Modulair rekenen en de Montgomery vermenigvuldiging

door Theo Kortekaas

Inleiding

Modulair rekenen is rekenen met resten. Als we een vliegreis maken van 31 uur en we vertrekken vandaag om 12:00 uur dan zijn we morgen 19:00 uur op onze bestemming; 7 uur later dan het vertrektijdstip vandaag.
Dat bereken we door tijdsduur van de reis te delen door 24; het aantal uren dat er in een etmaal zit. De rest tellen we op bij het tijdstip van vertrek en zo komen we op 19:00 uur als aankomsttijd. (Hierbij moeten we natuurlijk ook nog rekening houden met het tijdsverschil tussen de lokatie van vertrek en de lokatie van aankomst).

Dit is een simpel voorbeeld van rekenen met resten.
In de getaltheorie komt het rekenen met resten veelvuldig voor. Ook in de praktijk speelt rekenen met resten een belangrijke rol; met name bij het berekenen van grote priemgetallen die worden gebruikt bij cryptografie. En cryptografische toepassingen gebruiken we bijna dagelijks; meestal onbewust: bijvoorbeeld bij internetbankieren of bij pin-betalingen.

Het berekenen van resten bij delingen door grote getallen kost veel rekenkracht van een computer en daardoor veel tijd. Daarom zijn er allerlei methoden ontwikkeld om modulair rekenen met grote getallen te versnellen. Eén zo'n methode is ontwikkeld door Peter Montgomery in 1985 en wordt Montgomery Reduction en Montgomery Multiplication genoemd[1].
Daarnaast zijn er andere technieken om het rekenen met resten in bepaalde situaties te versnellen. Op internet is veel informatie te vinden over modulair rekenen, maar deze informatie is niet altijd even toegankelijk; vaak bevat de informatie ingewikkelde formules of is de informatie niet compleet. In dit document proberen we een zo eenvoudig mogelijke uitleg te geven van de Montgomery vermenigvuldiging en van andere methoden die gebruikt worden voor modulair rekenen met grote getallen.

Repetitie

Er valt bijna niet aan te ontkomen om toch enige wiskundige begrippen en eenvoudige formules te gebruiken in dit document. Daarom hier een kleine herhaling van wiskundige begrippen, symbolen en formules. Voorts een kleine inleiding tot de verwerking en de opslag van getallen in een binaire computer:

Modulus

De rest bepalen van een getal *a* dat gedeeld wordt door een getal *m* wordt wiskundig weergegeven als volgt:
a (mod m). Getal *m* heet de modulus. Het resultaat wordt voorafgegaan door het symbool ≡ , dit in tegenstelling tot het is gelijk (=) teken bij het resultaat van een som zonder modulus.
Dus de rest *r* van de deling is: *a (mod m)* ≡ *r* . Bij modulair rekenen gaat het om natuurlijke gehele getallen, dus geen decimalen of breuken. Alle getallen die een rest kunnen vormen bij een bepaalde modulus *m* noemen we het domein modulus *m.* Alle gehele getallen van nul tot en met *m-1* behoren tot dat domein. Indien een getal buiten dit domein valt kunnen we het getal binnen het domein brengen door een of meerdere keren de modulus er van af te trekken. Indien een getal negatief is kunnen we het getal binnen het domein brengen door de modulus een of meer keren er bij op te tellen. Negatieve getallen behoren in principe niet tot het domein modulus *m*. Er zijn wel uitzonderingen waarbij negatieve getallen worden gebruikt bij modulair rekenen, zoals bij de NTT (Number-theoretic Transform)[2].

Machten

Als *a* een natuurlijk getal is groter dan 1 dan geldt:
$a^{0} = 1$
$a^{1} = a$
$a^{3} \times a^{2} = a^{(3+2)} = a^{5}$
$a^{3} : a^{2} = a^{(3-2)} = a^{1}$
De inverse (het omgekeerde) van $a^{1} = a^{-1}$; Dit is dus één gedeeld door *a*.
Dus $a^{1} \times a^{-1} = a^{(1-1)} = a^{0} = 1$

