Vermenigvuldigen van grote getallen en het Schönhage-Strassen Algoritme

door Theo Kortekaas

Inleiding

Vermenigvuldigen doen we allang niet meer uit ons hoofd. Vaak wordt het voor ons gedaan. Denk aan de kassa in de supermarkt, als we zeven flesjes bier van 79 cent per stuk afrekenen. Als we nog eens zelf iets moeten vermenigvuldigen of uitrekenen dan gebruiken we een calculator. Voor iets uitgebreidere berekeningen kunnen we een spreadsheet op een computer gebruikten.
Met computerprogramma's zoals Excel of OpenOffice Calc kunnen we getallen met elkaar vermenigvuldigen van ongeveer acht cijfers. Het resultaat is dan een getal van ongeveer 15 cijfers en dat getal is exact juist. Dat is voor de meeste toepassingen voldoende. Bij grotere getallen wordt er een afronding toegepast.

Er zijn echter gebieden waar met veel grotere getallen moet kunnen worden gerekend en waarbij de uitkomst exact juist moet zijn. Zoals bij cryptografische toepassingen, waar grote priemgetallen worden gebruikt; getallen van soms wel honderden decimale cijfers. Ook op het terrein van de wiskunde en de getaltheorie worden grote getallen gebruikt; bij het berekenen van zoveel mogelijk decimalen van π (pi) of van *e* (het grondtal van natuurlijke logaritme). Ook is er een wedloop naar het vinden van al maar grotere priemgetallen. Het grootste priemgetal op dit moment (voorjaar 2015) heeft meer dan zeventien miljoen decimale cijfers (Mersenne priemgetal).
Bij al deze toepassingen moet er worden gerekend en vermenigvuldigd met zeer grote getallen. Het is duidelijk dat we dit alleen maar met een computerprogramma kunnen doen.

Complexiteit

Hoe groter de getallen, des te meer werk het is om deze getallen te vermenigvuldigen. Maar hoeveel meer? Daar hebben de geleerden een maat voor vastgesteld; de Big-O-notatie of Grote-O-notatie.[6] Deze notatie, die complexiteit wordt genoemd, geeft de verhouding weer van de hoeveelheid werk om een bepaalde formule of een bepaald algoritme uit te voeren ten opzichte van de input voor deze formule. De input wordt meestal weergegeven als n. Bij de formules en algoritmes die gebruikt worden bij grote vermenigvuldigingen wordt n meestal uitgedrukt in aantal decimale cijfers of ook wel aantal bits, waaruit de twee getallen bestaan die moeten worden vermenigvuldigd. Een stilzwijgende aanname is dat deze getallen ongeveer even groot zijn.
Zo is de complexiteit van een optelling van twee getallen $O(n)$. Dit betekent dat als de getallen van een optelling twee keer zo groot worden (dus n wordt twee keer zo groot), dat dan ook de hoeveel werk om de optelling uit te voeren twee keer zo groot wordt.
Vermenigvuldigen zoals we dat op school geleerd hebben (dat noemen we hier de klassieke methode) heeft een complexiteit van $O(n^2)$. Dat betekent dat als de te vermenigvuldigen getallen twee maal zo groot worden, de hoeveelheid werk viermaal zo groot wordt.

Vermenigvuldigen met een computerprogramma

Voor het vermenigvuldigen van getallen tot ongeveer acht decimale cijfers zijn voor de meeste computers standaard programma's beschikbaar zoals OpenOffice Calc of Excel. Voor het vermenigvuldigen van grotere getallen zijn speciale programma's nodig of speciale subroutines die kunnen omgaan met grote getallen. Deze subroutines werkten vroeger bij het vermenigvuldigen veelal volgens de klassieke methode.
Als echter de getallen heel groot worden, dan duurt een vermenigvuldiging volgens de klassieke methode, ook op snelle computers, toch nog onaanvaardbaar lang. Daarom heeft de wetenschap (wiskundigen en computer-specialisten) gezocht naar andere methoden om de berekening te versnellen. Dit heeft geleid tot speciale algoritmen die spectaculaire versnellingen kunnen opleveren. Daarover gaat dit document.

De algoritmen die worden besproken zijn: het Karatsuba algoritme; het Toom-Cook algoritme en het Schönhage-Strassen algoritme (SSA). Van dit laatste wordt een implementatie besproken voor een 32-bits Windows systeem. De verschillende algoritmen worden met elkaar vergeleken.

Een van de nieuwere ontwikkelingen is het Fürer algoritme. Dit is nog sneller dan SSA, maar zou die snelheid alleen kunnen bereiken met astronomisch grote getallen. In de praktijk wordt dit algoritme niet gebruikt. Daarom wordt het hier niet meegenomen.

Klassieke methode

Laten we eens kijken naar de methode die we op school geleerd hebben en die we in het vervolg de klassieke methode zullen noemen. Stel we willen twee getallen met elk vier decimale cijfers (1234 en 5678) met elkaar vermenigvuldigen. Gewoonlijk zouden we in zo'n geval bij de eerste vermenigvuldiging 8 x 4 = 32 twee opschrijven en drie onthouden, enz.
Nu gaan we dat iets anders doen; we schrijven 32 in z'n geheel op (dat noemen we een kern-vermenigvuldiging) en zorgen voor wat meer ruimte tussen de kolommen. Dat gaat dan zo:

```
              1  2  3  4
              5  6  7  8  x
              ----------
              8 16 24 32
           7 14 21 28
        6 12 18 24
     5 10 15 20
    --------------------
     5 16 34 60 61 52 32
```

De kolommen vertegenwoordigen elk een hoeveelheid maal tien tot een bepaalde macht; als volgt (bedenk dat $10^0 = 1$) :

$$
\begin{array}{rcll}
32 \times & 10^0 & = & 32 \\
52 \times & 10^1 & = & 520 \\
61 \times & 10^2 & = & 6.100 \\
60 \times & 10^3 & = & 60.000 \\
34 \times & 10^4 & = & 340.000 \\
16 \times & 10^5 & = & 1.600.000 \\
5 \times & 10^6 & = & 5.000.000 \\
\hline
1234 \times & 5678 & = & 7.006.652
\end{array}
$$

We zien dat er in dit voorbeeld zestien kern-vermenigvuldigingen zijn uitgevoerd. Indien we in plaats van twee getallen met elk vier decimale cijfers, twee getallen met elk zes decimale cijfers zouden hebben genomen, dan was het aantal kern-vermenigvuldigingen 36 geweest. Bij twee getallen van elk acht decimale cijfers zou het aantal kern-vermenigvuldigingen zelfs 64 bedragen. De hoeveelheid werk om twee decimale getallen met elkaar te vermenigvuldigen (uitgedrukt in aantal kern-vermenigvuldigingen) is dus, bij de klassieke methode, gelijk aan het kwadraat van de hoeveelheid decimale cijfers, waaruit die getallen bestaan. Kortom de complexiteit is $O(n^2)$. Hierbij gaan we er van uit dat de getallen ongeveer van gelijke grootte zijn.

Kern-vermenigvuldiging

We gebruiken deze term om een vermenigvuldiging op het laagste niveau aan te duiden.
In het vorenstaande voorbeeld is de vermenigvuldiging van 4 met 8 = 32 een kern-vermenigvuldiging omdat we deze vermenigvuldiging geleerd hebben met de tafels van vermenigvuldiging en we deze niet naar een nog eenvoudiger niveau kunnen brengen.
Wanneer we het hebben over een computerprogramma dan bedoelen we met een kern-

vermenigvuldiging: de vermenigvuldiging van twee integers van de woordlengte waarmee de computer werkt. Die woordlengte wordt uitgedrukt in bits en is voor de meest gangbare computers op dit moment 32 bits. De vermenigvuldiging van een 32-bits integer met een andere 32-bits integer levert een resultaat van maximaal 64 bits op en kan gewoonlijk in één computer-instructie worden uitgevoerd.

Verdeel en Heers

Een methode om ingewikkelde wiskundige en technische vraagstukken aan te pakken is: het probleem in steeds kleinere stukken opdelen totdat een niveau bereikt is waarop het deelprobleem overzichtelijk is en eenvoudig is op te lossen. Daarna kunnen de deeloplossingen weer worden samengevoegd (geïntegreerd) zodat de eindoplossing wordt gevonden. Een dergelijke methode wordt in de literatuur Divide en Conquer (D&C) genoemd, of in het Nederlands: Verdeel en Heers.[12] Deze techniek is ook toepasbaar op grote vermenigvuldigingen. Laten we ter illustratie het reeds uitgewerkte voorbeeld nemen uit de paragraaf over de klassieke methode: 1234 x 5678.
We gaan nu beide getallen in tweeën delen en er een naam aan geven als volgt:

```
1  2  = a                    3  4  = b
5  6  = c                    7  8  = d
```

Het eerste getal kunnen we nu schrijven als $(a*10^2+b)$ en het tweede getal als $(c*10^2+d)$. We kunnen nu wat algebra toepassen op de vermenigvuldiging: $(a*10^2+b) \times (c*10^2+d) = ac*10^4 + (ad + bc)*10^2 + bd$.
Het resultaat is dus vier kleinere vermenigvuldigingen en een aantal optellingen, als volgt:

```
a*c =   1  2     a*d =   1  2     b*c =   3  4     b*d =   3  4
        5  6 x           7  8 x           5  6 x           7  8 x
      -----            -----            -----            -----
        6 12             8 16            18 24            24 32
     5 10             7 14            15 20            21 28
    --------         --------         --------         --------
     5 16 12          7 22 16         15 38 24         21 52 32
```

Nu gaan we alles optellen; hierbij moeten we er aan denken dat a en c nog moeten worden vermenigvuldigd met 100. Dus het resultaat van ac moet met een factor 100 x 100 (10^4) worden vergroot en de resultaten van ad en van bc moeten met een factor 100 (10^2) worden vergroot.

```
                                                                    32 x 10^0 =    32
                                               24 x 10^0 =    24    52 x 10^1 =   520
                        16 x 10^0 =  16        38 x 10^1 =   380    21 x 10^2 = 2.100
12 x 10^0 =  12         22 x 10^1 = 220        15 x 10^2 = 1.500                ------
16 x 10^1 = 160          7 x 10^2 = 700                  ------                  2.652
 5 x 10^2 = 500                   ----            1.904 x 10^2 =               190.400
         ----                   936 x 10^2  =                                   93.600
          672 x 10^4 =                                                       6.720.000
                                                                            ----------
                                                           Totaal =        7.006.652
```

Het totaal is gelijk aan het resultaat van de klassieke methode, zoals de bedoeling is. Echter

ook het aantal kern-vermenigvuldigingen is 16 en dus gelijk aan het aantal van de klassieke methode, en het aantal optellingen is groter. Dus schieten we hier iets mee op? Later zullen we zien dat er methoden zijn om het aantal vermenigvuldigingen te beperken.

Recursiviteit

We kunnen ons een computerprogramma indenken dat als input twee getallen krijgt; deze getallen in tweeën (a en b; c en d) deelt en met deze delen vier vermenigvuldigingen uitvoert (ac, ad, bc en bd); het resultaat van deze vermenigvuldigingen bij elkaar optelt op de manier, zoals in het hier-vorenstaande voorbeeld is aangegeven, en het eindresultaat als output presenteert.
We kunnen beginnen met twee getallen van 8 cijfers elk. Deze getallen splitsen we in twee getallen van vier cijfers elk zodat we in totaal vier getallen van vier cijfers krijgen. Om het eindresultaat van 8 x 8 cijfers te krijgen moeten we vier vermenigvuldigingen uitvoeren van 4 x 4 cijfers.
Deze vier vermenigvuldigingen kunnen we ook weer door datzelfde computerprogramma *recursief* laten uitvoeren. Het gevolg is dat we 4x4=16 vermenigvuldigingen van twee bij twee cijfers krijgen. Nog een niveau dieper krijgen we 64 vermenigvuldigingen van één cijfer bij één cijfer. Dit is het laagste niveau; één cijfer kan niet verder gesplitst worden; op dit niveau passen we de tafels van vermenigvuldiging toe die we op school geleerd hebben en we hebben hieraan de naam van kernvermenigvuldiging gehangen.
Een vermenigvuldiging van 8 cijfers maal 8 cijfers leidt tot 8x8=64 kernvermenigvuldigingen. Dat klopt ook met de complexiteit van de klassieke vermenigvuldigingen zoals we eerder gezien hebben namelijk: $O(n^2)$.
Natuurlijk zullen we in een computerprogramma als laagste niveau niet de vermenigvuldiging van één decimaal cijfer bij één decimaal cijfer gebruiken, maar de eenheid van één computerwoord bij één computerwoord (dit is meestal in de tegenwoordige personal computers een woord van 32 bits of van 64 bits). Zo'n vermenigvuldiging wordt meestal in één computerinstructie uitgevoerd.
Voor kleinere getallen is de klassieke methode van vermenigvuldigen nog steeds de aanbevolen methode. De meer geavanceerde algoritmes geven juist bij grotere getallen een snelheidsvoordeel; later in dit document zullen we dat aantonen.

.

