itSik 2009, projektopgave

Torsten Jordt, 19751241

10.10.2009

En undersøgelse af faktoriseringsalgoritmen Pollard p-1

Indhold:

1 Opgavens mål og rammer

Denne rapport er en besvarelse af en projektopgave der udgør kurset "IT-sikkerhed i praksis". Opgaven er afslutningen på et forløb, der blev indledt med fagpakkerne "Introduktion til it-sikkerhed" og "Kryptologi". De tre kurser udgjorde tilsammen fagpakken "It-sikkerhed" som blev udbudt af IT-vest i 2009.

Opgavens overordnede formål var at undersøge og implementere Pollard's p − 1 algoritme som Stinson formulerer den. Fremgangsmåden i opgaven har været at implementere forskellige varianter af Stinson's formulering, undersøge hvordan de opfører sig, forklare hvorfor de opfører sig som de gør, og sluttelig udnytte denne viden til forsøg på optimering af Stinson's formulering af algoritmen.

Rammen for implementeringerne har været "standard Java" på en "standard-pc". Målet har ikke været at lave hurtige implementeringer, men derimod implementeringer som var "funktionelt effektive". Dvs. implementeringer som i mindre grad opgiver at faktorisere, og som ikke nødvendigvis kræver "forhåndsvalg" af input ud over det tal der ønskes faktoriseret.

Der er tale om en enmandsopgave, og den blev løst på en traditionel pc med traditionel software. Pc'en var en Dell Dimension 5150 med 1GB RAM og en Intel Pentium D CPU 2.8 GHz. Styresystemet var Windows XP SP3 og programomgivelserne var Java SDK 1.6 og JUnit3.

Fra java's værktøjskasse blev benyttet bl.a. klassen *BigInteger* og de metoder den som standard er udstyret med (*multiply, subtract*, *add*, *gcd*, *modPow*, *bitLength*, *isProbablePrime*, etc.).

Besvarelsen af opgaven baserer sig i det væsentlige på følgende materiale:

- Ivan Damgård: An Introduction to some Basic Concepts in IT, Security and Chryptography (kursusnoter 2009)
- Jesper Buus Nielsen: Introduction to Moderns Cryptography (kursusnoter 2009)
- Douglas R. Stinson: Chryptography Theory and Practice, 3rd edition

Rapporten refererer til dette materiale med betegnelserne hhv. Damgård, Nielsen og Stinson.

Kildekode, målinger og testdata (primtalsprodukter) findes her: http://itsik2009projekt.jordfugl.dk. Jeg har tilstræbt en konsekvent navngivning, således at der skulle være en nogenlunde entydig sammenhæng mellem kildekode, målinger, testdata og rapport. Målingerne findes som kolon-separeret tekst i csv-filer, som kan åbnes med et regneark. Vær i givet fald opmærksom på to forhold:

- Regnearksprogrammet kan være ude af stand til at vise tal med mange cifre. Filer indeholdende store tal (typisk på 30 bits og derover) bør åbnes med en almindelig teksteditor.
- Kolonner skal muligvis skal have udvidet bredden for at "lange" tal og navne bliver fuldt synlige.

Den begrænsede maskinkraft har påvirket valget af testdata, således at der hovedsageligt er undersøgt produkter af meget små primtal (dvs. primtal op til 32 bits). Og ved målinger på mange produkter ad gangen har det været nødvendigt at basere sig på produkter af endnu kortere primtal (typisk 20-24 bits). Men der er tilstræbt en systematik og opbygning af java-programmerne, så undersøgelsen i princippet kan gentages med vilkårligt store primtal.

Sidst men ikke mindst: Tak til Jesper Buus Nielsen uden hvis hjælp jeg aldrig havde forstået det jeg så. Og tak til min familie der sjældent så mig mens jeg så.

2 Introduktion til faktoriseringsalgoritmer og Pollard p-1

Generelt er udgangspunktet for en faktoriseringsalgoritme et tal n, der formodes at være produktet af to primtal p og q. Algoritmen tager som input et n, og den skal beregne enten p eller q (eller opgive at gøre det)

En sådan faktorisering er interessant fordi den eksempelvis kan bruges til at bryde RSA-kryptering. Kort fortalt bygger krypteringen i RSA på følgende principper (jf. Damgård):

- Kryptering sker med brug af en offentlig nøgle bestående af et talpar (n,e). En krypteret meddelelse c beregnes med udtrykket $c = m^e \bmod n$

- Dekryptering sker med brug af en hemmelig nøgle bestående af et talpar (n,d). En dekrypteret meddelelse m beregnes med udtrykket $m = c^d \bmod n$
- n er et produkt af to forskellige hemmelige primtal p og q. Dvs. $n = pq$
- d kan beregnes ud fra sammenhængen $ed = 1 \bmod (p-1)(q-1)$

Med andre ord: Kan man faktorisere n, er den hemmelige nøgle afsløret.

Pollard p − 1 algoritmen er en faktoriseringsalgoritme der blev formuleret af John Pollard i 1974. Den egner sig til faktorisering af tal der er sammensat af faktorer med særlige egenskaber. Det er simpelt at vælge p og q, så algoritmen vanskeligt kan faktorisere $n = pq$, og dermed bryde krypteringen i RSA. Dette beskrives nærmere i et følgende afsnit.

Trods dette er algoritmen relevant at undersøge nærme som et led i et undervisningsforløb. Den bygger på få (og enkelt formulerede) egenskaber for $\mathbb{Z}_n$, og implementeringen er enkel. Men undersøgelser af algoritmen afdækker (ikke overraskende) ikke-triviel matematisk kompleksitet, og arbejdet med implementeringer af algoritmen breder sig naturligt til andre områder af generel interesse. Eksempelvis generering af primtal med specifikke egenskaber, overvejelser om hvad der nærmere kan ligge i udsagnet "effektiv implementering" af en faktoriseringsalgoritme og hvordan dette kan effektueres, samt selve det at sætte sig ind i teoridannelser på området.

3 Pollard p-1 og Stinson's fremgangsmåde

Udgangspunktet for Pollard p-1 er et n der skal faktoriseres, og en på forhånd givet øvre grænse B. Antag følgende:

- Et primtal p faktoriserer n
- For alle primtalspotenser q der opfylder $q \mid (p-1)$ er $q \leq B$ (1)

3.1 Stinson's fremgangsmåde

Antagelsen (1) betyder at der gælder at $(p-1) \mid B!$.

