.87

A Monte Carlo Method for Factorization
J. M. Pollard, BIT 15 1975

Factoring
Carl Pomerance, Aug 89

Prime Numbers and Computer Methods for Factorization
Hans Riesel, Birkhäuser 1985

Computational Methods for Factoring Large Integers
M. C. Wunderlich, ABACUS Vol.5 no.2 1988

Tillegg 1: Diverse prosedyrer i C

Alt av prosedyrer er programmert av Tore Brattli. Prosedyrene er skrevet i programspråket C og kompilatoren Lightspeed C ver. 3.01 ble brukt.

Addisjon

Addisjon av 2 store tall. Svaret returneres gjennom x. Det første tallet i hver array inneholder lengden til tallet. Det korteste tallet trenger ikke være nullstilt i "toppen".

```
void  plus(x,y)
long *x, *y;
{
      long mente,t;
      int i,max;
      if (y[0]>x[0])
      {
            max=y[0];
            for (i=x[0]+1;i<=max;i++) x[i]=0;
      }
      else
      {
            max=x[0];
            for (i=y[0]+1;i<=max;i++) y[i]=0;
      }
      mente=0;
      for (i=1; i<=max; i++)
      {
            t=x[i]+y[i]+mente;
            x[i]=t%GRUNNTALL;
            mente=t/GRUNNTALL;
      }
      x[i]=mente;
      x[0]=max;
      if (mente>0) x[0]++;

}
```

Multiplikasjon

Multiplikasjon av et stort tall med et lite. Svaret returneres i egen variabel. Trenger ikke nullstilte arrays. Det første tallet i hver array inneholder lengden til tallet.

```
void  multipliser(langtall,tall,langsvar)
long *langtall, tall, *langsvar;
{
int i;
long mente,t;

      mente=0;
      for (i=1; i<=langtall[0]; i++)
      {
            t=langtall[i]*tall+mente;
            langsvar[i]=t % GRUNNTALL;
            mente=t/GRUNNTALL;
      }
      langsvar[i]=mente;
      if (mente>0)
      langsvar[0]=langtall[0]+1;
      else langsvar[0]=langtall[0];
}
```

Multiplikasjon av et stort tall med et lite. Svaret returneres gjennom langtall. Trenger ikke nullstilte arrays. Det første tallet i hver array inneholder lengden til tallet.

```
void  mult(langtall,tall)
long *langtall,tall;
{
int i;
long mente,t;

      mente=0;
      for (i=1; i<=langtall[0]; i++)
      {
            t=langtall[i]*tall+mente;
            langtall[i]=t % GRUNNTALL;
            mente=t/GRUNNTALL;
      }
      langtall[i]=mente;
      if (mente>0)
      langtall[0]++;
}
```

Divisjon

Divisjon av et langt tall med et kort. Resten returneres gjennom r og a blir ikke forandret.

```
long divkortrest(a,b,q)
long *a,b,*q;

{
      int m;
      long r;
      m=q[0]=a[0];
      r=0;
      while (m>0)
      {
            q[m]=(r*GRUNNTALL+a[m])/b;
            r=(r*GRUNNTALL+a[m])%b;

            m--;
      }
      if (q[q[0]]==0) q[0]--;
      a[0]=1;
      return r;
}
```

Divisjon av et langt tall med et kort. Resten returneres gjennom r og a returnerer kvotienten

```
unsigned long divkort(a,b)
unsigned long *a,b;

{
	int m;
	unsigned long r,t;

	r=0;
	m=a[0];
	while (m>0)
	{
		t=a[m];
		a[m]=(r*GRUNNTALL+a[m])/b;
		r=(r*GRUNNTALL+t)%b;

		m--;
	}
	if (a[a[0]]==0) a[0]--;
	return r;
}
```

Timer

Prosedyrer for å finne kjøretida til programmene. Dette er praktisk siden det er lett å finne tiden på nøyaktig det som er interessant f.eks multiplikasjonen og ikke konvertering av tall eller utskrift.

```
long timerstart()
{
	return TickCount();
}


long timerstopp(start)
long start;
{
	long tid,stopp,t,m,s,h;

	stopp=TickCount();
	tid=stopp-start;
	t=tid/216000;
	tid%=216000;
	m=tid/3600;
	tid%=3600;
	s=tid/60;
	tid%=60;
	h=tid*100/60;
	printf("\nmin%ld\tsek%ld\thun%ld\n",m,s,h);
}
```

Innlesning av tall

Leser inn tall til programeksemplene. De 2 neste prosedyrene er spesielt beregnet for tall med grunntall som er en tierpotens. Det er praktisk å lagre lange tall med det minst signifikante siffer i den laveste delen av arrayet. Det er imidlertid lettere å begynne med det mest signifikante siffer når tallet skal skrives inn. Derfor brukes et array lesinn til å legge tallet i midlertidig mens det leses inn. Når innlesningen er ferdig "snur" jeg tallet og konverterer det om til det ønskede tallsystem. Konverteringen er i dette tilfell kun et spørsmål om å gruppere desimalsifrene.

```
void lesinnettall(tall)
long *tall;
{
	long lesinn[ARRAYLENGDE*4];
	int i,ex,y,n,x;
	double b;

	b=log10(GRUNNTALL);

	i=0;
	lesinn[i]=getchar()-48;
	while (lesinn[i]<0 || lesinn[i]>9) lesinn[i]=getchar()-48;
	while (lesinn[i]>=0 && lesinn[i]<=9)
	{
		i++;
		lesinn[i]=getchar()-48;
	}
	ex=1;y=0;
	for (x=1;x<=i/b+1;x++) tall[x]=0;
	for (x=i;x>0;x--)
	{
		n=y/b+1;
		y++;
		tall[n]+=lesinn[x-1]*ex;
		ex*=10;
		if (ex >= GRUNNTALL) ex=1;
	}
	tall[0]=(i+b-1)/b;
	printf("\n");
}
```

Denne prosedyren er et tillegg til den forrige dersom lengden til de innleste tallene skal være en potens av to. n er lengde til tallene.

```
void raskinnles(a,b,n)
int *a,*b,*n;
{
	int i,lengde;
	int k;
	int c,d;

	lesinnettall(a);
	lesinnettall(b);
	c=floor(log10(a[0])/log10(2)-.000001)+1;
	d=floor(log10(b[0])/log10(2)-.000001)+1;
	if (c>d) k=c;
	else k=d;
	lengde=pow(2,k);
	for (i=a[0]+1;i<=lengde;i++) a[i]=0;
	for (i=b[0]+1;i<=lengde;i++) b[i]=0;
	a[0]=b[0]=1;
	*n=lengde;
}

void skrivarray(tall,n)
int *tall,n;
{
	int i;
	for (i=n;i>0;i--) printf("%d ",tall[i]);
	printf("\n");
}
```