Het algoritme van Karatsuba

Tot 1960 dacht iedereen dat de hoeveelheid werk bij het vermenigvuldigen van twee getallen toenam met het kwadraat van het aantal cijfers van de te vermenigvuldigen getallen. Dus indien de getallen twee keer zo groot worden, dan kost dat vier maal zoveel werk. Dus de complexiteit is $O(n^2)$. Dit werd in 1960 ook beweerd door Andrey Kolmogorov, een Russische wiskundige, tijdens een seminar over wiskundige problemen aan de universiteit van Moskou.[10]
Anatolii Karatsuba, een toen 23-jarige Russische student die het seminar bezocht, vond echter binnen één week een sneller algoritme, waarmee hij de stelling van Kolmogorov onderuit haalde. Hij liet dit weten aan Kolmogorov en deze was “not amused”. Kolmogorov vertelde de volgende les van de vondst en brak het seminar af.
Om te illustreren wat het idee van Karatsuba was kunnen we het beste de berekening en de formule van de paragraaf “Verdeel en Heers” gebruiken. De formule luidde, nadat we het eerste getal hadden gesplitst in twee delen: *a* en *b*, en het tweede getal in *c* en *d:*

```
(a+b) x (c+d) = ac + ad + bc + bd = ac + (ad + bc) + bd.
Hierbij dient ac nog te worden vermenigvuldigd met 10^4 en
(ad+bc) met 10^2.
```

Karatsuba bedacht het volgende: wanneer je *a* en *b* bij elkaar optelt evenals *c* en *d* en je vermenigvuldigt de resultaten van de optellingen met elkaar dan krijg je de waarde (*ac*+*ad*+*bc*+*bd*) in één getal. Als je dan *a* met *c* vermenigvuldigt (*ac*) en *b* met *d* (*bd*) dan heb je drie getallen: (*ac*), (*bd*), en (*ac*+*ad*+*bc*+*bd*). Je kunt nu de waarde van *(ac+ad+bc+bd)* verminderen met de waarde van (*ac)* en verminderen met de waarde van (*bd)*.
Je houdt dan *(ad+bc)* over en je beschikt over alle ingrediënten om de som af te maken.
Voorbeeld: Neem de getallen uit het voorbeeld van de Klassieke Methode

```
        1  2  = a                    3  4  = b
        5  6  = c                    7  8  = d
```

We tellen a en b bij elkaar op en c en d:

```
          a + b =  12 + 34 =  46
          c + d =  56 + 78 = 134
```

We gaan nu drie vermenigvuldigingen uitvoeren;

```
a*c:      1  2     (a+b)*(c+d):   4  6     b*d:     3  4
          5  6 x               1  3  4 x            7  8 x
          ----                 -------            -----
          6 12                   16 24            24 32
       5 10                   12 18            21 28
                            4  6
       -------              -----------        --------
       5 16 12              4 18 34 24         21 52 32
Aftrekken (ac)                 5 16 12 -
Aftrekken (bd)              21 52 32 -
                            ---------
                            2  8  4  0
```

Net als bij normaal aftrekken kunnen we ook hier leentjebuur spelen, maar als we in een kolom meer dan 10 tekort komen, moeten we ook meer dan één lenen van de linker kolom. Nu gaan we alles optellen; hierbij moeten we er aan denken dat *a* en *c* nog moeten worden vermenigvuldigd met 100. Dus het resultaat van *ac* moet met een factor 100 x 100 (10^4) worden vergroot en de resultaten van *ad* en van *bc* moeten met een factor 100 (10^2) worden vergroot.

```
     ac:                  (a+b)*(c+d):                bd:

                        0 x 10^0 =     0       32 x 10^0 =     32
                        4 x 10^1 =    40       52 x 10^1 =    520
12 x 10^0 =    12       8 X 10^2 =   800       21 x 10^2 =  2.100
16 x 10^1 =   160       2 x 10^3 = 2.000                    -----
 5 x 10^2 =   500                  -----                    2.652
          -----                    2.840 x 10^2 =         284.000
             672 x 10^4 =                               6.720.000
                                                        ---------
Totaal:                                                 7.006.652
```

We hebben nu 12 kern-vermenigvuldigingen uitgevoerd en een aantal optellingen. De vermenigvuldiging van 1 x 46 bij (a+b)*(c+d) tellen we niet mee; bij de optelling van twee getallen van twee cijfers kan er een derde cijfer in het resultaat komen, maar dit kan nooit meer dan een 1 worden. Die 1 x 46 hoeft niet als een vermenigvuldiging te worden behandeld, maar kan behandeld worden als een optelling.
In plaats van de drie maal vier vermenigvuldigingen kunnen we drie maal het Karatsuba algoritme recursief uitvoeren. Dan komt het totaal aantal op 3 x 3 = 9 vermenigvuldigingen. In principe kunnen we het Karatsuba algoritme net zo vaak recursief uitvoeren tot we de basis vermenigvuldiging hebben bereikt (één decimaal cijfer maal één decimaal cijfer, of één computerwoord maal één computerwoord). Bij elke verdubbeling van het aantal te vermenigvuldigen cijfers zijn nu driemaal zoveel vermenigvuldigingen nodig en niet viermaal zoals bij de klassieke methode. Dit levert een complexiteit op van $O(n^{\log 3})$, of ongeveer $O(n^{1,585})$.[10] Daardoor is de methode van Karatsuba voor grotere getallen veel sneller dan de klassieke methode; echter omdat er nogal wat extra bewerkingen nodig zijn zal bij kleinere getallen de methode van Karatsuba minder snel kunnen zijn dan de klassieke methode. Dit geldt eveneens voor de andere (nog te behandelen) algoritmes. Wat de omslagpunten zijn zal in een volgend hoofdstuk worden behandeld.

Omdat de getallen bij Karatsuba telkens gehalveerd moeten kunnen worden is het wel nodig dat het aantal cijfers waaruit die getallen bestaan, een macht van twee is. Indien de getallen dat niet zijn, kunnen ze worden aangevuld met nullen tot de eerstvolgende macht van twee.

Toom-Cook Algoritme

Dit algoritme is ontwikkeld door Andrei Toom en verder verbeterd door Stephen Cook. Er zijn meerdere versies van dit algoritme; het meest gebruikte wordt Toom-3 genoemd, alhoewel deze naam ook wel gebruikt wordt als verzamelnaam voor alle op Toom-Cook gebaseerde algoritmen.[11] We zullen ons concentreren op de versie Toom-3.
Net zoals bij het algoritme van Karatsuba worden bij het algoritme van Toom-3 de te vermenigvuldigen getallen in delen gesplitst, echter niet in twee delen, maar in drie delen. Op deze delen wordt dan weer recursief het Toom-3 algoritme toegepast. Dit gaat zo door totdat de grootte van een kern-vermenigvuldiging is bereikt, of totdat het efficiënter is om op het laatst een ander algoritme (zoals Klassiek of Karatsuba) te gebruiken. Het Toom-3 algoritme is veel ingewikkelder dan het Karatsuba algoritme. We zullen het hier slechts summier behandelen.
In het kort komt het Toom-3 algoritme hier op neer:
Twee getallen x en y worden elk in drie gelijke delen (gelijk aantal bits) gesplitst. Op de drie delen van getal x wordt een aantal eenvoudige formules toegepast, waardoor er vijf nieuwe getallen ontstaan. Hetzelfde gebeurt met de drie delen van getal y. De vijf getallen van x worden paarsgewijs vermenigvuldigd met de vijf getallen van y. De dan gevormde vijf producten worden wederom via formules omgevormd tot vijf getallen, die aan elkaar gevoegd kunnen worden om zo het eindproduct te leveren.
Waar oorspronkelijk (volgens de klassieke methode) drie maal drie vermenigvuldigingen nodig zouden zijn, kan met dit algoritme worden volstaan met vijf vermenigvuldigingen. Dit levert een besparing van 5/9 op. Daar staat tegenover dat er nogal veel optellingen en aftrekkingen en “kleine” vermenigvuldigingen of delingen nodig zijn. Bijvoorbeeld vermenigvuldigen met 2 of 3, of delen door 2 of door 3. Daardoor is Toom-3 pas bij grotere getallen sneller dan Klassiek of Karatsuba.
De complexiteit van het Toom-3 algoritme in grote O notatie is $O(n^{\log(5)/\log(3)})$ *of ongeveer* $O(n^{1,465})$.
Opm. In dit document wordt de benaming Toom-Cook en Toom-3 door elkaar heen gebruikt, maar er wordt telkens hetzelfde mee bedoeld.

Schönhage-Strassen Algoritme (SSA)

In 1971 bedachten Volker Strassen en Arnold Schönhage dat een veel verder gaande D&C (Verdeel en Heers) strategie mogelijk was door gebruik te maken van het convolutie theorema. Daarbij wordt een getal verdeeld in een aantal stukjes (elementen genoemd). Elk element bestaat uit een gelijk aantal cijfers of, in computertermen, gelijk aantal bits. Het laatste element wordt zo nodig aangevuld met nul-bits. Dergelijke getallen kunnen worden beschouwd als een vector. Op twee van deze vectoren kan een convolutie worden toegepast met als resultaat het product van deze twee getallen.[9] Voor grote getallen kan met het SSA algoritme een aanzienlijke versnelling worden bereikt. De complexiteit van het SSA algoritme is $O(n \log n (\log \log n))$.

Convolutie

Convolutie (samenvouwing) is een wiskundige bewerking op twee functies met een nieuwe functie (de convolutie) als resultaat. Een getal kan worden gezien als een functie (vector) en twee getallen met elkaar vermenigvuldigd kan gezien worden als het resultaat van een convolutie.
Volgens het "convolution theorem" kan de (acyclische) convolutie van twee vectoren a en b worden gevonden door de Discrete Fourier Transformatie (DFT) van beide vectoren te nemen, de aldus gevormde vectoren paarsgewijs (element voor element) met elkaar te vermenigvuldigen (dat noemen we in dit document de "Dyadic Stage") en de producten weer met een inverse DFT te bewerken zodat het eindproduct ab wordt gevormd.
Eerst wordt ingegaan op de DFT, waarna later wordt ingegaan op convolutie.

Discrete Fourier Transformatie

De Discrete Fourier Transformatie (DFT) is een wiskundige bewerking van (onder meer) een reeks getallen, die vooral een toepassing vindt in de digitale signaalverwerking. Met de DFT kan een frequentiespectrum van een digitaal signaal worden bepaald. Maar DFT kan ook worden gebruikt voor het uitvoeren van discrete convoluties.

Wanneer een DFT wordt toegepast op een reeks getallen, dan ontstaat een nieuwe reeks met hetzelfde aantal getallen als de oorspronkelijke reeks. Dit proces wordt het "forward" proces genoemd. Het proces is omkeerbaar (reversibel) zodat met een "inverse" operatie de oorspronkelijke reeks weer kan worden verkregen.

De Discrete Fourier Transformatie is een bewerkelijk proces. Voor lange reeksen vergt dat veel computertijd. In 1965 publiceerden James W. Cooley van IBM en John W. Tukey van Princeton een methode om een DFT zeer efficiënt op een computer uit te voeren; een methode waarbij "nested loops" worden gebruikt of routines recursief worden aangeroepen. Deze methode wordt Fast Fourier Transform (FFT) genoemd. In 1966 werd een vergelijkbare methode gepubliceerd door W.M. Gentleman en G. Sande.

Wanneer de FFT wordt gebruikt bij het vermenigvuldigen van getallen, dan worden de getallen gesplitst in een aantal gelijke delen; dat wil zeggen, een gelijk aantal decimale

cijfers, of in computertermen, een gelijk aantal bits. Deze delen (elementen genaamd) vormen dan de input-reeks voor de FFT. Het aantal elementen van de input-reeks wordt N genoemd (maar wordt soms ook aangeduid met D) en moet bij voorkeur een macht van twee zijn. De exponent van twee wordt meestal aangegeven met de kleine letter k ($N=2^k$). Indien het aantal elementen van de input-reeks kleiner is dan een macht van twee dan wordt de reeks aangevuld met elementen die een waarde van nul hebben.

Bij de Cooley-Tukey methode, zowel als bij de Gentleman-Sande methode, worden de elementen, waaruit de te vermenigvuldigen getallen bestaan, dusdanig opgeslagen in het computergeheugen dat ze na bewerking weer op dezelfde plek in het computergeheugen kunnen worden teruggezet. Een bewerking gebeurt op twee elementen tegelijk, en bestaat uit het bepalen van de som en het verschil van de twee elementen. De som wordt teruggeplaatst in de oorspronkelijke ruimte van het eerste element en het verschil wordt teruggeplaatst in de ruimte van het tweede element. Op deze wijze wordt een zeer geheugen-efficiënt proces verkregen.

Op een zeer bepaalde volgorde worden telkens twee elementen genomen en bewerkt, waarna twee volgende elementen worden behandeld. Dit gaat zo door tot alle elementen aan de beurt zijn geweest. Dan volgt een volgende ronde, waarbij een andere samenstelling van twee elementen plaatsvindt. Zo komen per ronde alle elementen een keer aan de beurt. Het aantal ronden is afhankelijk van het aantal elementen N ($=2^k$) en is gelijk aan k. Dit proces wordt uitgevoerd voor beide input-getallen. Daarna worden de elementen van beide input-getallen paarsgewijs met elkaar vermenigvuldigd. De aldus gevormde producten worden dan met een inverse FFT bewerkt waarna ze worden geïntegreerd om zo het eindresultaat te vormen.