Stinson formulerer Pollard p-1 algoritmen således:

Stinsons formulering af Pollard p-1

input : n, B
$a \leftarrow 2$
for $j \leftarrow 2$ **to** B
  $a \leftarrow a^j \bmod n$
$d \leftarrow \gcd(a-1, n)$
if $1 < d < n$ **then**
  return d
else
  return *failure*

Fremgangsmåden er implementeret i java-klassen `Pollard_p_1_Stinson`.

3.2 Derfor faktoriserer Stinson's fremgangsmåde

Da

$$2^{B!} = 2^{1\times 2\times ..\times B} = (...((2^1)^2)...)^B$$

må der ved afslutningen af for-løkken gælde at

$$a \equiv 2^{B!} \pmod{n}$$

Da vi har antaget at $p \mid n$, giver regnereglerne for kongruens-relationer at

$$\begin{aligned} &a \equiv 2^{B!} \pmod{n} \Rightarrow \\ &n \mid a - 2^{B!} \Rightarrow \\ &p \mid a - 2^{B!} \Rightarrow \\ &a \equiv 2^{B!} \pmod{p} \end{aligned}$$

Nu trækkes Fermat's lille teorem af stald. For et vilkårligt primtal p og et vilkårligt tal b gælder at

$$b^p \equiv b \pmod{p}$$

Hvis ydermere p og b er *coprimes* (dvs. $\gcd(p,b) = 1$) gælder at

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

For $b = 2$ er $\gcd(p,b) = 1$ (fordi p jo er et primtal), og dermed gælder at

$$2^{p-1} \equiv 1 \pmod{p}$$

Vi er nu klar til at etablere sammenhængen mellem a og p . Da $(p-1) \mid B!$ er $(p-1)k = B!$ for et eller andet tal k . Dermed gælder at

$$\begin{aligned} &a \equiv 2^{B!} \pmod{p} \Rightarrow \\ &a \equiv 2^{(p-1)k} \pmod{p} \Rightarrow \\ &a \equiv (2^{(p-1)})^k \pmod{p} \Rightarrow \\ &a \equiv 1^k \pmod{p} \Rightarrow \\ &a \equiv 1 \pmod{p} \end{aligned}$$

Og dermed (pr. definition af kongruens):

$$p \mid (a-1)$$

Når algoritmen har afsluttet sin for-løkke er situationen altså denne:

$$\begin{aligned} &p \mid n \Leftrightarrow pk' = n \\ &p \mid (a-1) \Leftrightarrow pk'' = a-1 \end{aligned}$$

for visse $k' > 0$ og $k'' > 0$.

Hvis $a > 1$ vil algoritmen således beregne

$$d = \gcd(a-1, n) = \gcd(pk'', pk') \geq p > 1$$

Da algoritmen beregner a som et tal $\bmod n$ er $0 \le a < n$ og dermed $a - 1 < n$. Derfor vil algoritmen altid (for $a > 1$) beregne

$$d = \gcd(a-1, n) < n$$

Med andre ord: Hvis blot $a \neq 1$ vil d være en faktorisering af n, såfremt forudsætningen (1) holder.

Hvis forudsætningen (1) ikke er opfyldt, holder $p \mid (a-1)$ ikke nødvendigvis, og vi kan få beregnet $d = 1$. Men der vil forsat gælde $d < n$.

Som Stinson også nævner, er d kun divisor såfremt $a \neq 1$. Men Stinson behandler dog ikke denne situation nærmere, og om fremgangsmåden skriver han generelt at "*if we increase the size of B drastically, say to $\sqrt{n}$, then the algorithm is guaranteed to be successful*".

Men det vil vise sig, at man ikke bare kan øge størrelsen af B og forvente samme resultat at Stinsons formulering af Pollard p-1. Allerede nu ses let, at hvis algoritmens for-løkke én gang har beregnet $a = 1$ for et givet B, så giver det ikke mening at øge størrelsen af B. Og det vil vise sig, at der findes produkter af svage primtal, som Pollard p-1 ikke kan faktorisere uanset valget af B.

3.3 Imødegåelse af truslen fra Pollard p-1

Generelt skal man naturligvis altid vælge sit primtalsprodukt som et produkt af store primtal p. Men man skal derudover vælge sit primtalsprodukt så forudsætningen (1) for Pollard p-1 kun vanskeligt bliver opfyldt. Det gøres ved at vælge p så $p-1$ har store primtalsfaktorer. Eksempelvis ved at lade p være et stærkt primtal (dvs. at p kan skrives som $p = 2p' + 1$ hvor p' er et primtal). Så bliver $p-1$ faktoriseret til $p - 1 = 2p'$, dvs. med netop en stor primtalsfaktor.

Men der findes faktisk produkter af svage primtal som Pollard p-1 ikke kan faktorisere overhovedet. Dette redegøres der nærmere for senere.

3.4 Simpel iterativ fastsættelse af B

Nielsen anviser en fremgangsmåde til at undgå at skulle give B som input. Man skal (groft sagt) blot udføre Pollard p-1 for $B = 2,4,8,...,2^k,...$ indtil man har fået beregnet en faktorisering eller B bliver for stor (dvs. $B > \sqrt{n}$). Eller at $a = 1$.

En umiddelbar fremgangsmåde til realisering af dette kunne være denne:

Pollard p-1 med iteration over B

input : n
$B \leftarrow 1$
$result \leftarrow failure$
while $result = failure$ **and** $B \leq \sqrt{n}$ **do**
$B \leftarrow 2B$
$result = \textbf{Pollard p} - \textbf{1}(n, B)$
if $\textbf{Pollard p} - \textbf{1}.a = 1$ **then**
return $failure$
return $result$

Fremgangsmåden er implementeret i java-klasserne `Pollard_p_1_Stinson` og `Pollard_p_1_Native`.

Betingelsen "$\textbf{Pollard p} - \textbf{1}.a = 1$" skal forstås således: "skaf den seneste værdi af a som er beregnet af $\textbf{Pollard p} - \textbf{1}$-algoritmen, og sammenlign denne værdi med 1. Testen på betingelsen er en optimering af hastigheden; når $a = 1$ giver det ingen mening at fortsætte iterationen over B.

Det ses, at fremgangsmåden vil resultere i én af følgende tre tilstande:

- $result = failure$ fordi $B > \sqrt{n}$
- $result = failure$ fordi $a = 1$
- $result = \gcd(a - 1, n)$

3.5 En simpel transformation af Stinson's algoritme

Analysen af Pollard p-1 og Stinson's implementering viser at en algoritme også kunne formuleres således:

Pollard p-1 (native)

input : n, B
$a \leftarrow 2$
for $j \leftarrow 2$ **to** B
$a \leftarrow a^j \bmod n$
if $a = 1$ **then**
return $failure$
$d \leftarrow \gcd(a - 1, n)$
if $d > 1$ **then**
return d
else
return $failure$

Fremgangsmåden er implementeret i java-klassen `Pollard_p_1_Native`.