De binaire computer

In de binaire wereld van de computer worden getallen meestal in de vorm van een rij bits opgeslagen. In de moderne computer is dat een rij (register) van 32 of van 64 bits; Dit wordt dan een integer(-register) genoemd en vormt de eenheid waarmee een computer rekent. Elke bit kan een waarde 0 of 1 bevatten en stelt een macht van twee voor: de bitnummering voor een 32-bits integer is van 0 tot/met 31.
De eerste bit (bit 0) staat voor de waarde 2^0 wanneer die bit op 1 staat; anders is de waarde nul. De volgende bit staat voor de waarde 2^1 wanneer de bit op 1 staat; anders is de waarde nul. Dan volgen 2^2, 2^3, 2^4...enz. tot 2^{31} voor bit 31 indien de bit op 1 staat; anders is de waarde nul. De totale waarde van een integer register is de som van de waarden van alle bits.
Visueel wordt een 32-bits integer-register aanschouwelijk gemaakt door een rij van 32 vakjes waarin het vakje 0 geheel rechts staat en het vakje 31 helemaal links. Kunstmatig kunnen we een grotere integer maken door meerdere 32- of 64-bits integers aan elkaar te koppelen.
We kunnen de inhoud van een integer vermenigvuldigen met twee door de inhoud van de bits elk een plaats naar links op te schuiven. Wat eerst op bit nummer 30 stond, gaat nu naar bit nummer 31; wat op bit 29 stond gaat naar bit nummer 30 … enz. Bit nummer 0 krijgt een 0 als inhoud. De inhoud van wat in bit nummer 31 stond, verdwijnt (of komt in een soort overflow bit, ook wel carry-flag genoemd).
We kunnen de inhoud van een integer delen door twee, door de inhoud van de bits elk een plaats naar rechts op te schuiven. Wat eerst op bit nummer 2 stond, gaat nu naar bit nummer 1; wat op 3 stond gaat naar 2...enz. Wat op bit 31 stond gaat naar bit 30, en op bit 31 wordt een nul gezet (bij een 32-bit integer). Wat er op bit 0 stond verdwijnt. Er zijn (in de meeste computers) computer-instructies voor het schuiven van de inhoud van een integer-register: instructie SHR staat voor Shift Right, en instructie SHL staat voor Shift Left.

Bijzondere moduli

Er is een aantal vormen van moduli dat modulaire berekeningen aanzienlijk vereenvoudigt. Vooral in een computer(-programma) kan de modulaire berekening eenvoudiger worden[3]. Maar zo'n modulaire berekening kan ook in computer-hardware worden uitgevoerd. Hier volgen drie van zulke moduli:

mod (2^n)

Bij *mod (2^n)* moeten we de rest bepalen van een getal dat we delen door een macht *n* van twee. Indien dit getal is opgeslagen in een binaire computer in de vorm van een of meerdere geschakelde integers, dan kunnen we de inhoud van deze integer(s) delen door een macht *n* van twee *(2^n)* door de de inhoud van de bits *n* plaatsen naar rechts op te schuiven. In dat geval verdwijnt de inhoud van *n* bits aan de rechterkant. We kunnen de bits die aan de rechterkant uitschuiven ook opvangen in een aparte integer. We kunnen dan, in die aparte integer, de bits op de juiste plaats zetten, zodat de bit die het eerst is uitgeschoven, op de plaats van bit 0 komt, enz. Dan vormt de inhoud van die aparte integer de rest van de deling door (een macht van) twee. Zo kunnen we vrij eenvoudig de waarde van een getal mod *(2^n)* bepalen.

mod ($2^n + 1$)

Op meerdere plaatsen in de getal theorie komt het rekenen *mod (2^n+1)* voor. De modulus *(2^n+1)* wordt ook wel “Fermat modulus” genoemd, omdat een Fermat-getal dezelfde vorm heeft: *2^n+1*. Het verschil is dat bij een Fermat-getal de exponent *n* zelf een macht van twee moet zijn, en bij een Fermat modulus is dat geen vereiste.
De Fermat modulus komt voor bij het “Strassen Schönhage Algorithm” voor het vermenigvuldigingen van grote getallen[4].