Wanneer deze berekening schematisch wordt uitgebeeld dan ontstaat een vorm die wel lijkt op een vlinder. Daarom wordt deze berekening de *"butterfly"* genoemd[4]. Voor de forward *butterfly* wordt het schema van Gentleman-Sande gebruikt en voor de inverse *butterfly* het schema van Cooley-Tukey.

Er zijn vele vormen van FFT algoritmes ontwikkeld; vaak is de input van een FFT een reeks complexe getallen, die in computers worden vastgelegd in de vorm van floating point getallen. Voor het vermenigvuldigen van grote getallen volgens het SSA algoritme is het echter voor de hand liggend om integer getallen te gebruikten. De voor SSA meest geëigende vorm van FFT is de *Number-theoretic transform.*

Number-theoretic Transform

De *Number-theoretic transform* (NTT) is een vorm van fast fourier transform die werkt met integer getallen in het domein modulo een getal[5]. Dit getal (de modulus) kan een priemgetal zijn, maar dat is niet per se noodzakelijk. Een bijzondere vorm van NTT is de Fermat transform modulo 2^n+1. (De term "Fermat-transform" is enigszins misleidend. De modulus van een Fermat-transform heeft wel de vorm 2^n+1, maar de exponent n hoeft zelf niet perse een macht van 2 te zijn. Bij een Fermat getal is n wel een macht van 2). [3]
Deze vorm speelt zich dus af in het domein van integers modulo (2^n+1) en wordt over het algemeen gebruikt in het SSA algoritme. De formule voor deze vorm van NTT is:

$$X_j \equiv \sum_{i=0}^{N-1} x_i \, g^{ij} \, (mod\, m)$$

In deze formule wordt de reeks met N getallen x_i getransformeerd naar de reeks X_j, ook bestaande uit N getallen, waarbij zowel i als j gaan van 0 tot en met N-1. De reeks x_i kan gezien worden als input terwijl de reeks X_j kan beschouwd worden als output. Verder is g de n-de eenheidswortel modulo m, waarbij m een getal is van de vorm: $2^n + 1$. De getallen die de output-reeks X_j vormen zijn allen (mod m). Verder moet worden opgemerkt dat N iets anders is dan n.

Eenheidswortels

De n-de eenheidswortel (root of unity) is het getal dat tot de n-de macht verheven als resultaat 1 oplevert[8]. Meestal zal dat een getal uit het complexe domein zijn. Bijvoorbeeld i (= √-1) is de 4-de eenheidswortel want $i^4 = 1$. Natuurlijk is 1 ook een wortel die tot de vierde macht 1 als resultaat oplevert, maar wortel 1 is in de wiskunde vaak niet interessant. Daarom is het begrip *primitieve eenheidswortel* geïntroduceerd; een n-de eenheidswortel x is primitief indien $x^n = 1$ en alle machten < n géén 1 opleveren.

Ook in het domein van getallen modulo m komt het begrip eenheidswortel voor[7].
Een k-de eenheidswortel modulo m is het getal x dat tot de k-de macht verheven (mod m) als resultaat 1 oplevert. Een k-de eenheidswortel x is primitief indien $x^k \equiv 1$ (mod m) en elke macht voor x kleiner dan k geen 1 (mod m) oplevert.

Fourier Transformatie volgens Number-Theoretic Transform

Als we de formule voor de NTT analyseren dan zien we als "input" een rij van N elementen x, en als "output" een rij van N elementen X, waarbij $N = 2^k$. Elk element van X (X_j) wordt gevormd door de som van de elementen x, vermenigvuldigd met een constante (de n-de eenheidswortel) tot een bepaalde macht ij. We kunnen nu de formule uitbeelden in de vorm van een tabel, waarbij de kolommen worden gevormd door de "input" en de rijen door de "output". Als voorbeeld nemen we een een rij van N elementen, met $N=2^k$, waarbij $k = 3$; dus $N=2^3 = 8$. De exponenten die worden gevormd door ij kunnen we als eerste in een tabel plaatsen:

Input i =	0	1	2	3	4	5	6	7
Output: j = 0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	8	10	12	14
3	0	3	6	9	12	15	18	21
4	0	4	8	12	16	20	24	28
5	0	5	10	15	20	25	30	35
6	0	6	12	18	24	30	36	42
7	0	7	14	21	28	35	42	49

Tabel 1

Nemen we nu als modulus bijv. $2^8+1 = 257$, dan is de achtste eenheidswortel gelijk aan 4. want $4^8 \equiv 1 \pmod{257}$. We kunnen nu de exponenten uit tabel 1 toepassen op de eenheids-wortel 4 en krijgen dan (rekening houdend met modulus 257) de resultaten in tabel 2.

Input i =	0	1	2	3	4	5	6	7
Output: j = 0	1	1	1	1	1	1	1	1
1	1	4	16	64	256	253	241	193
2	1	16	256	241	1	16	256	241
3	1	64	241	4	256	193	16	253
4	1	256	1	256	1	256	1	256
5	1	253	16	193	256	4	241	64
6	1	241	256	16	1	241	256	16
7	1	193	241	253	256	64	16	4

Tabel 2

Omdat alle waarden (mod 257) zijn kunnen deze ook negatief worden genomen: 256 wordt dan -1; 253 wordt -4; 241 wordt -16 en 193 wordt -64. We zien het resultaat in tabel 3.

Input i =	0	1	2	3	4	5	6	7
Output: j = 0	1	1	1	1	1	1	1	1
1	1	4	16	64	-1	-4	-16	-64
2	1	16	-1	-16	1	16	-1	-16
3	1	64	-16	4	-1	-64	16	-4
4	1	-1	1	-1	1	-1	1	-1
5	1	-4	16	-64	-1	4	-16	64
6	1	-16	-1	16	1	-16	-1	16
7	1	-64	-16	-4	-1	64	16	4

Tabel 3

We houden nu de waarden 1 en -1,   4 en -4,   16 en -16 en 64 en -64 over. Dit worden de "twiddle factors" genoemd en met deze factoren worden de elementen van de input vermenigvuldigd. We nemen hierbij aan dat de input wordt gevormd door een getal dat in acht gelijke delen (gelijk aantal cijfers, of gelijk aantal bits) is verdeeld en dat deze delen de input vormen voor de kolommen 0 t/m 7.
Wanneer we nu in de tabel alle elementen van kolom 0 vermenigvuldigen met de input-waarde die bij kolom 0 hoort, en dit voor alle kolommen zo doen, dan voeren we (handmatig) een complete Fourier Transformatie uit. De som van de producten van de rijen uit de tabel vormt de output.
Als we de output van rij 0 overbrengen naar de input voor kolom 0 en de output van rij 1 naar de input voor kolom 1 enz. dan kunnen we met deze nieuwe input de *inverse Number Theoretic Transform* toepassen. Daarbij gebruiken we dezelfde tabel met dezelfde "twiddle factors" en krijgen dan weer de oorspronkelijke input. De formule voor de *inverse Number Theoretic Transform is:*

$$x_i \equiv \frac{1}{N} \sum_{j=0}^{N-1} X_j g^{ij} (mod\, m)$$

De formules voor de *NTT* en de *inverse NTT* zijn nagenoeg gelijk; alleen moeten de elementen bij de inverse vorm nog worden gedeeld door *N (mod m)*. Ook hier zijn alle getallen *(mod m)*. Wel blijkt dat de volgorde gewijzigd is; kolom 0 blijft gelijk, maar wat oorspronkelijk in kolom 1,2,3,4,5,6,7 stond, staat nu in kolom 7,6,5,4,3,2,1.

Fast Fourier Transform

Als we nauwkeurig kijken naar de tabel van het voorbeeld dan kunnen we een paar symmetrieën opmerken; de linker kolommen (kolom 0 t/m 3) bevatten dezelfde factoren als de rechter kolommen (kol. 4 t/m 7), alleen de tekens kunnen verschillen. Voor de even genummerde rijen zijn de tekens gelijk, maar voor de oneven genummerde rijen zijn de tekens tegengesteld. Hiervan kan slim gebruikt gemaakt worden door de input-waarden van kolom 0 en kolom 4 bij elkaar op te tellen en van elkaar af te trekken. De som is dan van toepassing op de even rijen en het verschil is bruikbaar voor de oneven rijen.
Stellen we ons nu een ***container*** voor, waarin plaats is voor *N* elementen (in het voorbeeld dus 8 elementen, genummerd 0 t/m 7). Elk element kan een waarde bevatten die begrensd is tot de gebruikte modulus, in ons voorbeeld dus begrensd **tot** $2^{8+1} = 257$. We beginnen met het plaatsen van de input-waarden in de elementen van de container. We nemen nu kolom 0 en kolom 4, ofwel element 0 en element 4 van de container; we bepalen de som van element 0 en element 4 en plaatsen dat terug in element 0; we bepalen het verschil van element 0 en element 4 en plaatsen dat terug in element 4. Vóór of ná het berekenen van het verschil moeten we nog het element vermenigvuldigen met de twiddle factor (daarover straks méér).
Op dezelfde manier kan voor elementen 1 en 5 de som en het verschil worden bepaald. Evenzo voor kolommen 2 en 6 en voor kolommen 3 en 7. De container bevat nu acht elementen met een compleet andere inhoud dan aan het begin.

Het proces kan nu met deze nieuwe inhoud herhaald worden, maar dan met element 0 gepaard aan element 2; element 1 gepaard aan element 3; 4 gepaard aan 6 en 5 aan 7.
Er volgt nu nog een laatste keer waarbij element 0 en 1 samen genomen worden; element 2 en 3; 4 met 5 en 6 met 7. In plaats van acht maal het berekenen van 8 elementen, hoeft er nu maar *k* maal (*k* = 3) te worden berekend wat de nieuwe waarde van 8 elementen is. Dit is een aanzienlijke versnelling door het principe van de Fast Fourier Transform.

Twiddle Factors

De twiddle factors in tabel 3 geven een beeld van het soort factoren en de plaats waar deze factoren worden toegepast. Maar als bij FFT de rijen van de tabel worden samengevoegd kloppen de factoren niet (geheel) meer. Dankzij de slimme algoritmes van Cooley-Tukey en van Gentleman-Sande worden deze factoren eenvoudig berekend en toegepast (Zie het uitgewerkte voorbeeld in C^{++} code).
Zoals bij het voorbeeld in tabel 3 te zien was zijn er bij N = 8 vier verschillende twiddle-factoren van toepassing. Het blijkt dat voor elke waarde van N het aantal verschillende twiddle factoren gelijk is aan N/2.
De eerste factor moet zijn: een primitieve eenheidswortel van de orde van N. Dus bij k=3 en $N=2^k=8$ moet de eerste twiddle-factor een primitieve 8ste machts eenheidswortel zijn. Bij integers in het domein (mod 2^n+1) zijn alle twiddle factors machten van twee. Bij k = 3 is de eerste factor $2^{(n/4)}$; de volgende factoren zijn $2^{(2n/4)}$ en $2^{(3n/4)}$). De laatste twiddelfactor is 2^n. Al deze factoren zijn modulus 2^n+1. De laatste factor heeft dus de waarde 2^n (mod 2^n+1) = -1.

(De hoofdletter N wordt hier gebruikt om het aantal elementen aan te geven; de kleine letter n wordt gebruikt als exponent van 2 om de grootte van een getal aan te geven (zoals in de modulus 2^n+1). Tevens wordt n gebruikt om het aantal bits aan te duiden, dat nodig is om een getal tot 2^n in op te bergen.)

Convoluties

We hebben nu gezien dat een getal kan worden gesplitst in elementen en hoe op deze elementen een Fast Fourier Transform kan worden toegepast. Als we twee getallen *a* en *b*, willen vermenigvuldigen volgens het SSA algoritme, dan kan dit volgens het convolutie theorema door de Discrete Fourier Transformatie (DFT) te berekenen van zowel *a* als *b;* de elementen van deze DFT's paarsgewijs met elkaar te vermenigvuldigen en op deze producten weer de Inverse Discrete Fourier Transformatie (IDFT) toe te passen. De resultaten daarvan worden weer samengevoegd tot het eindresultaat: *ab*. Schematisch ziet dat er zo uit:

Product *ab* = IDFT (DFT (*a*) * DFT (*b*)) [2]

Voor het SSA algoritme wordt de DFT uitgevoerd als een Number-theoretic transform; we kunnen in dit geval de termen IDFT en DFT vervangen door inverse NTT en NTT.

Formeel belooft het SSA algoritme ons het product $ab \pmod{2^n+1}$, indien de input-getallen *a* zowel als $b \leq 2^n+1$.[13] We kunnen dat zo vertalen: als de input-getallen niet groter zijn dan *n* bits, dan geeft het SSA algoritme ons het product van deze getallen mod (2^n+1) .
Het product van twee getallen van maximaal *n* bits levert een getal van maximaal *2n* bits op. We zijn natuurlijk niet geïnteresseerd in het product *ab* als dat beperkt is tot *n* bits (wat het geval is als mod (2^n+1) wordt toegepast. Om het volledige product *ab* te krijgen moeten we *n* zo groot kiezen, dat $ab < 2^n+1$. We kunnen er ook voor kiezen om de input getallen te beperken tot *n/2* bits.