I forhold til Stinsons algoritme er der lagt en test ind på hver beregning af a, og betingelsen $d < n$ er fjernet. Beregnes $a = 1$ er der ingen grund til at fortsætte, og algoritmen melder fejl.

Funktionelt opfører begge fremgangsmåder sig ens udadtil, forstået på den måde at de vil producere identisk output hvis de gives identisk input. Men det vil vise sig, at testen på $a = 1$ i fremgangsmåden Native danner grundlag for en væsentlig forbedring af evnen til at faktorisere.

Hastigheden blev målt på 10 produkter af to 32 bits svage primtal:

			Stinson			Native		
n	p	q	Tid/sek	Resultat	k_B	Tid/sek	Resultat	k_B
4611686138686472687	2147483659	2147483693	95	2147483659	22	96	2147483659	22
4611686181636145867	2147483659	2147483713	0	2147483713	12	0	2147483713	12
4611686246060655637	2147483659	2147483743	96	2147483659	22	96	2147483659	22
4611686319075100043	2147483659	2147483777	0	2147483777	10	0	2147483777	10
4611686396384511767	2147483659	2147483813	95	2147483659	22	96	2147483659	22
4611686490873792763	2147483659	2147483857	95	2147483659	22	96	2147483659	22
4611686512348629353	2147483659	2147483867	0	2147483867	14	0	2147483867	14
4611686516643596671	2147483659	2147483869	96	2147483659	22	96	2147483659	22
4611686555298302533	2147483659	2147483887	95	2147483659	22	96	2147483659	22
4611686568183204487	2147483659	2147483893	0	2147483893	12	0	2147483893	12
Samlet tidsforbrug[1]			**573**			**574**		

Notation: $k_B = \{k \mid B = 2^k\}$. Der blev faktoriseret ud fra n alene, med iterativ fastsættelse af B.

Målingerne viser ikke nogen grund til at antage at fremgangsmåden beregner væsentligt langsommere (eller hurtigere) end Stinson's fremgangsmåde. På den ene side vil for-løkken og implementeringen stoppe med det samme når a beregnes til 1. På den anden side er der lagt en ekstra test ind på a. Men hverken potensopløftning af 1 eller en test for en konstant-værdi kan anses for specielt tunge operationer. Den tunge beregning er $a^j \bmod n$ for $a \neq 1$, og der vil være lige mange af disse beregninger i begge implementeringer.

3.6 En hurtigere iterativ fastsættelse af B

Den tidligere beskrevne fremgangsmåde til iterative beregning af B har den egenskab, at der foretages gentagne beregninger af potensopløftningen $a \leftarrow a^j \bmod n$, hvor j starter fra 2 hver gang der prøves med et nyt B. Nogle af disse beregninger kan spares, idet et forsøg med et nyt B kan starte på grundlag af de værdier der blev beregnet for den forudgående værdi af B.

[1] Målt for alle 10 produkter som en helhed

Pollard p-1 (B)

input : n
$a \leftarrow 2$
$B_0 \leftarrow 2$
$B \leftarrow 1$
while $B \leq \sqrt{n}$ **do**
$B \leftarrow 2B$
for $j \leftarrow B_0$ **to** B
$a \leftarrow a^j \bmod n$
if $a = 1$ **then**
return *failure*
$B_0 \leftarrow B + 1$
$d \leftarrow \gcd(a-1, n)$
if $d > 1$ **then**
return d
return *failure*

Fremgangsmåden er implementeret i java-klassen `Pollard_p_1_B`.

Hastigheden blev målt for et antal serier 10 forskellige primtalsprodukter. Hver serie var dannet af svage primtal på hhv. 22 bits, 24 bits, … 32 bits.

Tid/sek	22 bit	24 bit	26 bit	28 bit	30 bit	32 bit
Native	2	8	6	175	89	574
B	1	4	3	82	47	301
B/Native	50%	50%	50%	47%	53%	52%

Der blev faktoriseret ud fra n alene, med iterativ fastsættelse af B.

Det ses at der er opnået en reduktion af beregningstiden på ca. 50%. Det er som forventet, idet halveringen kan forklares med at der ikke laves overflødige gentagelser af potensopløftningen $a \leftarrow a^j \bmod n$. Antag nemlig at beregningstiden i det væsentlige stammer fra potensopløftningen. Lad T_{Native} betegne antal potensopløftninger i fremgangsmåden Native, lad T_B betegne antal potensopløftninger i fremgangsmåden B. En grænse for værdien B der beregner en faktorisering er givet ved $B = 2^k$. Så er $T_{Native} = 2^0 + 2^1 + ... + 2^k$ og $T_B = 2^k$. Dermed fås for "store" k

$$\frac{T_{Native}}{T_B} = \frac{2^0 + 2^1 + ... + 2^k}{2^k} = 1 + \frac{2^0 + 2^1 + ... + 2^{k-1}}{2^k} = 1 + \frac{\sum_{i=0}^{k-1} 2^i}{2^k} = 1 + \frac{2^k + 1}{2^k} \approx 1 + 1 = 2$$

Dvs. at fremgangsmåden Native tager dobbelt så meget beregningstid som fremgangsmåden B, svarende til at fremgangsmåden B reducerer beregningstiden med ca. 50%.

Funktionelt opfører begge fremgangsmåder sig ens udadtil, forstået på den måde at de vil producere identisk output hvis de gives identisk input.

4 Analyse af Stinson's fremgangsmåde

4.1 Hvorfor antager a værdien 1?

Hvorfor antager a værdien 1? Det korte svar er: Fordi $n = pq$ hvor p og q er primtal.

Antag nemlig at $n = pq$ hvor p og q er forskellige primtal. Så er

$$ord(\mathbb{Z}_n) = \phi(n) = (p-1)(q-1)$$

Og dermed gælder for alle $x \in \mathbb{Z}_n$

$$x^{(p-1)(q-1)} \bmod n = 1$$

og specielt

$$2^{(p-1)(q-1)} \bmod n = 1$$

Med andre ord: På et eller andet tidspunkt vil a nødvendigvis antage værdien 1.

4.2 Findes der produkter af svage primtal som ikke kan faktoriseres?

Findes der produkter af svage primtal som Pollard p-1 ikke kan faktorisere? Det korte svar er: Ja.