Bij het berekenen van getallen modulo een modulus in de vorm *(2^n+1)* kan de volgende truc worden gebruikt: Deel het getal eerst door *2^n*. We krijgen dan een quotiënt *q* en een rest *r*. We hadden door ***(2^n+1)*** moeten delen; we moeten de rest dus nog verminderen met *q*. Indien *r* dan negatief wordt, dan moeten we de rest *r* weer met *(2^n+1)* verhogen.
Zoals we gezien hebben gaat in een binaire computer het delen van een binair

getal door (een macht van) 2 door middel van een shift naar rechts over 1 of meerdere bits. De bits die overblijven vormen het quotiënt en de *n* bits die uit worden geschoven (de *n* bits van de modulus) vormen de rest, nadat van deze rest het quotiënt is afgetrokken[5].

mod ($2^n - 1$)

Deze modulus wordt Mersenne modulus genoemd omdat de modulus dezelfde vorm heeft als een Mersenne getal. Deze vorm van modulus wordt ondermeer gebruikt in de Lucas Lehmer test voor Mersenne priemgetallen[6].
Bij het berekenen van getallen modulo een modulus in de vorm $(2^n - 1)$ kan de volgende truc worden gebruikt: Deel het getal eerst door 2^n. We krijgen dan een quotient *q* en een rest *r*. We hadden door $(2^n - 1)$ moeten delen; we hebben dus door een te groot getal gedeeld. Daardoor is de rest *r* te klein geworden, dat corrigeren we door *r* te verhogen met quotient *q*. Wanneer dan *r* groter wordt dan de modulus, dan moeten we *r* nog eenmaal verminderen met de modulus.
Het delen door 2^n gaat door een shift naar rechts over *n* bits. De bits die overblijven vormen het quotient *q* en de *n* bits die uit worden geschoven (de *n* bits van de modulus) vormen de rest *r*. We moeten nu *q* nog optellen bij de rest *r* en controleren of het resultaat dan niet groter is dan de modulus. Indien het resultaat groter is dan verminderen we het resultaat eenmaal met de modulus.

Montgomery vermenigvuldigen

Het principe van de Montgomery vermenigvuldiging is om gelijktijdig met een vermenigvuldiging de modulus bewerking uit te voeren. Dus de vermenigvuldiging van getal *a* met getal *b,* en de modulus bewerking *(mod N)* gaat in één bewerking: *a* x *b (mod N)*. Het belangrijkste verschil met een "normale" mod bewerking is dat bij de Montgomery bewerking geen deel-bewerking wordt uitgevoerd; slechts vermenigvuldigingen, optellingen en shift operaties zijn nodig. Delingen zijn erg duur in termen van computer-kracht (ze kosten veel tijd)[xx]. Door in plaats van delingen, vermenigvuldigingen en shift bewerkingen te gebruiken, kan een Montgomery vermenigvuldiging veel sneller zijn dan een gewone vermenigvuldiging gevolgd door een "normale" mod bewerking.

Hiertoe worden de factoren *a* en *b* eerst in een "*Montgomery vorm*" gebracht. Deze vorm houdt in: het getal vermenigvuldigen met een factor "*R*" en toepassen van *(mod N)*. Later meer over deze factor "*R*" en de *(mod N)* bewerkingen.
We krijgen dus *a* x *R (mod N)* en *b* x *R (mod N)*.
Vermenigvuldigen van *a* met *b* wordt dan: *a* x *R* x *b* x *R (mod N)* = *ab* x R^2 *(mod N)*. Door nu dit resultaat te delen door *R* krijgen we *ab* x *R (mod N)*. Het resultaat is opnieuw een getal in de Montgomery vorm. Op deze wijze kan worden dóórvermenigvuldigd met andere factoren die in de Montgomery vorm zijn.

Het delen door *R* wordt mogelijk door de modulus bij het tussen-resultaat op te tellen, net zo vaak als nodig is om dit tussenresultaat deelbaar te maken door *R* (door de modulus op te tellen verandert het modulair resultaat niet). De factor *R* wordt zo gekozen dat deling door *R* eenvoudig kan geschieden (in een decimale omgeving kan voor *R* een 10 tot een bepaalde macht worden genomen; in een binaire omgeving wordt een macht van 2 genomen). Dat delen door R kan ook in stappen gebeuren; dus Bij R = 10.000 kan je ook viermaal delen door 10. We zien straks hoe dat gaat.