Er zijn verschillende vormen van convolutie die ook verschillende resultaten opleveren [2]:

De acyclische convolutie ***(acyclic convolution)*** levert ons het volledige product *ab*. Dit bereiken we door de input-getallen te beperken tot *n/2* bits. (Waar we hier van *n* spreken wordt de exponent van 2 bedoeld, waaruit de modulus (2^n+1) bestaat). Denken we aan de container met elementen van het FFT proces, dan kunnen we het aantal elementen op N stellen. Door slechts de helft van die elementen (*N/2)* te vullen en de overige elementen op nul te zetten (dit wordt *"Zero Padding"* genoemd) verkrijgen we het volledige product *ab*. We moeten er dan wel voor zorgen dat het aantal elementen, of de grootte van de elementen, voldoende is om de getallen *a* en *b* in de helft van de elementen te kunnen opbergen.

De cyclische convolutie ***(cyclic convolution)*** krijgen we indien we alle elementen van de container vullen. Het resultaat is dan $ab \pmod{2^n-1}$. Een voorbeeld hiervan zien we verderop. Dit wordt ook wel een *Mersenne Transform* genoemd, vanwege de vorm van de modulus. In sommige SSA implementaties wordt deze transformatie gebruikt[3]. We gaan er hier verder niet op in.

De negacyclische convolutie ***(negacyclic convolution)*** levert als resultaat $ab \pmod{2^n+1}$ op. Deze verkrijgen we door de input in de elementen met een wegings-factor te vermenig-vuldigen vóór de forward butterfly en het resultaat van de FFT na de inverse butterfly weer met een wegingsfactor te bewerken. Dit proces heet DWT (Discrete Weighted Transform)

en het oorspronkelijke SSA algoritme[2] maakt gebruik van DWT. De reden waarom het SSA algoritme gebruik maakt van de *negacyclic convolution* is om het algoritme recursief te kunnen toepassen. Tijdens de “Dyadic Stage” wordt de inhoud van de elementen paarsgewijs met elkaar vermenigvuldigd (mod 2^n+1) zodat de deel-producten weer passen in de elementen van de container. Als we voor de “Dyadic Stage” recursief het SSA algoritme met de negacyclische convolutie gebruiken dan sluiten het hogere en lagere niveau van de recursie mooi op elkaar aan; beide *(mod 2^n+1)*.

In de praktijk wordt het SSA algoritme niet altijd recursief en met de negacyclische convolutie gebruikt. Op het hoogste niveau is in ieder geval de acyclische convolutie nodig om het gehele product te verkrijgen. Indien SSA recursief wordt gebruikt dan wordt dit tot slechts 1 of 2 levels diep beperkt[13].

Content

De elementen van een container zijn zo ingericht dat ze elke mogelijke waarde kunnen bevatten modulus (2^n+1). De kleinste waarde is nul; de grootste waarde is 2^n. Bij het begin van het FFT proces worden de elementen gevuld, maar dat kan niet met elke waarde want gedurende het proces wordt de inhoud van de elementen bij elkaar opgeteld en met elkaar vermenigvuldigd. Daarom is er een beperking aan de waarde waarmee de elementen initieel kunnen worden gevuld.
Nemen we als voorbeeld getallen modulus 2^{16}+1. Deze getallen willen we gaan opbergen in een binair veld van 16 bits. De kleinste waarde is nul; de grootste waarde is 65.536. Elk van de verschillende waarden vanaf de kleinste tot de grootste kunnen in een binair veld van 16 bits worden opgeslagen (met uitzondering van 2^{16} = 65.536, maar daarover straks meer).
Nemen we verder k = 3 zoals we dat ook bij tabel 1,2 en 3 hebben gedaan. Dan krijgen we een tabel met 8 regels, waarvan de elementen eventueel bij elkaar opgeteld moeten worden. Van de 16 bits van het binaire veld moeten we er daarom drie (8 = 2^3) reserveren voor deze optellingen. Het aantal bits dat resteert moeten we delen door twee vanwege de paarsgewijze vermenigvuldiging. Dus 13 bits gedeelde door twee is naar beneden afgerond 6 bits.
De maximale waarde waarmee de elementen van de containers bij de aanvang van het FFT proces kunnen worden gevuld is dus in ons voorbeeld 6 bits of waarden tussen 0 en 63. Deze waarde wordt in de literatuur vaak aangeduid met *M*; in dit document wordt de aanduiding *"Content"* gebruikt.
Content is dus het aantal bits waarmee de elementen bij het begin van het FFT proces worden gevuld; ook de stukken waarin de input getallen worden verdeeld bestaan uit ditzelfde aantal bits. Bij een Content van *n* bits kan een waarde van 0 **tot** 2^n worden opgeslagen (2^n kan dus net niet meer).

Het SSA algoritme stap voor stap

We hebben nu voldoende informatie om het hele proces van het SSA algoritme te schetsen. We hebben twee (grote) getallen *a* en *b* die we willen vermenigvuldigen zodat we het product $c = a * b$ krijgen. We nemen aan dat de getallen *a* en *b* dezelfde orde van grootte hebben. Aan de hand van deze grootte bepalen we een *k*; een aantal elementen $N = 2^k$; en een modulus *m*. (Later zullen we zien hoe we dat doen). Voorts creëren we twee containers; één voor *a* en één voor *b*. De containers bevatten elk een aantal van *N* elementen; elk element kan een waarde (mod *m*) bevatten.

Omdat we de acyclische convolutie toepassen splitsen we getal *a* in maximaal *N/2* gelijke delen. Elk deel bevat een gelijk aantal bits, waarbij dat aantal gelijk is aan *Content*, en vullen daarmee de elementen van container *a*. We gebruiken slechts de eerste helft van de elementen; de niet (volledig) gevulde elementen vullen we aan met nul. Hetzelfde doen we met getal *b* en container *b*. (De containers zijn van gelijke grootte).

We gaan nu voor container *a* de FFT uitvoeren volgens de Gentleman-Sande methode, waarbij de twiddle factoren worden berekend en toegepast en de elementen worden opgeteld en afgetrokken. De elementen in container *a* hebben nu een andere inhoud gekregen. Alle bewerkingen (optellen, aftrekken en vermenigvuldigen met de twiddle factor) gebeuren modulus *m*, zodat het resultaat altijd past in een element van de container. Op gelijke wijze wordt de FFT op de elementen van container *b* uitgevoerd. Daarna worden de elementen van container *a* paarsgewijs vermenigvuldigd met de elementen van container *b*. Dus element nul van *a* vermenigvuldigd met element nul van *b*; element een van *a* met element een van *b* enz. Deze stap wordt in de literatuur aangeduid met “dyadic stage”.
Voor deze vermenigvuldigingen kunnen we gebruik maken van de klassieke methode, of de methode van Karatsuba of Toom-3. Maar we kunnen ook het SSA algoritme gebruiken (in ons voorbeeld programma zullen we de klassieke methode gebruiken).
De resultaten worden (mod *m*) in de elementen van container *c* geplaatst. (We noemen de container waarin het resultaat van de dyadic stage wordt geplaatst container *c*. In werkelijkheid is dit container *a* of container *b,* welke dat maakt niet uit).
De volgende stap is het uitvoeren van de inverse FFT volgens het schema van Cooley-Tukey. De resultaten worden weer geplaatst in de elementen van container *c*. Als laatste stap wordt de inhoud van de elementen gedeeld door N (mod *m*). De container *c* bevat nu de *acyclische convolutie* van getal *a* en getal *b*. De elementen van container *c* worden nu op de juiste volgorde gezet en samengevoegd tot het eindresultaat $c = a * b$.

Voorbeeld

Als voorbeeld nemen we de vermenigvuldiging van de klassieke methode. Getal a = 1234 en getal b = 5678. We creëren twee containers a en b elk met 8 elementen en gebruiken als modulus $2^{16}+1=65537$. In dit voorbeeld plaatsen we in elk element slechts één decimaal cijfer (de maximale *Content* bij de modulus $2^{16}+1$ is 6 bits of waarden van 0 t/m 63. We gebruiken nu als *Content* de waarde 10; dus de waarden 0 tot en met 9 en dat is ruimschoots binnen de marge die mogelijk is).

We gaan nu de elementen van container a vullen met de cijfers van het eerste getal: 4 in element 0; 3 in element 1; 2 in element 2 en 1 in element 3. De elementen 4 t/m 7 vullen we met nul. We voeren dus de ***acyclische convolutie uit***, zodat we een juist resultaat bereiken.

element :	0	1	2	3	4	5	6	7
	4	**3**	**2**	**1**	**0**	**0**	**0**	**0**

Container a

De elementen 0,1,2,3 van container b vullen we met de cijfers van het tweede getal; resp. 8,7,6,5 en elementen 4 t/m 7 vullen we ook met nul.

element:	0	1	2	3	4	5	6	7
	8	**7**	**6**	**5**	**0**	**0**	**0**	**0**

Container b

We voeren nu de convolutie uit zoals hierboven beschreven. Het resultaat komt in de elementen van container c (in werkelijkheid container a); die bevat nu:

element :	0	1	2	3	4	5	6	7
	32	**0**	**5**	**16**	**34**	**60**	**61**	**52**

Container a na de convolutie

De elementen worden nu in de juiste volgorde gezet; bijvoorbeeld door element 0 schijnbaar te verplaatsen naar virtueel element 8. Nu vertegenwoordigen de elementen 8 t/m 1 in aflopende volgorde de oplopende waarden 1, 10, 100, 1000 enz. Dit is precies gelijk aan het voorbeeld dat we gebruikten bij de klassieke methode.
De inhoud van de elementen wordt nu vermenigvuldigd met de bijbehorende macht van tien (net zoals we in het voorbeeld van de klassieke methode zagen).

Element 0 32 x 10^0 = 32
Element 7 52 x 10^1 = 520
Element 6 61 x 10^2 = 6.100
Element 5 60 x 10^3 = 60.000
Element 4 34 x 10^4 = 340.000
Element 3 16 x 10^5 = 1.600.000
Element 2 5 x 10^6 = 5.000.000

1234 x 5678 = 7.006.652

De kolommen in de container vertegenwoordigen blijkbaar elk een bepaalde waarde, die een macht van tien is en het is het *convolutie* proces dat er voor zorgt dat de cijfertjes in de juiste kolom terechtkomen. Bij de klassieke school-methode zorgen we ervoor dat een “carry” direct wordt geteld in de kolom met de hogere macht van tien, maar bij de SSA methode tellen we gewoon door in een element en zorgen we er later voor dat de “carries” worden verwerkt.

In dit voorbeeld hebben we de containers maar voor de helft gevuld. Wat gebeurt er wanneer we een container verder vullen? We nemen de proef op de som en voegen in element vier van container *a* een 1 toe en voeren dan de convolutie opnieuw uit. Het resultaat is:

element:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	
	32	**5**	**11**	**23**	**42**	**60**	**61**	**52**	**32**

Container *a* ná de convolutie van 11.234 x 5.678

We zien dat in element 4 een waarde 8 is bijgeteld; in element 3 een waarde van 7; in element 2 is een waarde van 6 bijgeteld en in element 1 een waarde van 5. Dat is precies wat we hadden kunnen verwachten, want de 1 van element vier in container *a* vertegenwoordigt een waarde van 10.000; dat vermenigvuldigd met 5.678 levert een bijtelling op van resp. 5, 6, 7 en 8 in de resp. elementen 1, 2, 3, en 4. Element 1 vertegenwoordigt de hoogste waarde en wel 10.000.000 of 10^7 en is nu ook gevuld (met 5).

Wat gebeurt er nu wanneer we vervolgens bijvoorbeeld een 4 toevoegen aan container *b* in element vier? Element vier vertegenwoordigt een waarde van 10.000 en wanneer we dus een 4 van dit element van container *b* vermenigvuldigen met de 1 van element vier van container *a* dan is het resultaat 4 maal 100.000.000. Er is in de containers echter geen element die de waarde 100.000.000 of 10^8 vertegenwoordigt!
Als we de convolutie opnieuw uitvoeren zien we het resultaat:

element:	0	1	2	3	4	5	6	7	
	36	9	19	35	58	60	61	52	36

Container *a* ná de convolutie van 11.234 x 45.678

element 0	36	x	1	=	36		
element 7	52	x	10	=	520		
element 6	61	x	100	=	6.100		
element 5	60	x	1.000	=	60.000		
element 4	58	x	10.000	=	580.000		
element 3	35	x	100.000	=	3.500.000		
element 2	19	x	1.000.000	=	19.000.000		
element 1	9	x	10.000.000	=	90.000.000		
					113.146.656	(mod 99.999.999) =	**13.146.657**
	11.234	x	45.678	=	513.146.652	(mod 99.999.999) =	**13.146.657**

In element vier is 4 x 4 = 16 bijgeteld; 4 x 3 = 12 is bijgeteld in element drie; 4 x 2 = 8 is bijgeteld in element twee en 4 x 1 is bijgeteld in element een. Dan blijft er nog 4 x 1 = 4 over (van de 1 die eerder is toegevoegd aan container *a)* die bijgeteld zou moeten worden in een element dat de waarde 10^8 vertegenwoordigt; maar dat element bestaat niet. De 4 wordt gewoon bijgeteld bij element 0. Dit is wat in het Engels “wrap around” wordt genoemd. Voor elementen met waarden, die niet meer passen in de container, wordt gewoon weer bij element 0 begonnen en de nieuwe inhoud wordt samengevoegd met de bestaande inhoud! Het resultaat van deze *convolutie* is daardoor **niet** gelijk aan het resultaat van de gewone vermenigvuldiging; maar als de resultaten (mod 99.999.999) worden genomen zijn deze **wel** aan elkaar gelijk. Mod 99.999.999 is niet zomaar willekeurig genomen, dit getal is gelijk aan *Content* tot de macht acht (het aantal elementen van de container) minus één; is dus modulus 10^8–1. Dit is dus het voorbeeld van de ***cyclische convolutie***.