Pr. definition af lcm (*Least Common Multiple*) gælder

$$(p-1) \mid lcm(p-1, q-1) \wedge (q-1) \mid lcm(p-1, q-1)$$

Af Fermats lille teorem kan udledes, at hvis p er et primtal og $(p-1) \mid a$ så er $x^a \bmod p = 1$[2]. Dermed gælder for alle $x \in \mathbb{Z}_n$ at

$$x^{lcm(p-1,q-1)} \bmod p = 1 \wedge x^{lcm(p-1,q-1)} \bmod q = 1$$

Chinese Reminder teoremet indebærer at der nødvendigvis må gælde for alle $x \in \mathbb{Z}_n$ at

$$x^{lcm(p-1,q-1)} \bmod n = 1$$

og specielt

$$2^{lcm(p-1,q-1)} \bmod n = 1$$

Det betyder, at når $lcm(p-1, q-1) \mid B!$ vil Pollard p-1 nødvendigvis beregne $a = 1$. Hvis dette sker "før" $(p-1) \mid B!$ eller $(q-1) \mid B!$ vil algoritmen altid fejle, uanset valg af B. Mere formelt udtrykt:

Hvis

---

[2] Hvis $(p-1) \mid a$ er $a = (p-1)k = pk - k$ for et eller andet k. Dermed fås for vilkårlige x at

$$x^a \bmod p = x^{pk-k} \bmod p = \frac{x^{pk}}{x^k} \bmod p = \frac{(x^p)^k}{x^k} \bmod p = \frac{(x)^k}{x^k} \bmod p = 1$$

hvor Fermats teorem blev benyttet i næstsidste lighedstegn

$$lcm(p-1,q-1) \mid B! \wedge \neg(p-1) \mid B'! \wedge \neg(q-1) \mid B'! \text{ for alle } B' < B$$

vil algoritmen fejle, uanset om B vælges større.

Dette er eksempelvis tilfældet for 520671797=16411*31727, svarende til $n = 520671797$, $p = 16411$ og $q = 31727$. Her bliver så

$$p-1 = 16410 = 2 \times 3 \times 5 \times 547$$
$$q-1 = 31276 = 2 \times 29 \times 547$$
$$lcm(p-1,q-1) = 2 \times 3 \times 5 \times 29 \times 547.$$

Det mindste (dvs. "første") B hvor $lcm(p-1,q-1) \mid B!$ er $B = 427$.

Men der findes ingen $B' < 427$ så $(p-1) \mid B'!$ eller $(q-1) \mid B'!$. Vi har altså her et $B = 427$ så $lcm(p-1,q-1) \mid B! \wedge \neg(p-1) \mid B'! \wedge \neg(q-1) \mid B'!$ for $B' < B$

Pollard p-1 algoritmen vil derfor fejle, uanset hvordan B vælges.

4.3 Skal man bare gøre B stor nok?

Skal man bare gøre B stor nok? Det korte svar er: Nej.

Det kan være nyttigt at gøre B større, men generelt er det ikke sikkert at det hjælper. Dette er eksempelvis tilfældet for 269485031=16411*16421, svarende til $n = 269485031$, $p = 16411$ og $q = 16421$. Her er

$$p-1 = 16410 = 2 \times 3 \times 5 \times 547$$
$$q-1 = 16420 = 4 \times 5 \times 821$$
$$lcm(p-1,q-1) = 3 \times 4 \times 5 \times 547 \times 821$$

Vælges eksempelvis $B = 547$ vil algoritmen faktorisere (fordi $(p-1) \mid 547!$). Men vælges eksempelvis $B = 821$ vil algoritmen ikke faktorisere (fordi $lcm(p-1,q-1) \mid 821!$).

Eksempelvis vil en iterativ forøgelse af B fra 2^k til 2^{k+1} betyde at algoritmen vil "misse" en faktorisering af $n = pq$, såfremt

$$1 < \gcd(2^{'B'!} - 1, n) < n \text{ for } 2^k < B' < 2^{k+1} \text{ og } lcm(p-1,q-1) \mid 2^{k+1}!$$

Med andre ord: Princippet at udføre Pollard p-1 for $B = 2,4,8,...,2^k,...$ har sin pris, nemlig at dette ikke beregner visse faktoriseringer, der ellers ville kunne have været beregnet for mere heldige valg af B.

5 Optimering af Stinson's fremgangsmåde

Hvad kan man gøre når først algoritmen har beregnet $a = 1$? Svaret viser sig at være simpelt: Man kan tage fat i den umiddelbart forudgående værdi af a og benytte den til beregningen $d \leftarrow \gcd(a-1,n)$. Forbløffende nok bevirker dette en markant forbedring af algoritmens evne til at faktorisere.

Fremgangsmåden Native kan således omformes til denne fremgangsmåde:

Pollard p-1 (a)

input : n, B
$a \leftarrow 2$
for $j \leftarrow 2$ **to** B
  $a_{previous} \leftarrow a$
  $a \leftarrow a^j \bmod n$
  if $a = 1$ **then**
    $d \leftarrow \gcd(a_{previous} - 1, n)$
    if $d > 1$ **then**
      return d
  return *failure*
$d \leftarrow \gcd(a - 1, n)$
if $d > 1$ **then**
  return d
else
  return *failure*

Fremgangsmåden er implementeret i java-klassen `Pollard_p_1_aB`.

Fremgangsmåden B ("hurtigere iteration over B") kan omformes til denne fremgangsmåde:

Pollard p-1 (aB)

input : n
$a \leftarrow 2$
$B_0 \leftarrow 2$
$B \leftarrow 1$
while $B \leq \sqrt{n}$ **do**
{
  $B \leftarrow 2B$
  for $j \leftarrow B_0$ **to** B
  {
    $a_{previous} \leftarrow a$
    $a \leftarrow a^j \bmod n$
    if $a = 1$ **then**
    {
      $d \leftarrow \gcd(a_{previous} - 1, n)$
      if $d > 1$ **then**
        return d
      return *failure*
    }
  }
  $B_0 \leftarrow B + 1$
  $d \leftarrow \gcd(a - 1, n)$
  if $d > 1$ **then**
    return d
}
return *failure*

Fremgangsmåden er implementeret i java-klassen `Pollard_p_1_aB`.

Hastigheden og effektiviteten blev målt på 1000 produkter to 24 bits svage primtal:

	B	aB
Antal faktoriseringer	941	1000
Funktionel effektivitet	94%	100%
Tid/sek.	244	245

Der blev faktoriseret ud fra n alene, med iterativ fastsættelse af B. Funktionel effektivitet er beregnet som antal faktoriseringer divideret med antal produkter.