Voorbeeld:
Modulus *N = 843*
R = 1.000
T = tussenresultaat; *T* is gelimiteerd tot de modulus, maar tijdens berekeningen kan het resultaat tijdelijk groter worden dan de modulus.
Omdat we het tussenresultaat *T* deelbaar willen maken door *R* (1.000) laten we in dit voorbeeld alle duizendtallen van *T* even weg; deze zijn zowiezo deelbaar door duizend.
We zoeken nu een factor *m* zodanig dat *m* maal *N,* opgeteld bij *T* een tussen-resultaat oplevert van *T = ...000*, dus deelbaar door *R*. We kunnen de factor *m* bepalen door de laatste drie cijfers van *T* te vermenigvuldidgen met een factor *x*. (In een formule uitgedrukt: *m = T(mod R) * x*).
De factor *x* is in dit geval 293.
Bijvoorbeeld:
Als *T = ...001* dan is *m = 001*x = 293* en *m*N = 246.999.* Dit opgeteld bij *T* levert *...000 op.*
*Als T = ...123 dan is m = 123*x = 123*293 = 36.039 en m*N = 30.380.877.*
Dit opgeteld bij *T* levert *...000* op.
Voor elke waarde van *T (mod R)* kunnen we een waarde voor factor *m* vinden, zodanig dat *m*N* opgeteld bij *T* een duizendvoud oplevert.

Hoe komen we aan de factor *x* ?
Factor *x* berekenen we met het *uitgebreid algoritme van Euclides*[8]. Het ligt buiten het bestek van dit document om hier dieper op in te gaan.
Als input voor dit algoritme gebruiken we *R* en *N*. In ons voorbeeld is dat dus *R = 1.000* en *N = 843*. Het resultaat is *x* = 293. Er kan altijd een *x* gevonden worden mits *GGD(R,N) = 1*. Meestal is aan deze voorwaarde gemakkelijk te voldoen indien *R* een macht van twee is en *N* een oneven getal. Indien *N* een macht van 10 is (zoals in dit voorbeeld) wordt aan de voorwaarde voldaan indien *N* oneven is en niet eindigt op een 5.
Na de Montgomery vermenigvuldiging(en) is het resultaat nog in de Montgomery-vorm: *T x R (mod N)*. Dit moet weer terug gebracht worden in gewone vorm door *T* te delen door *R (mod N)*. Dat kan het beste door het laatste tussenresultaat *T* via een montgomery-vermenigvuldiging te vermenigvuldigen met een gewone 1.

Voorbeeld

We willen berekenen: a x b *(mod N)* ≡ c volgens de Montgomery methode.
Getal a = 1.234; getal b = 5.678; de modulus N = 6.543.
We gaan uit van een "base" van 10 om de berekening aanschouwelijk te maken.
Eerst brengen we beide getallen in de Montgomery vorm. Omdat de getallen uit vier decimale cijfers bestaan gebruiken we een "R" van 10^4 ; dus we vermenigvuldigen beide getallen met 10^4 en delen door de modulus waarbij de rest het resultaat is als volgt:

a' = 1.234 x 10.000 (mod 6.543) ≡ 6.445
b' = 5.678 x 10.000 (mod 6.543) ≡ 6.389

We vermenigvuldigen nu 6.445 x 6.389 volgens de klassieke methode en voegen na elke regel een "Montgomery" regel toe waarmee we het resultaat tot dan toe op een 10-voud brengen. We hadden ook kunnen kiezen voor het toevoegen van éénmaal een "Montgomery"regel om daarmee het resultaat direct op een 10.000- voud te brengen. Dat doen we niet omdat dat complexer is en niet tot een hogere efficiency leidt, zoals we straks zullen zien.