Als we ook het voorbeeld willen uitwerken met de ***negacyclische convolutie*** dan moet de input eerst met een wegingsfactor worden vermenigvuldigd. Deze wegingsfactor is voor elk element van de container verschillend en is afhankelijk van het aantal elementen in de container *N* en de grootte van de modulus: de *n* van mod (2^n+1). Concreet is de wegingsfactor twee tot een bepaalde exponent; de basiswaarde voor deze exponent is *n/N* en deze exponent wordt vermenigvuldigd met *j* als index voor de elementen.
In ons voorbeeld is *N=8*; *n=16* (van de modulus 2^n+1); en de index *j* gaat van 0 t/m 7. De waarde *n/N=16/8=2* en de exponenten zijn dus: 0,2,4,6,8,10,12 en 14. De wegingsfactoren zijn dus 2^0, 2^2, 2^4 enz. tot 2^{14}. De input van de elementen 0 t/m 7 wordt dus vermenigvuldigd met resp. 1,4,16,64,256,1024,4096 en 16392; alles mod (2^{16}+1). De input (van ons voorbeeld van de klassieke methode: 1234 x 5678) ziet er dan, nadat de weging is toegepast, zó uit:

element :	0	1	2	3	4	5	6	7
	4	**12**	**32**	**64**	**256**	**0**	**0**	**0**

Container a, input na vermenigvuldiging met wegingsfactor

element:	0	1	2	3	4	5	6	7
	8	**28**	**96**	**320**	**1024**	**0**	**0**	**0**

Container b, input na vermenigvuldiging met wegingsfactor

Met de aldus gewogen input wordt de Number-theoretic transform uitgevoerd. De elementen van container *a* en *b* worden met elkaar vermenigvuldigd. Daarna wordt de inverse Number-theoretic transform uitgevoerd en de output wordt in container *a* geplaatst (dit kan ook container *b* zijn, dat maakt niet uit) en ziet er zó uit:

element :	0	1	2	3	4	5	6	7	8
	224	**65519**	**32759**	**24572**	**53247**	**30720**	**7808**	**1664**	**224**

Container a, output van negacyclische convolutie, vóór weging

De volgorde van de elementen is door de transformatie weer veranderd: element 0 vertegenwoordigt nog steeds de waarde 1, maar het element 7 vertegenwoordigt nu de waarde 10; element 6 de waarde 100 enz. Door de inhoud van element 0 naar het virtuele element 8 te verplaatsen, vertegenwoordigen de elementen 8 t/m 1 in aflopende volgorde de waarden 1 t/m 100.000.000 in oplopende volgorde.
De output dient nu nog “gewogen” te worden; dus gedeeld worden door dezelfde wegingsfactoren, waarmee de input is vermenigvuldigd. Bovendien moet de output ook nog gedeeld worden door de factor *N* (zie de formule voor de *inverse NTT*). Verder dient het delen ook mod (2^n+1) te gebeuren!

Modulair delen is, in termen van computer cycles, een kostbaar proces . Gelukkig is er bij een deler, bestaande uit twee tot een zekere macht, en een modulus van de vorm (2^n+1) een truc die het delen een stuk eenvoudiger maakt.

Het delen kan worden omgezet in vermenigvuldigen door de exponent van de modulus te vermenigvuldigen met twee en te verminderen met de exponent van de deler. In ons voorbeeld zijn de delers gelijk aan de wegingsfactoren: voor de elementen 8 t/m 1 zijn dat resp. $2^0,2^2,2^4,2^6,2^8,2^{10},2^{12},2^{14}$. De exponent van de modulus maal twee is 32; deze moet dus verminderd worden met resp. 0,2,4,6,8,10,12 en 14. De deling door *N* kan tegelijk met de deling door de wegingsfactoren worden meegenomen. Dus moet van de 32 nog eens extra 3 worden afgetrokken $(N=2^3)$. De exponenten worden dus voor de elementen 8 t/m 1 resp. 29,27,25,23,21,19,17 en 15. De waarde van de elementen moet nu vermenigvuldigd worden met resp. $2^{29},2^{27},2^{25},2^{23},2^{21},2^{19},2^{17}$ en 2^{15} en dat allemaal $(mod\ 2^{16}+1)$. Het resultaat is:

element:	0	1	2	3	4	5	6	7	8
	28	**9**	**19**	**35**	**58**	**60**	**61**	**52**	**28**

Container *a* ná de negacyclische convolutie van 11.234 x 45.678, ná weging

element 0	28	x	1	=	28		
element 7	52	x	10	=	520		
element 6	61	x	100	=	6.100		
element 5	60	x	1.000	=	60.000		
element 4	58	x	10.000	=	580.000		
element 3	35	x	100.000	=	3.500.000		
element 2	19	x	1.000.000	=	19.000.000		
element 1	9	x	10.000.000	=	90.000.000		
					113.146.648	(mod 100.000.001)=	**13.146.647**
	11.234	x	45.678	=	513.146.652	(mod 100.000.001)=	**13.146.647**

Het resultaat van de negacyclische convolutie (mod 10^8+1) is nu gelijk aan de vermenigvuldiging van 11.234 x 45.678 (mod 10^8+1). Dit is dus het voorbeeld van de ***negacyclische convolutie***.

Het verhaal van de negacyclische convolutie is echter nog niet af. Onder sommige omstandigheden dient er nog een correctie plaats te vinden op de weging: indien de inhoud van een element na de weging een grotere waarde heeft dan theoretisch mogelijk is, dan moet de inhoud worden gecorrigeerd (worden verminderd) met de modulus; dus verminderd met (2^n+1). Hierdoor kan de inhoud van een element een negatieve waarde krijgen. Bij de implementatie dient daar rekening mee te worden gehouden.

De theoretisch maximale waarde die een element kan bevatten is voor elk element verschillend en wordt als volgt vastgesteld: De index *j* van een element wordt met één verhoogd en dan vermenigvuldigd met de *Content* van een element in het kwadraat (in ons voorbeeld is dat 10^2, maar in de praktijk zal de *Content* een aantal bits zijn, die de waarde 2^{Content} vertegenwoordigt.

De index *j* van een element is de index die ontstaat na de omkering door het FFT proces en is dus in ons voorbeeld j=0 voor element 8 enzovoorts tot index j=7 voor element 1.

Er is hier sprake van twee moduli en dat kan tot verwarring leiden. Modulus (2^n+1) in de formule van de NTT en modulus (10^8+1) die wordt toegepast op het eindresultaat van de negacyclische convolutie en die verder in dit document als *modulusN* wordt aangeduid.

Zoals in het voorbeeld hier is uitgewerkt zijn dit twee verschillende moduli. Als de negacyclische convolutie echter wordt toegepast in de Dyadic Stage, dan is het de bedoeling dat beide moduli samen vallen en dus aan elkaar gelijk zijn.

De grootte van de elementen en de containers

Bij een vermenigvuldiging van twee verschillende getallen gebruiken we twee containers *a* en *b,* die even groot moeten zijn. (Indien we het kwadraat van een getal willen berekenen dan kunnen we volstaan met één container.) De grootte van een container wordt bepaald door het aantal elementen in een container en de grootte van een element. Het aantal elementen *N* wordt gegeven door $N=2^k$. De grootte van een element is afhankelijk van de modulus die wordt gebruikt. De modulus is 2^n+1 (*n* is een andere dan *N*). De maximale waarde die een element moet kunnen bevatten is modulus-1; is dus 2^n. Een element dat uit *n* bits bestaat kan de maximale waarde 2^n-1 bevatten en voldoet dus, mits we een oplossing kunnen bedenken voor de (zeldzame) situaties dat de waarde 2^n moet worden opgeslagen.

Bij het begin van de forward butterfly procedure worden de containers geladen met de te vermenigvuldigen getallen. Die getallen moeten worden gesplitst in delen; hoeveel dat hangt af van de convolutie die we willen toepassen. Bij de *acyclische* convolutie mogen we maar de helft van de elementen vullen. Dus moeten we de getallen in maximaal *N/2* delen splitsen. De overige elementen worden gevuld met nullen (*zero-padding*).
Dat opsplitsen van een getal kan op verschillende manieren; we kunnen opsplitsen in decimale cijfers zoals in het voorgaande voorbeeld is uitgewerkt. De kolommen (elementen) krijgen dan resp. de waarde 1,10,100.. enz. Bij het samenstellen van het eindresultaat wordt dan de inhoud van een elementen vermenigvuldigd met de bijbehorende waarde (1,10,100.. enz.).
In een computerprogramma worden getallen meestal weergegeven als een rij bits. We kunnen zo'n getal opsplitsen in delen met een gelijk aantal bits. We kunnen een getal bijvoorbeeld opsplitsen in delen van 32 bits. De waarde van de elementen wordt dan resp. $1,2^{32},2^{64}$,.. enz. Het aantal bits waarmee we de elementen bij de aanvang van het forward butterfly proces kunnen vullen is beperkt tot *Content,* zoals we hebben geconstateerd bij de paragraaf genaamd Content. De effectieve *Content* is maximaal *(n-k)/2* bits, maar mag ook minder zijn.
Omdat het werken met aantallen bits, die géén veelvoud zijn van acht, in de meeste computers nogal inefficiënt verloopt, kan *Content* het beste naar beneden worden afgerond tot een veelvoud van acht. Bij veelvouden van acht bits kan er namelijk gewerkt worden met complete bytes. Bij de implementatie die hier wordt besproken is er voor gekozen om bij grotere getallen de *Content* af te ronden op een veelvoud van 32. Daardoor kunnen de input-getallen in veelvouden van een computerwoord van 32 bits worden verwerkt.

De keuze van *k*

De waarde van *k* speelt een belangrijke rol bij het bepalen van de grootte van de container en de effectieve inhoud die er in kan worden geplaatst (dus de input). Het aantal elementen van een container is 2^k. Hiervan is, bij de acyclic convolution, in aanvang de helft bruikbaar; dus $2^k/2$ elementen.
De minimale grootte van een element is $2^k/2$ bits om het aantal verschillende twiddle factors te kunnen herbergen (zie twiddle factors).
De *Content* van één element (van minimale grootte) is dan: $(2^k/2 - k)/2$ bits (zie vorige paragraaf). De *Content* van de gehele container is dan: $2^k/2$ elementen maal $(2^k/2 - k)/2$ bits.
Indien *k* één groter wordt, dan nemen het aantal elementen en de (minimale) grootte van de

elementen met (bijna) een factor twee toe. (Bijna omdat de *Content* van een element met de factor k wordt verminderd). Dus de capaciteit van de gehele container (in aantal bits) neemt dan met ongeveer een factor vier toe!
We kunnen ook de elementen van de container met een factor twee vergroten. Daarmee neemt de capaciteit van de container toe met een factor twee.

Indien de factor k bekend is kunnen we de capaciteit van de container wel berekenen. Maar indien we k niet weten, maar wel de capaciteit die we nodig hebben om de input in de container kwijt te kunnen, dan kunnen we slechts een ruwe schatting maken van de factor k. In de praktijk blijkt dat de optimale waarde van k het beste proefondervindelijk kan worden vastgesteld. Dit zullen we dan ook tijdens het testen van de SSA implementatie doen.

Acyclische en negacyclische convolutie met elkaar vergeleken

De acyclische convolutie levert het complete product *ab* van getallen *a* en *b*. De negacyclische convolutie levert slechts het product *ab mod* (2^n+1). Het is voor de hand liggend om op het hoogste niveau van het SSA algoritme de acyclische convolutie te gebruiken. De negacyclische convolutie kan prima worden toegepast op het lagere niveau (tijdens de dyadic stage). Toch is het geen uitgemaakte zaak dat op het lagere niveau de negacyclische convolutie de beste oplossing is en niet de acyclische convolutie.
De voor- en nadelen van de twee verschillende convoluties zijn hier opgesomd:

Acyclische convolutie	*Negacyclische convolutie*
Het resultaat van deze convolutie is het volledige product *ab* van de beide factoren *a* en *b*.	Het resultaat van deze convolutie is het product *ab mod* (2^n+1) van factoren *a* en *b*. Deze convolutie levert dus niet het volledige product.
Slechts de helft van de elementen van de containers kunnen gevuld worden met input-data.	Alle elementen van de containers kunnen gevuld worden met input-data.
De grootte van een element moet deelbaar zijn door de helft van K: dus deelbaar door $2^{(k-1)}$. Dit levert een iets grotere flexibiliteit op bij het bepalen van een geschikte grootte van de elementen van de container.	De grootte van een element moet deelbaar zijn door K: dus deelbaar door 2^k.
Er is geen weging nodig.	De elementen moeten vermenigvuldigd worden met een wegingsfactor.
Na de backward FFT dienen de elementen te worden gedeeld door *N mod* (2^n+1). Indien echter *Content* in bits wordt beperkt tot: *(Size of element in bits)/2 – k*, dan kan het toepassen van modulus (2^n+1) achterwege blijven. Er kan dan worden volstaan met een deling door *N* die wordt uitgevoerd als een simpele shift-bewerking naar rechts.	De elementen dienen niet alleen gedeeld te worden door *N* mod (2^n+1) maar ook door de wegingsfactor mod (2^n+1). Deze delingen kunnen worden omgezet in één vermenigvuldiging mod (2^n+1). Er dient gecontroleerd te worden of de waarde van de elementen het theoretische maximum te boven gaat. In dat geval dient de waarde van een element te worden verminderd met de modulus (2^n+1). De waarde kan dan negatief worden, waardoor weer extra maatregelen nodig zijn.