Det ses at fremgangsmåden aB beregner ca. $\frac{1000-941}{1000} \approx 6\%$ flere faktoriseringer end fremgangsmåden B. Til gengæld tager den $\frac{245-244}{245} \approx 0.4\%$ længere tid. Altså stort set samme tidsforbrug.

Der er egentlig ikke noget overraskende i at der kan findes en faktorisering ved at prøve med en tidligere værdi af a (jf. afsnittet "Skal man bare gøre B stor nok?" skal man blot undersøge tilstrækkeligt mange tidligere værdier). Det overraskende er at man kan finde en faktorisering ved blot at tage den umiddelbart forudgående værdi af a.

6 Hvorfor fungerer den optimerede fremgangsmåde?

Hvorfor er fremgangsmåden aB effektiv? Det korte svar er: Fordi de B'er for hvilke Pollard p-1 faktoriserer udgør et interval.

6.1 Intervallet af de B'er for hvilke Pollard p-1 faktoriserer

Lad i det følgende n_1 og n_2 betegne primtallene p og q og lad $n = n_1 n_2$. Dvs. $(n_1, n_2) = (p, q)$ eller $(n_1, n_2) = (q, p)$. Lad endvidere

$$a[j] = a^{j!} \bmod n$$
$$a_1[j] = a^{j!} \bmod n_1$$
$$a_2[j] = a^{j!} \bmod n_2$$

Så indebærer Chinese Reminder teoremet at $a[j]$ kan udtrykkes ved $(a_1[j], a_2[j])$ og omvendt. Dvs.

$$a[j] = (a_1[j], a_2[j])$$

Forløbet af en faktorisering med Pollard p-1 kan dermed illustreres således:

j	$a[j]$	$a_1[j]$	$a_2[j]$
2	$\neq 1$	$\neq 1$	$\neq 1$
. . .	. . .	. . .	. . .
$j_{\min} - 1$	$\neq 1$	$\neq 1$	$\neq 1$
$j_{\min}$	$\neq 1$	1	$\neq 1$
. . .	. . .	. . .	. . .
$j_{\max}$	$\neq 1$	1	$\neq 1$
$j_{\max} + 1$	1	1	1
. . .	. . .	. . .	. . .

Notation: $\neq 1$ betegner en værdi der er forskellig fra 1.

Intervallet med de B'er for hvilke Pollard p-1 faktoriserer er fastlagt ved $[j_{\min}, j_{\max}]$. Dette gælder dog kun når der faktisk findes $j_{\min} \leq j_{\max}$. Intervallet kan være tomt, svarende til at der findes produkter af svage primtal, som Pollard p-1 ikke kan faktorisere (som vist tidligere).

6.2 Argumentation

At forløbet af en faktorisering faktisk forløber som illustreret ovenfor bygger på nogle simple sammenhænge. Disse er:

a) $a_1 = 1, a_2 \neq 1 \Rightarrow 1 < d < n$

b) $a_1 \neq 1, a_2 \neq 1 \Rightarrow d = 1$
c) $a_1 = 1, a_2 = 1 \Rightarrow d = n$
d) $a_k[j] = 1 \Rightarrow a_k[i] = 1 \; \forall \; i > j$

For nemheds skyld benyttes notationen a_1 for $a_1[j]$ og a_2 for $a_2[j]$. a) udtrykker situationen hvor Pollard p-1 faktoriserer. d) udtrykker "én gang 1 altid 1" (dvs. at når først a er beregnet til 1 vil a vedblive at få værdien).

a) $a_1 = 1, a_2 \neq 1 \Rightarrow 1 < d < n$
Udgangspunktet er:

$$\begin{cases} a \bmod n_1 = 1 \\ a \bmod n_2 \neq 1 \end{cases}$$
$$\Rightarrow \begin{cases} (a-1) \bmod n_1 = 0 \\ (a-1) \bmod n_2 \neq 0 \end{cases}$$
$$\Rightarrow \begin{cases} n_1 \mid (a-1) \\ \neg n_2 \mid (a-1) \end{cases}$$

Sæt $d = \gcd(n, a-1)$. Så gælder $n_1 \mid d$. Dermed fås $d \geq n_1 > 1$. Da $n_2 \mid n$ og $\neg n_2 \mid (a-1)$ er $d < n$.

b) $a_1 \neq 1, a_2 \neq 1 \Rightarrow d = 1$
Udgangspunktet er:

$$\begin{cases} a \bmod n_1 \neq 1 \\ a \bmod n_2 \neq 1 \end{cases}$$
$$\Rightarrow \begin{cases} (a-1) \bmod n_1 \neq 0 \\ (a-1) \bmod n_2 \neq 0 \end{cases}$$
$$\Rightarrow \begin{cases} \neg n_1 \mid (a-1) \\ \neg n_2 \mid (a-1) \end{cases}$$

Da $n = n_1 n_2$ er $d = \gcd(n, a-1) = 1$

c) $a_1 = 1, a_2 = 1 \Rightarrow d = n$
Udgangspunktet er:

$$\begin{cases} a \bmod n_1 = 1 \\ a \bmod n_2 = 1 \end{cases}$$
$$\Rightarrow \begin{cases} (a-1) \bmod n_1 = 0 \\ (a-1) \bmod n_2 = 0 \end{cases}$$
$$\Rightarrow \begin{cases} n_1 \mid (a-1) \\ n_2 \mid (a-1) \end{cases}$$

Da $d = \gcd(n, a-1)$ er $n_1 \mid d$ og $n_2 \mid d$ og dermed $d = n$

d) $a_k[j]=1 \Rightarrow a_k[i]=1 \; \forall \; i > j$
Udgangspunktet er (for $k=1,2$):

$$a^{j!} \bmod n_k = 1$$

Så bliver for $i > j$

$$a_k[i] = a^{i!} \bmod n_k = a^{j!\frac{i!}{j!}} \bmod n_k = (a^{j!})^{\frac{i!}{j!}} \bmod n_k = 1^{\frac{i!}{j!}} \bmod n_k = 1$$

6.3 Derfor fungerer den optimerede fremgangsmåde

Det ses nu hvorfor fremgangsmåden aB fungerer. Når fremgangsmåden har beregnet $a=1$ svarer det til at j har værdien $j_{\max}+1$, Fremgangsmåden tager den forudgående værdi af a, nemlig $a[j_{\max}]$ og beregner d ud fra denne værdi. Og det er netop en værdi som giver en faktorisering (såfremt $j_{\min} \leq j_{\max}$).