Eerst moeten we nog factor x uitrekenen via het *uitgebreide algoritme van Euclides.* Als input voor deze berekening nemen we $R = 10$ en voor N nemen we het laatste cijfer van de modulus; dus $N = 3$. De factor x wordt dan *3*. Het laatste cijfer van elk tussenresultaat T wordt dan vermenigvuldigd met x *(mod 10)* om de factor m te krijgen. De modulus wordt dan vermenigvuldigd met m en opgeteld bij het tussenresultaat T.

```
          multiplicand  a':      6389    factor x = 3
          multiplier    b':      6445 x          ↓
          modulus:               ------
5 x 6389                          31945    m =(5 x 3 mod 10) = 5
          6543 x 5                32715
                                   6466
4 x 6389                          25556
                                  32022    m =(2 x 3 mod 10) = 6
          6543 x 6                39258
                                   7128
4 x 6389                          25556
                                  32684    m =(4 x 3 mod 10) = 2
          6543 x 2                13086
                                   4577
6 x 6389                          38334
                                  42911    m =(1 x 3 mod 10) = 3
          6543 x 3                19629
                        c':       6254
```

Het resultaat c' = 6.254 is nog steeds in de Montgomery vorm. Met dit getal kunnen we (als dat nodig zou zijn), doorgaan met de Montgomery vermenigvuldigingen. Dat zullen we nu niet doen. Om het resultaat terug te brengen in de 'gewone' vorm, moeten we het getal 6.254 vermenigvuldigen met een gewone 1 volgens de Montgomery manier:

```
          multiplicand                1    factor x = 3
          multiplier      c':      6254 x          ↓
          modulus:                ------
4 x    1                                4    m =(4 x 3 mod 10) = 2
          6543 x 2                  13086
                                     1309
5 x    1                              5
                                     1314    m =(4 x 3 mod 10) = 2
          6543 x 2                  13086
                                     1440
2 x    1                              2
                                     1442    m =(2 x 3 mod 10) = 6
          6543 x 6                  39258
                                     4070
6 x    1                              6
                                     4076    m =(6 x 3 mod 10) = 8
          6543 x 8                  52344
                          c:        5642
```

Het resultaat c = 5.642 en dat is precies de uitkmomst van *a* x *b (mod N)* = 1.234 x 5.678 (mod 6.543) ≡ 5.642

We zien dat we wel heel erg veel extra werk moeten uitvoeren voor een betrekkelijk simpele modulaire vermenigvuldiging. Later zullen we zien dat er situaties zijn waarin deze methode juist veel werk bespaart.
In bovenstaand voorbeeld hebben we de deling door 10.000 in vier stappen van 10 uitgevoerd. We hadden dit ook in één stap van 10.000 kunnen doen. Het aantal kern-vermenigvuldigingen[9] blijft dan echter gelijk. Het berekenen van de factor *m* met het *uitgebreide algoritme van Euclides* is echter voor getallen *R=10.000* en *N = 6.543* veel complexer dan voor getallen *R=10* en *N=9*. Daarentegen is in het laatste geval het aantal optellingen weer groter. Al met al is het eenvoudiger om bij de Montgomery berekening de factoren *N* en *R* klein te houden; in een decimale omgeving (zoals in ons voorbeeld) dus *R* op *10* stellen en voor *N* alleen het laatste cijfer van de modulus.
In een binaire omgeving (in een computer programma) wordt *R* dan op de waarde 2^{32} of 2^{64} gesteld. De factor *N* wordt dan de waarde van de laatste 32 of 64 bit van de modulus.

Modulair machtsverheffen

In de getaltheorie komen vaak formules voor in de vorm $a^p \ (mod\ N)$. Het basis getal *a* is meestal klein maar de exponent *p* kan heel groot zijn (honderden decimale cijfers, of in binaire vorm duizenden bits). Het is duidelijk dat *a* x *a* x *a* x … en dat 10^p maal (indien *p* is uitgedrukt in decimale cijfers) of 2^p (indien *p* is uitgedrukt in bits) een onmogelijke opgave is. Dat zou veel te lang duren en zou een veel te groot getal opleveren, zelfs als we voor *a* het kleinst mogelijke getal > 1 nemen: *a = 2*. Dit is zo als we met de *(mod N)* bewerking zouden wachten tot de machtsverheffing helemaal klaar is.
We kunnen ook de *(mod N)* bewerking na elke vermenigvuldiging met *a* uitvoeren. Het resultaat blijft dan gelijk aan het resultaat dat we zouden krijgen als we de *(mod N)* bewerking helemaal op het einde zouden doen. We krijgen dan ((*a* x *a (mod N)* x *a*) *(mod N)* …. enz. De grootte van het getal blijft dan beperkt tot maximaal *N*.
Het onmogelijk grote aantal (2^p) vermenigvuldigingen en *(mod N)* bewerkingen blijft een wezenlijke beperking, tenzij we dit aantal kunnen reduceren. Dat kan als volgt: We kunnen de exponent *p* opdelen in stukjes die elk gemakkelijk te berekenen zijn. Op die stukjes kunnen we de mod bewerking toepassen.Daarna kunnen we die stukjes met elkaar vermenigvuldigen om zo het eindresultaat te verkrijgen; voorwaarde is wel dat het aantal stukjes veel kleiner is dan *p*.
We kunnen de exponent *p* noteren als een binair getal; elke bit vertegenwoordigt dan een getal dat een macht van twee is. De bits stellen de waarden 2^i voor, voor i = 0, 1, 2 .. voor resp. bit 0, 1, 2 ...