Samengevat: de negacyclische convolutie is aanzienlijk complexer dan de acyclische convolutie, maar maakt het mogelijk om tweemaal zoveel elementen te vullen. Testen zullen aantonen wat de voordelen qua snelheid zijn bij het toepassen van de negacyclische convolutie tijdens de Dyadic Stage.

De Implementaties

Om de verschillende algoritmen met elkaar te kunnen vergelijken qua uitvoeringstijd dienen we over implementaties van deze algoritmen te beschikken. In de Lba-library (zie een volgend hoofdstuk) zijn reeds routines beschikbaar voor het kwadrateren volgens de klassieke methode, volgens het Karatsuba algoritme en volgens het Toom-3 algoritme. Al deze routines zijn bedoeld voor een 32-bits Intel architectuur in een Windows omgeving. Het is dus het meest praktisch om een routine voor kwadrateren volgens het SSA algoritme te implementeren, eveneens voor een 32-bits Windows omgeving.
Kwadrateren is een beperking ten opzichte van vermenigvuldigen. Toch komt kwadrateren in veel toepassingen voor en vormt een belangrijk deel van alle vermenigvuldigingen.[13]
Daarom is het te rechtvaardigen om vergelijkingen specifiek voor kwadrateren uit te voeren. De routines zijn in assembler geschreven maar zijn aan te roepen vanuit een C^{++} omgeving zowel als vanuit een assembler omgeving.

De implementatie van het Klassieke algoritme

De implementatie van een klassieke vermenigvuldiging is een vrij eenvoudige routine die in twee geneste loops telkens een computer-woord van het vermenigvuldigtal vermenigvuldigt met alle woorden van het vermenigvuldiger en de resultaten optelt in het outputveld, waarna het volgende computerwoord van het vermenigvuldigtal aan de beurt komt. Dit alles rekening houdend met de positionering van de producten bij het optellen in het outputveld. In de Lba-library is deze functie beschikbaar onder de naam Mul. Er is in deze library ook een aparte functie aanwezig voor kwadrateren onder de naam Smul (Square Multiplication).

De implementatie van het Karatsuba algoritme

Er zijn twee Karatsuba routines in de Lba-library beschikbaar: KARAT voor vermenigvuldigen van twee getallen en SKARAT voor kwadrateren van een getal. Bij het Karatsuba algoritme wordt gebruik gemaakt van recursiviteit. Bij Skarat wordt de input telkens in twee delen gedeeld. Deze delen vormen twee getallen waaruit een derde getal wordt berekend. Voor elk van de drie getallen wordt een Karatsuba routine weer (recursief) aangeroepen; de Skarat routine voor de oorspronkelijke delen en de Karat routine voor het berekende deel. Dit gaat zo door totdat de getallen zo klein zijn dat het efficiënter is om ze met het klassieke algoritme verder te berekenen.
De input bestaat uit een getal in de vorm van een aantal computer-woorden, waarbij dat aantal, in het ideale geval, een macht van twee is. Meestal voldoet de input niet aan de ideale situatie. Daarom is er voor gekozen om dynamisch een werkveld te creëren dat wel de ideale vorm heeft en waarin het input-getal wordt gekopieerd. Verder wordt een outputveld gemaakt, tweemaal zo groot als het inputveld en wordt een werkveld gemaakt voor tussenresultaten.
De totale benodigde geheugenruimte voor de werkvelden is circa acht maal de grootte van de input, in computerwoorden van 32 bits, afgerond naar boven tot een macht van twee. Van die achtmaal is éénmaal voor het input werkveld; tweemaal voor het output werkveld en vijfmaal als werkvelden voor tussenresultaten voor alle niveaus van recursiviteit.
Het resultaat wordt vanuit het output werkveld gekopieerd naar het externe resultaatveld.

Implementatie van Toom-Cook algoritme

Bij het Toom-Cook algoritme wordt eveneens gebruik gemaakt van recursiviteit. De input wordt telkens in drie delen gedeeld. Op deze delen worden enige berekeningen toegepast, waaruit vijf getallen ontstaan. Elk van deze vijf getallen wordt gekwadrateerd door de Toom-Cook routine weer (recursief) aan te roepen. Op het resultaat van de vijf kwadraten worden weer enige berekeningen toegepast, waarna deze getallen aan elkaar gevoegd worden om zo het eindresultaat te vormen[11].
De recursieve aanroepen gaan door totdat de getallen zo klein zijn, dat ze beter door het Karatsuba algoritme kunnen worden behandeld. Deze grens is gesteld op 32 woorden van 32 bits, dus 1024 bits.
Om een soepele overgang tussen Toom-Cook en Karatsuba te bereiken dient de (optimale) input een grootte te hebben van 1024 bits (of 32 woorden van 32 bits) maal een macht van drie. Net als bij Karatsuba worden dynamisch een aantal interne werkvelden gecreëerd: een inputveld gelijk aan de grootte van de optimale input; een outputveld tweemaal zo groot als het inputveld; en werkvelden voor de tussenresultaten ter grootte van driemaal de optimale input. Dit is dan voor alle niveaus van recursie samen. Het input-getal wordt gekopieerd naar het interne inputveld. Het eindresultaat wordt vanuit het interne outputveld gekopieerd naar het externe outputveld.

De implementatie van het SSA algoritme

Voor het SSA algoritme zijn twee implementaties gemaakt. Eén, geschreven in C^{++}, is bedoeld om het principe van SSA en Fast Fourier Transform (FTT) uit te testen en te demonstreren. Bij de tweede implementatie, geschreven in Assembler, ligt de nadruk meer op de uitvoeringssnelheid. Deze implementatie wordt gebruikt voor de vergelijking tussen de verschillende algoritmen. Beide implementaties berekenen het kwadraat van het input-getal.

De C^{++} implementatie van het SSA algoritme

De listing van de C^{++} implementatie (althans het relevante deel ervan) is in dit document opgenomen (aan het einde).
De C^{++} implementatie kent twee varianten: de eerste variant is de eenvoudigste en werkt volgens de acyclische convolutie. Deze variant levert het complete kwadraat van de input op. De tweede variant werkt volgens de negacyclische convolutie en levert het kwadraat van de input mod *modulusN* op. In de listing is te zien dat deze variant aanzienlijk complexer is dan de eerste.
In beide varianten heeft de container een vaste structuur met een gelimiteerd aantal elementen met een vaste grootte. De factor *k* wordt op 5 gesteld en de grootte van de elementen op 64 bits. De forward FFT en de inverse FFT zijn uitgevoerd in de vorm van een aantal geneste loops volgens het schema van resp. Gentleman-Sande en Cooley-Tukey.
Voor de “Dyadic Stage” wordt gebruik gemaakt van de klassieke methode van vermenigvuldigen via de routine Smul (Square multiply) uit de Lba library.
De input wordt gegenereerd aan de hand van een “requested number of bits” en bestaat dan uit een random getal dat via een PRNG routine wordt verkregen (PRNG staat voor Pseudo Random Number Generator).
Ter controle wordt de input ook direct ingevoerd in de routine Smul, die het kwadraat berekent volgens de klassieke methode. Het resultaat dient gelijk te zijn aan de output van SSA. In de negacyclische variant dienen de resultaten mod *modulusN* aan elkaar gelijk te zijn.

De Assembler implementatie van het SSA algoritme

Bij de assembler implementatie ligt de nadruk op uitvoeringssnelheid. Deze implementatie heeft de vorm van een assembler routine die toegevoegd wordt aan de Lba functie-library.
De assembler routine accepteert een willekeurig getal in de vorm van een binaire string bits als input. De routine is bedoeld om getallen te kwadrateren die maximaal zo groot zijn dat ze nog net passen in adres-ruimte die Windows in de 32-bits uitvoering biedt. Dat stelt een limiet aan de factor *k*.
Een waarde van 17 voor *k* zou betekenen een aantal (N) van 2^{17} = 131.072 elementen. De minimale grootte van een element zou dan dienen te zijn: N/2 = 65.536 bits of 8.192 bytes; dit om alle verschillende twiddle-factoren te kunnen herbergen. (Er is een manier om met kleinere elementen te volstaan; “the √2 trick”[3]. Daar gaan we in dit document verder niet op in). Totaal zou dat neerkomen op minimaal 131.072*8.192 bytes = één Gigabyte voor de container. Daarnaast is ruimte nodig voor input en output-velden. Dit is teveel voor de

maximaal 2GB adres-ruimte van 32 bits Windows. Daarom is er gekozen voor een waarde van *k* die maximaal 16 is.
In tegenstelling tot de implementaties van Karatsuba en Toom-Cook wordt bij SSA de input niet gekopieerd naar een intern werkveld. Er wordt slechts één werkveld gebruikt en dat is de container. (Bij kwadrateren is er maar één input-getal; dus ook maar één container).
Het aantal betekenisvolle bits van het input-getal wordt bepaald en aan de hand daarvan wordt de factor *k* vastgesteld. Het daarbij behorende aantal elementen en de grootte ervan (uitgedrukt in aantal bits), evenals de modulus, worden berekend en dynamisch wordt de container gecreëerd. De grootte van de container (het beslag dat de container legt op het beschikbare computer-geheugen) is ongeveer viermaal de grootte van de input. De maximale content per element wordt berekend en de container wordt gevuld met de input. De niet gebruikte elementen worden gevuld met nul.
Op het hoogste niveau wordt de acyclische convolutie toegepast zodat slechts de helft van de elementen wordt gevuld. Bij het bepalen van de waarde van *k* en de grootte van de elementen wordt daarmee rekening gehouden.

Voor de Dyadic Stage, waarin de inhoud van de elementen wordt gekwadrateerd, wordt bij kleine elementen gekozen voor het klassieke algoritme (Smul). Voor grotere elementen wordt gekozen voor de Karatsuba routine voor kwadrateren (Skarat). De grens wordt bij 4.096 bits gelegd. Bij elementen van 10.240 bits en groter wordt in de Dyadic Stage het SSA algoritme uitgevoerd. Dit gebeurt **niet** recursief omdat voor de tweede niveau SSA routine gekozen is voor de negacyclische convolutie, die dermate afwijkt van de eerste niveau acyclische convolutie dat het handiger leek om twee volledig onafhankelijke routines te ontwikkelen.
De optimale waarde van *k* voor het tweede niveau SSA algoritme wordt proefondervindelijk vastgesteld bij het testen.
Voor het tweede niveau Dyadic Stage wordt uitsluitend gebruik gemaakt van het klassieke algoritme. Er is dus geen derde niveau SSA algoritme.

Technische aspecten

In de C^{++} routines wordt gebruik gemaakt van de Lba functie-library voor het werken met grote getallen. De Lba functies werken alleen met positieve getallen en nul. Omdat de inhoud van de elementen bij de negacyclische convolutie negatief kan worden, wordt het resultaat-veld geïnitialiseerd met *modulusN*. Het eventueel verminderen van de inhoud van de elementen met modulus (2^n+1) gebeurt nu niet in een element maar rechtstreeks in het resultaat-veld. Door het resultaat-veld te initialiseren met *modulusN* kan dat niet negatief worden. Op het einde van de berekening wordt het resultaat-veld mod *modulusN* genomen zodat de initiële waarde indien nodig wordt gecompenseerd.

Bij de assembler routine worden ook Lba functies gebruikt. De elementen uit de container zijn uitgevoerd als Lba-veld. Een Lba-veld bestaat uit een lengteveld en een aantal limbs (de term limb wordt gewoonlijk gebruikt bij software die rekent met grote getallen. Het geeft aan dat een limb een deel is van een groter geheel; in dit geval is het een deel ter grootte van een computerwoord van 32 bits). Een element bestaat altijd uit een aantal volledige limbs; de minimum grootte is één limb van 32 bits. Het lengteveld is ook een computerwoord van

32 bits, waarin het aantal limbs staat waaruit het Lba-veld bestaat.
De inhoud van de elementen wordt begrensd door de modulus en kan dus niet groter zijn dan of gelijk zijn aan de modulus, die de vorm heeft van 2^n+1. De grootte van een element van de container stellen we op n bits. Deze n is dus deelbaar door 32. De maximale binaire waarde in een computerveld van n bits is 2^n-1. Er moest dus een oplossing gevonden worden voor de (zeldzame) situatie dat de inhoud van een element precies de waarde van 2^n zou hebben.
Het lengteveld van een LBA-veld heeft 32 bits waarvan 30 bits gebruikt worden voor het aangeven van het aantal limbs; (dit aantal kan in de 32-bits uitvoering van Windows nooit meer dan 2^{30} (ongeveer 1 miljard) worden. Er blijven dus twee bits over. De SSA implementatie maakt gebruik van deze twee bits; één bit om aan te geven dat de inhoud gelijk moet worden gesteld aan 2^n en één bit wordt gebruikt om aan te geven dat de inhoud gelijk is aan nul.