6.4 Eksempler på intervaller

I det følgende vises nogle eksempler på hvordan intervallet kan være opbygget. Alle eksempler er produceret og verificeret med JUnit3-klassen `AnalyzeInterval`.

Tilfældet 4288717=2053*2089 (et interval af længden 10):

j	$a[j]$	$a_1[j]$	$a_2[j]$
16	2601	548	512
17	378365	613	256
18	1283691	566	1045
19	1728627	1	1024
20	3112349	1	1828
21	2833141	1	457
22	3212946	1	64
23	2763339	1	1681
24	3212946	1	64
25	3691295	1	32
26	3711825	1	1761
27	119075	1	2
28	4229181	1	1045
29	1	1	1
30	1	1	1

Interval $=\{B \mid 19 \leq B \leq 28\}$

Tilfældet 17203204321=131101*131221 (et interval af længden 1):

j	$a[j]$	$a_1[j]$	$a_2[j]$
15	11315784624	64011	72910
16	2626071187	118157	76535
17	772209754	24864	105390
18	430142439	58	1
19	1	1	1
20	1	1	1

Interval $=\{B \mid B = 18\}$

Tilfældet 269485031=16411*16421 (et større interval):

j	$a[j]$	$a_1[j]$	$a_2[j]$
544	176436033	1372	8809
545	268160706	4966	5776
546	30306236	11530	9491
547	207041177	1	5209
548	258194264	1	6881
.	.	.	.
.	.	.	.
.	.	.	.
818	108099258	1	16236
819	91934423	1	9665
820	51629007	1	1383
821	1	1	1
822	1	1	1

Interval $= \{B \mid 547 \le B \le 820\}$

Tilfældet for 520671797=16411*31727 (hvor Pollard p-1 fejler):

j	$a[j]$	$a_1[j]$	$a_2[j]$
544	37615384	1372	18889
545	276891358	4966	9829
546	356540505	11530	24206
547	1	1	1
548	1	1	1

Interval $= \emptyset$

7 Et andet optimering af Stinson's fremgangsmåde

7.1 Fremgangsmåden X1

De foregående undersøgelser viser at en fremgangsmåde også kan formuleres således:

Pollard p-1 (X1)

input : n
$a \leftarrow 2$
$j \leftarrow 2$
while $a > 1$ **do**
{
  $a_{previous} \leftarrow a$
  $a \leftarrow a^j \bmod n$
  if $check_{n,a,j}$ **then**
  {
    $d \leftarrow \gcd(a_{previous} - 1, n)$
    if $d > 1$ **then**
      return d
  }
  $j \leftarrow j + 1$
}
$d \leftarrow \gcd(a_{previous} - 1, n)$
if $d > 1$ **then**
  return d
else
  return *failure*

Fremgangsmåden er implementeret i java-klassen `Pollard_p_1_X1`.

Notation: $check_{n,a,j}$ er en test på om der skal tjekkes på om "vi har ramt intervallet hvor Pollard p-1 faktoriserer".

Den afgørende forskel i forhold til Stinson's fremgangsmåde er at der hele tiden tjekkes på om der er fundet en kandidat til en faktor. Med Stinson's fremgangsmåde risikerer man bogstavelig talt at "skyde forbi målet", fordi der her kun tjekkes én kandidat. En anden fordel er at fremgangsmåden samler Stinson's

fremgangsmåde og ”gættet på rette interval” i ét hele. Og at fremgangsmåden tydeliggør og fokuserer det forhåndsvalg der skal træffes, nemlig ”gættet på rette interval”.

7.2 Strategier for check

Strategien for hvornår testen $check_{n,a,j}$ skal give **true** er et valg man skal træffe. Helt i lighed med fremgangsmåden B, hvor der tjekkes for $B = 2,4,8,...,2^k,...$. Så ”nissen flytter med” – man skal stadig træffe et forhåndsvalg. På den anden side giver fremgangsmåden mulighed for en meget fleksibel implementering af forskellige strategier. Java-implementeringen benytter derfor et *Strategy Design Pattern*[3].

[3] Beskrevet nærmere i Erich Gamma et al.: Design Patterns, Elements of Reusable Object-Oriented Software, 1995, p. 315

BitLength

En strategi for $check_{n,a,j}$ kan være at testen skal give **true** når bitlængden af j er øget med. Så vil fremgangsmåden i realiteten opføre sig som fremgangsmåden B.

Strategien er implementeret med java-klassen `Pollard_p_1_X1.BitLengthCheckStrategy`.

NoCheck

I princippet kunne man vælge at testen altid skal give **false** . Så vil fremgangsmåden stadig beregne det samme, men kan tage væsentligt længe tid, fordi det først "opdages til sidst" at "vi har ramt intervallet hvor Pollard p-1 faktoriserer". Hvis vi havde en forhåndsviden om at det pågældende interval er "kort", er det oplagt at testen altid skal give **false** . Problemet er at denne viden har man ikke. Helt svarende til at man ikke har viden til at vælge det præcise B i Stinson's fremgangsmåde.

Strategien er implementeret med java-klassen `Pollard_p_1_X1.NoCheckStrategy`.

BoundCheck

Man kan vælge en strategi der giver **true** når bitlængden af j er overstiger en vis grænse B . Så vil fremgangsmåden i realiteten opføre sig som fremgangsmåden Native, bortset fra at iterationen fortsættes hvis B er sat "for lavt".

Strategien er implementeret med java-klassen `Pollard_p_1_X1.BoundCheckStrategy`.

AnalyzeInterval

Man kan vælge en strategi der analyserer hvert skridt i udviklingen $a \leftarrow a^j \bmod n$. Fremgangsmåden kan så bruges til at producere de eksempler der er i afsnittet "Eksempler på intervaller".

Strategien er implementeret med java-klassen `AnalyzeIntervalStrategy`, og eksemplerne i afsnittet er produceret med JUnit3-klassen `AnalyzeInterval`.