Binaire ladders [10]

Het algoritme dat machtsverheffen met een exponent uitgedrukt als binair getal beschrijft heeft de naam “binaire ladder” gekregen (meer gebruikelijk is de Engelse term “binary ladder exponentiation”).
Als voorbeeld nemen we de formule $a^y \ (mod\ N)$, waarbij we de exponent y de waarde 37 geven en y uitdrukken als binair getal.

Exponent y uitgedrukt als binair getal:

y = 37 = 32 + 4 + 1 = 1 0 1 0 0 1

Bitnummer:	5	4	3	2	1	0
	2^5	2^4	2^3	2^2	2^1	2^0
Waarde per bit:	32	16	8	4	2	1
Binair getal	1	0	0	1	0	1
Waarde binair getal	32 +	0 +	0 +	4 +	0 +	1 = 37
Als exponent van a:	a^{32}			a^4		$a^1 = a^{37}$

Waar in het binaire getal een 0 staat is de waarde $a^0 = 1$.
De totale waarde van a^y kan nu worden verkregen door de termen a^1, a^4 en a^{32} met elkaar te vermenigvuldigen.
Door *a* telkens te kwadrateren kunnen we nu de waarden vinden voor a^1, a^2, a^4, a^8, a^{16}, a^{32}. Door in een apart getal de waarden door te vermenigvuldigen bij de bits die op één staan vinden we a^{37}.

De hierboven beschreven methode staat bekend als *RightToLeftBinaryExponentiation.* Er is ook een methode die enige voordelen biedt boven de eerstgenoemde methode en die de naam heeft van *LeftToRightBinaryExponentiation*[13].

Left To Right Binary Exponentiation

Deze methode begint bij de meest linker bit die op 1 staat (dus de bit met de grootste waarde) en gaat naar rechts tot en met de bit met de waarde a^1.
Bij het voorbeeld met exponent y = 37 beginnen we met de meest linker bit, waarbij het tussen-resultaat T de waarde a^1 krijgt. We vervolgen naar rechts met een bit die op 0 staat en kwadrateren de waarde in T; deze wordt nu a^2.
Bij de volgende bit kwadrateren we weer; $T = a^4$. De volgende bit naar rechts

heeft de waarde 1. We kwadrateren nu niet alleen ($T=a^8$), maar vermenigvuldigen T nog een keer extra met a; dus $T = a^9$. Dan komt weer een nul-bit dus T wordt a^{18}. De laatste (meest rechter) bit staat op 1 dus we kwadrateren T ($T = a^{36}$) en vermenigvuldigen nog een keer met a ($T=a^{37}$).
Als de exponent *y* een grote waarde heeft ontstaat er een onmogelijk groot getal. Maar door de modulaire waarde van elk tussen-resultaat te bepalen blijft de grootte van de tussen-resultaten beperkt tot de modulus.

We kunnen nu de methode van de binary ladder combineren met de Montgomery vermenigvuldiging en krijgen zo de Montgomery Exponentiation. Elke kwadratering van het tussen-resultaat gebeurt nu met de "Montgomery square multiplication". Een mooie toepassing van de Montgomery Exponentiation is bij de berekening van de kleine stelling van Fermat.