Op dezelfde wijze als bij de C^{++} routines wordt bij de negacyclische convolutie het resultaat-veld geïnitialiseerd met *modulusN*. Dit om te voorkomen dat het resultaat-veld negatief wordt.

De *Content* per element (dus de beginwaarde waarmee de elementen gevuld kunnen worden bij het begin van de FFT transformatie) wordt altijd afgerond op gehele limbs van 32 bits. Voor zover met de input niet de *Content* van de laatste limb(s) volledig kan worden gevuld, worden deze aangevuld met “0” bits.

Er is een speciale routine ontwikkeld voor de add/sub functie van elementen. Hierbij worden de limbs van de twee elementen limb voor limb in interne computer registers geplaatst, de som en het verschil wordt bepaald en de limbs worden weer teruggeplaatst in de elementen. Dit voorkomt dat er een kopie van een element moet worden gemaakt (zoals dat wél in het C^{++} voorbeeld gebeurt) en dat naderhand die (bewerkte) kopie weer moet worden teruggeplaatst. In deze speciale routine moet wel een goede administratie worden bijgehouden van de eventuele carries, zowel bij het optellen als bij het aftrekken.

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.
In een binaire computer gaat 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 uit worden geschoven (uit de n bits van de modulus) vormen het quotiënt en de bits die overblijven vormen de rest, nadat van deze rest het quotiënt is afgetrokken[2].

Een andere speciale routine is de Mul/Mod routine. Deze zorgt voor de vermenigvuldiging van de waarde van een element met de twiddle-factor (mod m). Aangezien alle twiddle-factoren machten van twee zijn kunnen de vermenigvuldigingen worden uitgevoerd als een shift naar links over een of meerdere bits, afhankelijk van de twiddle-factor (de twiddle-factor vertegenwoordigt het aantal bits dat naar links verschoven dient te worden). De daarop volgende bewerking mod (2^n+1) gaat zoals hiervoor is beschreven.

Na de inverse FFT moeten de resultaten gedeeld worden door 2^k mod (2^n+1). Deze deling door 2^k kan worden omgezet in een vermenigvuldiging met 2^{2n-k} mod (2^n+1)[2]. Vermenigvuldigen met een macht van twee kan door een shift-bewerking naar links.
Bij de negacyclische convolutie kan het delen door 2^k gecombineerd worden met het delen door de wegingsfactor. Het omzetten in een vermenigvuldiging gebeurt dan door de exponent van twee op *2n-k-w* te zetten, waarbij *w* de wegingsfactor is.

Testen

Doel van het testen

Doel van het testen is om te controleren of de SSA implementatie goed werkt en de juiste resultaten oplevert. Voorts om vast te stellen wat bij elke input-grootte een optimale waarde van k is. Ten slotte wordt voor de snelheid van vermenigvuldigen een vergelijking gemaakt tussen de vier methoden: De Klassieke methode; de Karatsuba methode; de Toom-3 methode en het SSA algoritme. Van al deze methoden zijn versies specifiek voor kwadrateren beschikbaar.
Een verder vraagpunt is wat het beste algoritme voor de “Dyadic Stage” is. Dit zal worden onderzocht.

De C^{++} implementatie is gebruikt om er de (deel-)resultaten van de assembler implementatie mee te vergelijken en wordt niet verder bij het testen betrokken.

Testmethode

Voor het testen is een desktop computer gebruikt met 4 Gigabyte geheugen werkend onder Windows 7. Er is een testprogramma ontwikkeld dat een input-getal genereert. Dit input-getal wordt door de resp. routines Klassiek, Karatsuba, Toom en SSA gekwadrateerd. Het resultaat van Klassiek wordt opgeslagen en de resultaten van de andere routines worden met het klassieke resultaat vergeleken. Verder wordt het aantal klok-cycles van de uitvoeringstijd voor elke routine apart gemeten en gerapporteerd in de programma-log.
(Dit testprogramma heet “SSAtest.exe” en kan worden gedownload van Web-pagina's: www.tonjanee.home.xs4all.nl).
Het input-getal wordt gevormd door een van te voren bepaald aantal bits op een bepaalde waarde te zetten.
Na de berekening door de vier routines wordt het aantal (bits) vergroot (vermenigvuldigd met ongeveer $\sqrt{2}$) en wordt een nieuw input getal gevormd waarmee de vier routines opnieuw aan het werk worden gezet.
Dit wordt telkens herhaald totdat het programma eindigt omdat het gevraagde geheugen (dat immers steeds groter wordt naarmate met een groter input-getal wordt gewerkt) niet meer beschikbaar is. Op deze wijze wordt een reeks uitvoeringstijden verkregen van input-getallen waarvan de grootte (in aantallen bits) logaritmisch oploopt.
De input-getallen kunnen op drie verschillende manieren worden gevuld en vormen zo drie aparte testen die elk afzonderlijk kunnen worden geactiveerd. Bij de eerste test wordt het kleinst mogelijke getal gevormd binnen het aantal significante bits. Dit is één binaire 1 met verder alleen binaire nullen. Bij de tweede test wordt het grootst mogelijke getal gevormd; dit bestaat uit alleen maar binaire enen. Voor de derde test wordt het input-getal gevuld met random binaire nullen en enen. Hiervoor wordt een Lba-functie PRNG (Pseudo Random Number Generator) gebruikt. (Zie verderop voor Lba).
Om de doorloop-tijden niet te veel te laten oplopen worden bij oplopende input-grootte de algoritmen voor resp. Klassiek, Karatsuba en Toom-3 uitgeschakeld. Voor de heel grote getallen blijft alleen het SSA algoritme over. Dan is er ook geen controle meer op de juistheid van het resultaat mogelijk. (Als Klassiek is uitgeschakeld, neemt het resultaat van

Karatsuba de rol van vergelijkings-product over. Als Karatsuba is uitgeschakeld, dan neemt Toom-3 deze functie over). Waar tijdens het testen bleek dat het SSA algoritme een onjuist resultaat opleverde, is het testen afgebroken en is de fout in de SSA implementatie opgespoord en verholpen. Van elke test-sessie wordt een programma-log bijgehouden.

Met dit test-programma (met enige aanpassingen) wordt ook de optimale waarde van de factor k vastgesteld bij de verschillende input-grootten.

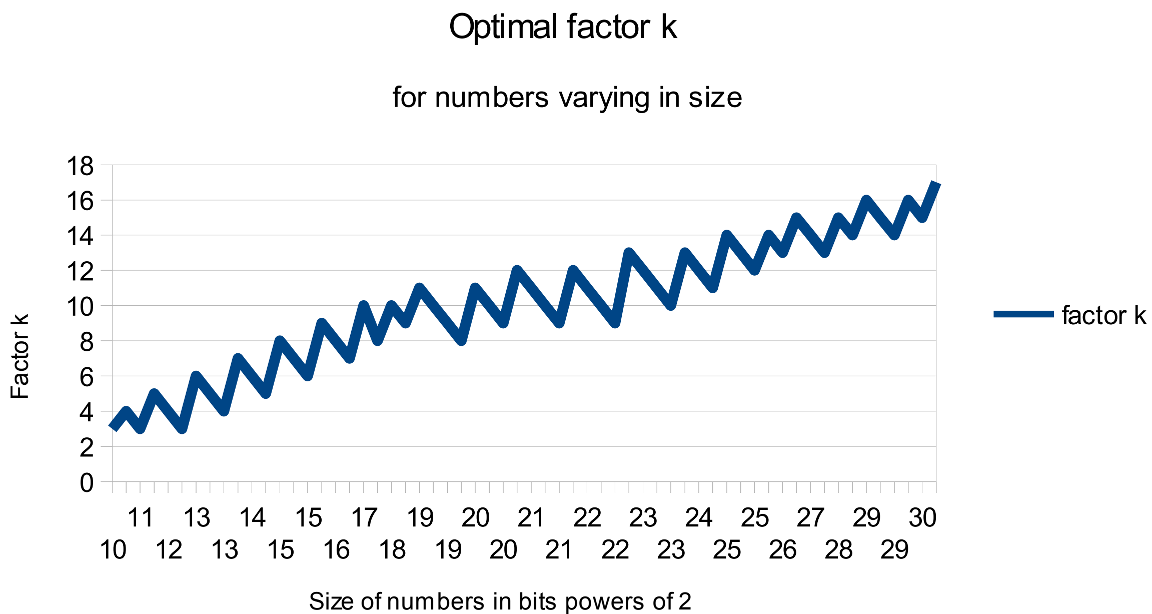
Omdat het de bedoeling is dat dit test-programma op verschillende computers kan draaien wordt vooraf in de programma-log opgenomen de karakteristieken van de processor en het aantal klok-cycles per seconde van de computer waarop het test-programma draait.

Testresultaten

Bij het beoordelen van de testresultaten dient rekening te worden gehouden met het feit dat deze resultaten sterk afhankelijk zijn van de gebruikte hardware en systeem-software.

Als eerste is onderzocht wat de beste waarde voor *k* is bij de SSA routine op het eerste niveau bij verschillende grootten van de input. Hier wordt dus de acyclische convolutie gebruikt. De kleinste waarde voor *k* is 3. In de praktijk zal deze waarde niet worden gebruikt omdat *k=3* alleen optimaal is voor getallen kleiner dan 256 bits. Deze kleine getallen kunnen met andere algoritmes veel sneller worden gekwadrateerd.
Voor getallen groter dan 1,5 Gigabit geldt een waarde voor *k* van 17; dit soort grote getallen kunnen niet op de test-computer worden gekwadrateerd. Het test-programma stopt met een mededeling dat er niet voldoende geheugen is.

De verwachting was dat bij het SSA algoritme de optimale waarde van *k* evenredig zou toenemen met het aantal bits input. Het blijkt echter dat er geen lijnrecht verband bestaat tussen de grootte van het input-getal en *k*, maar dat dit verband eerder het patroon van een zaagtand vertoont.
Bij toenemende input-grootte wordt de optimale waarde van *k* kleiner (!) in stapjes van één tot een bepaald niveau; daarna springt de optimale waarde van *k* drie of vier stappen omhoog tot (meestal) een waarde die hoger ligt dan de vorige hoogste waarde. Dit patroon herhaalt zich telkens. Het volgende diagram geeft deze waarden, die proefondervindelijk zijn vastgesteld, weer.



De optimale waarden voor *k* zijn ingebracht in de assembler routine *SSA*.

Vervolgens is onderzocht wat de optimale algoritmes zijn voor de Dyadic Stage. In de Dyadic Stage worden de elementen ná de forward butterfly gekwadrateerd. Elementen tot en met 4.096 bits kunnen het beste met de klassieke methode worden behandeld. Elementen tot

10.240 bits kunnen het beste met het Karatsuma algoritme worden gekwadrateerd. Het Toom-3 algoritme is hier minder geschikt omdat de optimale input-grootte in aantal bits voor Toom-3 een macht van drie bevat, terwijl de meeste elementen uit een aantal bits bestaan van uitsluitend machten van twee.
Vanaf 10.240 bits kan het beste het SSA algoritme volgens de negacyclische convolutie worden toegepast. Onderzocht is of ook de acyclische convolutie zinvol toepasbaar is. Uit metingen blijkt dat de acyclische convolutie voor elementen vanaf 10.240 bits ongeveer vergelijkbare resultaten geeft als het Karatsuma algoritme, maar dat de negacyclische convolutie circa 40% sneller is dan de acyclische convolutie. Dit was weliswaar te verwachten maar moest toch voor de volledigheid worden uitgetest.
Voor de Dyadic Stage van het tweede niveau wordt alleen maar het klassieke algoritme gebruikt.
Ook de optimale waarde van k bij de Dyadic Stage is vastgesteld en is 6 voor elementen vanaf 10.240 bits. Voor elementen vanaf 65.536 bits is 7 de optimale waarde voor k.
Het gebruik van de negacyclische convolutie en deze waardes voor k zijn ingebracht in de assembler implementatie.

Vervolgens zijn testen uitgevoerd, waarbij steeds grotere input-getallen ter kwadratering zijn aangeboden aan resp. de routines voor klassieke methode; het Karatsuba algoritme; het Toom-3 algoritme en het SSA algoritme. Begonnen werd met een getal van 64 bits; dit aantal bits werd telkens verhoogd met een factor van ongeveer $\sqrt{2}$. Bij elke vermenigvuldiging (kwadratering) werd de uitvoeringstijd gemeten in klok-cycles. Er werd zoveel mogelijk voor gezorgd dat het systeem “rustig” was zodat er geen andere processen tegelijk met de meting actief waren. Dit kon toch niet geheel worden voorkomen. Waar er abnormale afwijkingen in de metingen voorkwamen, zijn deze handmatig gecorrigeerd.