7.3 Hastighed

Hastigheden i forhold til fremgangsmåden aB blev målt på 10 produkter af to 28 bits svage primtal:

			aB	X1(BitLengthCheck)	X1(NoCheck)
n	p	q	Tid/sek	Tid/sek	Tid/sek
18014408441595161	134217757	134217773	0	0	14
18014409246901703	134217757	134217779	1	1	14
18014409515337217	134217757	134217781	0	0	14
18014411931256843	134217757	134217799	0	0	14
18014412468127871	134217757	134217803	14	14	14
18014415152483011	134217757	134217823	15	15	15
18014421326499833	134217757	134217869	24	24	443
18014422400241889	134217757	134217877	0	0	14
18014424547726001	134217757	134217893	24	24	442
18014427768952169	134217757	134217917	3	3	14
Samlet tidsforbrug[4]			**80**	**81**	**997**

Der blev faktoriseret ud fra n alene, med iterativ fastsættelse af B .

Som ventet er der ingen væsentlig forskel i hastigheden på fremgangsmåderne aB og X1(BitLengthCheck). Og fremgangsmåden X1(NoCheck) tager som ventet længere tid. Der er tale om en væsentlig øget tidsforbrug, som illustrerer nødvendigheden af en strategi for $check_{n,a,j}$.

8 Sammenfatning

Det er vist, at Stinson's formulering simpelt kan modificeres så den faktoriserer bedre når B beregnes iterativt.

[4] Målt for alle 10 produkter som en helhed

Det er vist, at de B 'er for hvilke Pollard p-1 faktoriserer udgør et interval, og der er fundet et udtryk for den øvre grænse på dette interval. Specielt viser dette, at man generelt ikke kan øge værdien af B for at få Stinsons formulering til at faktorisere.

Det er vist at der findes produkter svage primtal som ikke kan faktoriseres af Stinson's fremgangsmåde, og det er beskrevet hvordan disse kan beregnes ud fra kendskab til produktets primtalsfaktorer.

Der er beskrevet en fremgangsmåde der (på grundlag af Stinson's formulering) på én gang favner både Stinson's fremgangsmåde (der forudsætter angivelse af et B) og en situation hvor man ikke angiver et B. Denne fremgangsmåde er lige så hurtig som Stinson's fremgangsmåde, samtidig med at den faktoriserer bedre. Fremgangsmåden fordrer ikke angivelse at et B, men derimod et bud på "intervallet af gode B'er".

Selvom målet med undersøgelsen ikke har været at lave hurtige implementeringer, er det nærliggende at kigge lidt på hvordan implementeringerne arter sig i en lidt større sammenhæng. Derfor afsluttes med en hastighedsmåling på tre udformninger af Pollard p-1, nemlig Stinson's, X1 og en udformning lavet af Paolo Ardoino.

Paul Ardoino har implementeret Pollard p-1 på en helt anden måde end Stinson, og angiver en benchmark af to lidt større produkter af stærke primtal (http://ardoino.com/9-maths-factoring-pollard/ refereret fra http://en.wikipedia.org/wiki/Pollard%27s_p_-_1_algorithm).

			Ardoino's egen tidsmåling[5]	Min implement. af Ardoino	Stinson	X1 (BitLengthCheck)
n	p	q	Tid/sek	Tid/sek	Tid/sek	Tid/sek
15236506168104630133	15337307	993427735919	22	485	218	114
3369738766071892021	204518747	16476429743	352	513	3810	1975
Samlet tidsforbrug[6]			**374**	**998**	**4028**	**2089**

Der blev faktoriseret ud fra n alene, med iterativ fastsættelse af B.

Det ses at Ardoino's implementering kan være markant hurtigere. Dette kunne tyde på, at skal Pollard p-1 optimeres hastighedsmæssigt, skal man nok kigge på andre udformninger af Pollard p-1 end den Stinson formulerer (og som X1 er baseret på).

At X1 er hurtigere end Stinson's fremgangsmåde bør ikke opfattes som at den hastighedsmæssigt er "bedre". Præmissen for Stinson's formulering er angivelse af en specifik øvre grænse B, og denne præmis er ikke opfyldt af ovenstående måling (fordi den fastsætter B iterativt). Opfyldte man præmissen og angav en "god" B, ville Stinson's algoritme være lige så hurtig som X1. Men man kan sige at X1 virker som en oplagt og ligefrem overbygning på Stinson's formulering, der udstiller en essentiel egenskab ved de værdier af B for hvilke Stinson's formulering vil faktorisere.

9 Appendix: Generering af primtalsprodukter til brug i undersøgelsen

Implementeringerne er afprøvet med produkter af svage primtal. Med et svagt primtal menes et primtal p der ikke er stærkt. Et stærkt primtal p er et primtal der kan skrives som $p = 2p'+1$ hvor p' er et primtal. Det giver ingen egentlig mening at undersøge tal der ikke er produkter af primtal, eller som er produkter af stærke primtal. Fordi Pollard p-1 her på forhånd vides at fejle.

Der er kun undersøgt primtalsprodukter af primtal med samme bitlængde. For at have et enkelt og ensartet grundlag for undersøgelsen har jeg fravalgt primtalsprodukter hvor kun det ene primtal i produktet er svagt, og primtalsprodukter med forskellig bitlængde af primtallene. Der beregnes ikke produkter af ens primtal, idet

[5] Hastigheden af Ardoino's algoritme kan variere væsentligt fra måling til måling (på det samme produkt). Dette skyldes at hans algoritme benytter et tilfældigt a som input til $\gcd$.

[6] Målt for begge produkter som en helhed

udgangspunktet for undersøgelsen er primtalsprodukter af forskellige primtal. Det afgøres om et tal er et primtal med en sikkerhed på $1-2^{-100}$.

Algoritmen der danner primtal er denne:

Generering af primtal af længde l

input : l
$n \leftarrow 2^{l-1}+1$
loop
  if n **is prime then**
    if $\frac{n-1}{2}$ **is prime then**
      output n (*as strong prime*)
    else
      output n (*as weak prime*)
  $n \leftarrow n+2$

Fremgangsmåden er implementeret i java-klassen `GeneratePrimeProducts`.

Altså en meget simpel algoritme. Men den er tilstrækkelig. I forhold til faktorisering af de resulterende primtalsprodukter er tidsforbruget for denne algoritme marginalt.