De kleine stelling van Fermat

Pierre de Fermat leefde begin 17e eeuw in Frankrijk. Hij heeft veel onderzoek gedaan op het gebied van wiskunde en getaltheorie. Hij is vooral bekend vanwege de "laatste stelling van Fermat" maar zijn "kleine stelling van Fermat" is in de praktijk veel belangrijker gebleken. Deze stelling ligt aan de basis van veel cryptografische toepassingen. Deze stelling luidt:

Indien p een priemgetal is, en a is een positief getal > 1, en verder hebben a en p geen gemeenschappelijke deler, dan is

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

We noemen deze stelling in dit document verder de Fermat test of de kleine Fermat test. Indien het resultaat ongelijk 1 (mod p) is, dan is p samengesteld. We kunnen met deze stelling een schifting maken tussen getallen die samengesteld zijn en getallen die **mogelijk** priem zijn. Bij cryptografische toepassingen zijn grote priemgetallen nodig. Die kan men verkrijgen door random getallen te genereren en deze getallen op priem te testen. Daar kan men de kleine Fermat test voor gebruiken. Als een getal door de kleine Fermat test komt is de kans groot dat het een priemgetal is en met aanvullende testen kan een bijna zekerheid worden verkregen dat het getal een priemgetal is.

De kleine Fermat test kan prima worden uitgevoerd met de Montgomery Exponentiation. Dit zou als volgt kunnen worden gedaan:
(in de volgende beschrijving gaan we uit van een waarde van 2 voor a. Indien voor a een andere waarde wordt gekozen, dan gelden sommige van de hier beschreven versnellingen niet).

We willen een getal p testen op priem zijn. We kiezen $a = 2$. Voor R kiezen we een macht van twee, en wel een veelvoud van 32 (we gaan uit van een 32-bits processor). Eerst bepalen we de grootte van p in aantal bits. Dit aantal ronden we op tot een 32-voud en dat noemen we q; dat vormt dan de exponent van 2 voor R. Dus $R = 2^q$

Voor de Montgomery vermenigvuldiging gaan we *a* in de Montgomery vorm brengen: dus $a' = R$ x $a \pmod p = 2^q$ x $2 \pmod p$. We nemen het aantal bits waaruit *p* bestaat (dat hebben we eerder al bepaald) maar **zonder** de opronding tot en 32-voud. Indien dat aantal géén veelvoud van 32 is moeten we modulus *p* berekenen door het getal 2^q x *2* te delen door *p* om zo de rest te verkrijgen. Dit is een computer intensieve berekening.
Indien het aantal bits van *p* wél een veelvoud van 32 is kan de berekening veel eenvoudiger, en gaat dan als volgt: $a' = 2$ x $(R - p)$. Indien a' dan groter dan *p* is dan wordt a' nog verminderd met *p*.

(Bij het berekenen van priemgetallen voor cryptografische toepassingen komt het vaak voor dat het aantal bits van het te berekenen priemgetal een veelvoud van 32 is).

Dan gaan we de exponent *p* voor *a* uitdrukken in binaire vorm (eerst *p* met 1 verminderen). Dan voeren we de Montgomery Exponentiation uit: voor elke bitpositie van $(p - 1)$ voeren we een Montgomery square multiplication uit, te beginnen met tussenresultaat $T = a'$ en beginnend bij de tweede bit van links (de bits met de hoogste waarden). Voor elke "0" bit voeren we de Montgomery square multiplication uit, en voor elke "1" bit doen we dit ook en bovendien vermenigvuldigen *T* met *a* ; dus met twee.
Dit vermenigvuldigen met twee doen we door het tussenresultaat één bit naar links op te schuiven. Daarna moeten we nog *(mod p)* doen; dit bereiken we door, voor zover het tussenresultaat $T > p$, éénmaal *p* van *T* af te trekken.
Zo werken we alle bits van $(p - 1)$ af, en houden we uiteindelijk een tussenresultaat over dat we alleen nog maar vanuit de Montgomery vorm terug moeten brengen naar de gewone vorm.
Dit doen we door *T* te delen door $R \pmod p$; ofwel te delen door $2^q \pmod p$.

Maar wacht even; we hebben eerst *a* (twee), $(p-1)$ maal met zichzelf vermenigvuldigd *(mod p)*, en nu moeten we het tussenresultaat weer door $2^q \pmod p$ delen. **Dat kan beter !**
We kunnen de exponent die bij het begin van de binary ladder $(p - 1)$ was, verminderen met *q* voordat de exponentiation begint. Die exponent wordt dan

(p – 1 – q). Daarmee wordt de laatste stap, om van de Montgomery vorm naar de gewone vorm te komen, overbodig gemaakt!
Als het eindresultaat ongelijk 1 is, dan is *p* zeker géén priemgetal maar is *p* samengesteld. Indien het eindresultaat nu gelijk is aan 1, dan is *p* waarschijnlijk een priemgetal. Met aanvullende testen kunnen we de waarschijnlijk dat *p* een priemgetal is verder opvoeren. We kunnen bijvoorbeeld de kleine stelling van Fermat nogmaals uitvoeren voor *p*, maar dan met een andere base *a*; bijvoorbeeld *a = 3.* Dezelfde berekening met de Montgomery squaring kan dan worden uitgevoerd, maar dan zonder de versnellingen, die bij *a = 2* horen, zoals het verminderen van de exponent met *q*.
Indien het eindresultaat van een Fermattest voor een *samengesteld getal* toch ≡ 1 is, noemen we zo'n getal een pseudo-priem getal voor base *a*. Indien een samengesteld getal bij de Fermattest ≡ 1 oplevert bij elke waarde voor base a, dan wordt zo'n getal een *Carmichael* getal genoemd.

In de praktijk wordt de Fermat test bij slagen gevolgd door de Miller-Rabin test[13]. Deze test lijkt op de Fermattest en maakt ook gebruik van de "Montgomery exponentiation". Verder gebruikt deze test ook een base *a*. Als de test tien maal succesvol wordt uitgevoerd voor verschillende waarden van *a*, dan is de kans dat het geteste nummer een priemgetal is groter dan 99,9999 %.

References

1. *Montgomery* (http://en.wikipedia.org/wiki/Peter_Montgomery_(mathematician))
2. Theo Kortekaas. *Vermenigvuldigen van grote getallen en het Schönhage-Strassen Algoritme* (http://tonjanee.home.xs4all.nl/SSAbeschrijving.pdf)
3. Richard Crandall en Carl Pomerance. *Prime Numbers – A Computational Perspective,* Second Edition, Springer, 2005. Sectie 9.2.3: Moduli of special form, Blz. 454-457.
4. Pierrick Gaudry, Alexander Kruppa, Paul Zimmermann. *A GMP-based Implementation of Schönhage-Strassen's Large Integer Multiplication Algorithm* (http://www.loria.fr/~gaudry/publis/issac07.pdf)
5. *Shift optimization* (https://en.wikipedia.org/wiki/Schönhage-Strassen_algorithm) Section 1.4
6. *Lucas-Lehmer primality test* (https://en.wikipedia.org/Lucas-Lehmer_primality_test)
7. Richard Brent, Paul Zimmermann. *Modern Computer Arithmetic* Cambridge University Press, Section 2.1.2: Montgomery's form, page 48.
8. *Uitgebreid algoritme van Euclides* (https://nl.wikipedia.org/wiki/Uitgebreid_algoritme_van_Euclides)
9. Theo Kortekaas. *Vermenigvuldigen van grote getallen en het Schönhage-Strassen Algoritme* (http://tonjanee.home.xs4all.nl/SSAbeschrijving.pdf) Kern-vermenigvuldiging, blz. 3.
10. Richard Crandall en Carl Pomerance. *Prime Numbers – A Computational Perspective,* Second Edition, Springer, 2005. Section 9.3.1: Basic binary ladders blz. 458-460.
11. Richard Brent, Paul Zimmermann. *Modern Computer Arithmetic* Cambridge University Press, Section 2.6.1: Binary exponentiation, LeftToRightBinaryExponentiation, blz. 70.
12. Kleine stelling van Fermat (https://nl.wikipedia.org/wiki/Kleine_stelling_van_Fermat)
13. Miller-Rabin test (https://nl.wikipedia.org/wiki/Miller-Rabin-priemgetaltest)

Auteur: T.P.J. Kortekaas
Email: t.kortekaas@xs4all.nl
Web-pagina's: www.tonjanee.home.xs4all.nl