Algoritmen der danner primtalsprodukter benytter outputtet fra primtalsalgoritmen som input er simpel:

Generering af primtalsprodukter

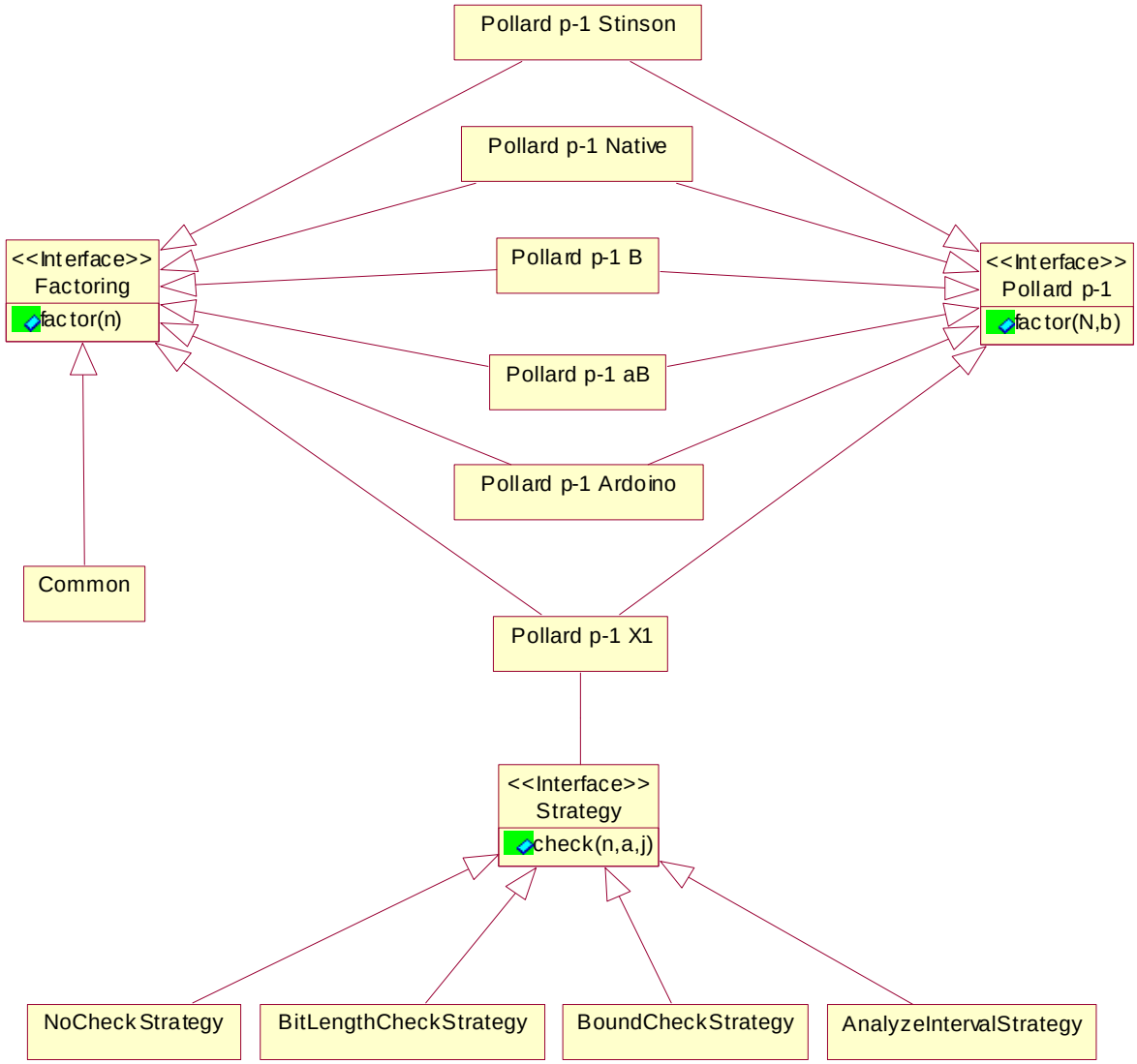
input : *list* (*of primes*)
for $i \leftarrow 1$ **to** $\#list$
  for $j \leftarrow i+1$ **to** $\#list$
    output $list_i \times list_j$

Fremgangsmåden er implementeret i java-klassen `GeneratePrimeProducts`.

Notation: $\#list$ betegner antallet af elementer i $list$, $list_i$ betegner det i'te element i $list$, og $\times$ betegner sædvanlig multiplikation.

10 Appendix: Implementering i java

Fremgangsmåderne til faktorisering er implementeret med disse java-interfaces og –klasser:



Interface	*Funktion*
`Factoring`	The essential interface for factoring a number in general
`Pollard_p_1`	The essential interface for factoring a number by means of Pollard p-1
`Strategy`	The interface for $check_{n,a,j}$ in the X1 implementation

Klasse	*Funktion*
`Pollard_p_1_Stinson`	Stinson's algorithm for factoring using Pollard p-1
`Pollard_p_1_Native`	A slight modification of Stinson's algorithm. Computes same results as Stinson's algorithm
`Pollard_p_1_B`	Based on Stinson's algorithm. B is iteratively computed
`Pollard_p_1_aB`	Based on Stinson's algorithm. B may be iteratively computed. If a=1 attempt to factor from the previous value of a.
`Pollard_p_1_X1`	An alternative version of Stinson's algorithm
`Pollard_p_1_Ardoino`	Factoring using Paolo Ardoino's implementation
`Common`	Trivial factoring. Contains also (not very fast) helper methods to be used in analysis of implementations: - `computePower` (computes k from a number of the form 2^k and k is an integer) - `factorize` (computes all factors of a number n) - `factorizeAsPrimePowers` (computes all prime power factors of a number n) - `factorsAsString` (returns a "pretty" string representation of a list of factors.) - `hasMultipleFactors` (determines if a list of factors contains multiple factors) - `lcm` (computes Least Common Multiple)
`NoCheckStrategy`	A $check_{n,a,j}$ which always returns **false**
`BitLengthCheckStrategy`	A $check_{n,a,j}$ which returns **true** when the bit length of j has increased
`BoundCheckStrategy`	A $check_{n,a,j}$ which returns **true** when j has reached a specific bound B
`AnalyzeIntervalStrategy`	A $check_{n,a,j}$ which analyzes the iteration $a \leftarrow a^j \bmod n$ (displays the values of B for which the algoritms factors).

Afprøvninger og undersøgelser af fremgangsmåderne er implementeret med følgende JUnit3-klasser:

Klasse	*Funktion*
`Benchmark`	Benchmarking and performance measurements
`TestAll`	Verification of implementations and findings
`TestCommon`	Verification of class Common and its methods
`TestCaseProject`	A project specific extension of the JUnit3 TestCase
`IntervalExamples`	Examples of intervals of B's for which Pollard p-1 factors

De afprøvede fremgangsmåder er implementeret således:

	Stinson	Native	B	a	aB	X1
`Pollard_p_1_Stinson`	x					
`Pollard_p_1_Native`		x				
`Pollard_p_1_B`			x			
`Pollard_p_1_aB`				x	x	
`Pollard_p_1_X`						x