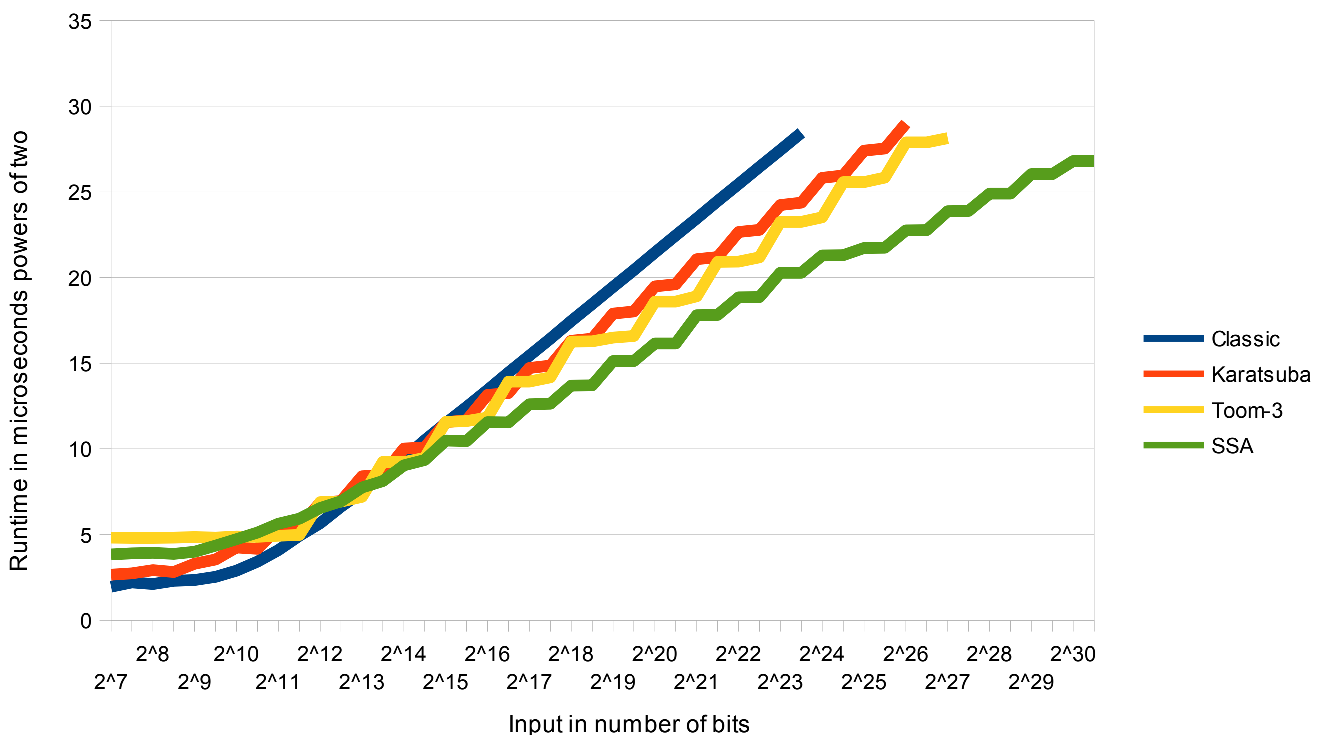
Er zijn meerdere testruns gemaakt. Een met als input een reeks binaire enen en een met als input een reeks random bits. Daarnaast een testrun met als input één 1 met verder slechts nullen. Dit is eigenlijk een onzinnige run omdat de input een pure macht van twee is. Het kwadrateren is een kwestie van het verdubbelen van de exponent van twee. Toch is dit een waardevolle test gebleken om fouten in de routines te ontdekken.

Uit de analyse van de testresultaten blijkt dat het aantal benodigde computer-cycles voor de kwadratering, zoals was te verwachten, oploopt naarmate de input uit meer significante bits bestaat. De mate waarin dat aantal oploopt is sterk afhankelijk van het algoritme dat wordt gebruikt. Maar ook blijkt dat het aantal benodigde cycles afhangt van het soort input.
Uit een controle-run op een andere dan de test-computer blijkt verder dat het onderlinge verschil tussen de vier algoritmes tevens wordt beïnvloed door de gebruikte processor en het operating system. Daarom moeten de conclusies van de metingen met enige voorzichtigheid worden gehanteerd.

De meting van de test-run op de **test-computer** met **random** input beschouwen we als de basis, omdat random input het meest overeen zal komen met de praktijk. De test met de kleinste waarde als input (dus één binaire 1 en verder alleen binaire nullen) heeft overwegend **minder** cycles nodig dan de basis; gemiddeld circa 25 % voor het klassieke algoritme; circa 15% voor Karatsuba en voor Toom-3; en circa 30% voor SSA. De test met de grootste waarde als input (dus allemaal binair 1) heeft overwegend **meer** cycles nodig

dan de basis; gemiddeld 40% voor Klassiek; gemiddeld 4% voor Karatsuba; voor Toom-3 zijn er gemiddeld 4% meer cycles nodig voor de kleinere getallen maar circa 3% minder cycles voor de grotere getallen. Voor SSA is het beeld wisselend, maar bij grotere getallen is er nauwelijks verschil in aantal cycles.

De onderlinge vergelijking tussen de algoritmes wordt bemoeilijkt door deze verschillen in testresultaat tussen de test-runs met verschillende input. Ook test-runs op andere computers leveren niet meer duidelijkheid. Daarom is er voor gekozen om alleen de resultaten van de basis test (random input)verder uit te werken:



De testresultaten zijn in de hiervóór afgebeelde grafiek verwerkt. (De aanduiding 2^7.... enz. onderaan de grafiek staat voor 2^7..enz. en geeft de grootte van het getal aan uitgedrukt in aantal bits.)
De resultaten van de andere tests bevestigen de trend die uit de grafiek blijkt, maar zijn niet verder in de analyse meegenomen.

In de grafiek is de doorlooptijd weergegeven voor het kwadrateren van random getallen met logaritmisch oplopende aantallen bits input voor de vier algoritmen. De doorlooptijd is uitgedrukt in aantallen microseconden waarbij die aantallen zijn weergegeven als machten van twee (dus ook logaritmisch). De horizontale lijn bij 20 geeft aan 2^{20} microseconden; dit is dus ongeveer één seconde. De doorlooptijd is berekend door het aantal computer-cycles te

delen door het aantal cycles per seconde van de desbetreffende processor.

We zien hier dat de klassieke methode tot ongeveer 4.000 bits de snelste resultaten geeft. Vanaf circa 11.000 bits is het SSA algoritme het snelst (dat geldt voor alle drie soorten input). Tussen 4.000 en 11.000 bits ontlopen de algoritmes elkaar niet veel als het om snelheid gaat. Bij oplopende aantallen bits tussen de 4 duizend en 11 duizend wisselen de algoritmes elkaar af als het er om gaat welke de snelste is.

Conclusies

De snelheid (in aantal computer-cycles) waarmee een getal kan worden gekwadrateerd is niet alleen afhankelijk van de grootte van dat getal en het gebruikte algoritme, maar ook van de samenstelling van dat getal en de gebruikte computer hardware en software. Daarom zijn de volgende uitspraken niet algemeen geldig, maar zijn ze slechts geldig voor de algoritmes zoals deze hier zijn geïmplementeerd en de computer-omgeving waarin deze algoritmes zijn getest.
Het kwadrateren van kleinere getallen (getallen kleiner dan ongeveer 1.200 decimale cijfers) kan nog steeds het snelst met het klassieke algoritme. Getallen vanaf circa 3.300 decimale cijfers kunnen het snelst met SSA worden gekwadrateerd. Tussen ongeveer 1.200 en 3.300 decimale cijfers ontlopen de vier algoritmen elkaar niet veel als het om snelheid gaat; bij oplopende aantallen bits input wisselen de algoritmes elkaar af als het er om gaat welk de snelste is.

Het kwadraat berekenen van een getal met één miljoen decimale cijfers kan op elke moderne computer draaiend onder een 32-bits Windows operating systeem met de SSA implementatie zoals hier beschreven binnen één seconde gebeuren. Het grootste getal dat met deze implementatie op een computer met 4 Gigabyte geheugen, draaiend onder Windows 7, kan worden gekwadrateerd is een getal van 1,5 Gigabit of iets minder dan 500 miljoen decimale cijfers. De doorlooptijd is dan circa 2 minuten.

Het SSA algoritme (met negacyclische convolutie) is eigenlijk bedoeld om recursief te gebruikten. Met de beschreven SSA implementatie is dat niet echt het geval. Voor het eerste niveau van de implementatie wordt de acyclische convolutie gebruikt omdat deze eenvoudiger is dan de negacyclische convolutie en het volledige kwadraat oplevert.
Op het tweede niveau wordt voor getallen tot ongeveer 10 miljoen decimale cijfers in de Dyadic Stage gebruik gemaakt van het klassieke of het Karatsuba algoritme. Voor getallen groter dan ongeveer 10 miljoen decimale cijfers wordt in de Dyadic Stage het SSA algoritme met de negacyclische convolutie gebruikt. Dit is een aparte routine, die afwijkt van de routine voor het eerste niveau. De acyclische convolutie biedt hier geen voordelen en heeft circa 40% meer computer cycles nodig. Op het derde niveau wordt in de Dyadic Stage alleen maar het Klassieke algoritme gebruikt. Dus geen recursie.

Voor getallen die groter zijn dan 500 miljoen cijfers kan het recursief gebruik van de negacyclische convolutie mogelijk wel een optie zijn. Dit gaat echter de capaciteit van de gebruikte computer-omgeving (hardware en software) te boven; het past ook niet binnen de doelstelling van dit document om dit te onderzoeken.

Er zijn nog veel verdere verbeteringen mogelijk in de beschreven implementaties van het SSA algoritme. Een overgang naar een 64-bits architectuur zal ongetwijfeld een flinke versnelling opleveren. Daarnaast zijn er in de literatuur veel suggesties en opmerkingen te vinden die al dan niet met succes zijn geïmplementeerd in SSA producten[3].

Voorbeeld programma

Hierna staat een voorbeeld programma in C^{++} code, waarin het SSA algoritme is geïmplementeerd. Zowel de acyclische convolutie als de negacyclische convolutie zijn opgenomen. Slechts de relevante code wordt getoond. Er wordt van uitgegaan dat de container is gedefinieerd met voldoende elementen en dat de elementen van voldoende grootte zijn. Er wordt een kwadraat berekend, zodat er slechts één input getal is, en dus ook maar één container. De X in het programma is een tabel met adres-pointers naar de elementen van de container.
Voor beide convoluties wordt in het programma *k* op 5 gesteld, zodat het aantal elementen van de container $2^5 = 32$ is. De grootte van een element wordt op 64 bits gesteld; de gebruikte modulus is $2^{64}+1$. (In werkelijkheid is een element groter dan 64 bits, zodat ook de waarde 2^{64} kan worden opgeborgen).
Het input-getal is een getal van een bepaald aantal bits dat met random data wordt gevuld. Daarvoor wordt de Lba-functie PRNG gebruikt (Pseudo Random Number Generator).

Bij de acyclische convolutie wordt slechts de helft van de elementen bij de start gevuld. De overige elementen worden op nul gesteld (zero-padding). De "start-content" (dit is het maximale aantal bits waarmee een element bij het begin van de berekening mag worden gevuld) wordt op een andere wijze berekend dan hiervoor is aangegeven en is: $(64/2) - k = 32 - k = 27$ bits. Hiermee wordt voorkomen dat op het resultaat van de deling door *N* de modulus moet worden toegepast. Er kan nu worden volstaan met een simpele shift naar rechts. Als input wordt een random getal met 432 bits gebruikt; zodat alle 16 elementen volledig kunnen worden gevuld (16 x 27 bits).

Bij de negacyclische convolutie worden alle elementen gebruikt. De "start-content" wordt berekend zoals is beschreven: (Size-of-element – k) /2 = (64- 5) / 2 = 29. De input bestaat uit een random getal van 928 bits waarmee de 32 elementen volledig worden gevuld (32x29 bits).

In de programma-log wordt het input-getal getoond, en tevens het kwadraat, zoals dat met de functie Smul via de klassieke methode is berekend. Daarna wordt volgens het SSA algoritme het kwadraat berekend en in de programma-log getoond ter vergelijking met het resultaat volgens de klassieke methode.
De berekeningen met de acyclische en de negacyclische convolutie kunnen in het programma afzonderlijk van elkaar worden geactiveerd en zijn in de voorbeeld-code achter elkaar weergegeven.

Er wordt in het voorbeeld-programma in de C^{++} code een aantal functies gebruikt die nodig zijn voor het rekenen met grote getallen. Deze functies behoren tot de Lba architectuur en worden verderop in dit document beschreven.

Example C++ Code

```
//==========================================================================
//        SSA Algorithm - Acyclic convolution
//--------------------------------------------------------------------------
//        Definition of SSA Integers
//--------------------------------------------------------------------------
          int     ReqSize       ; // required number of bits in input
          int     Sel           ; // size of one element in # bits
          int     k             ; // exponent for two for
          int     D             ; // # elements in container
          int     Content       ; // start-content of one element # bits
//--------------------------------------------------------------------------
//        Definition of FFT variables
//--------------------------------------------------------------------------
          void*   p             ; // address-pointers to elements
          void*   q             ; // of container
          int     a             ; // twiddle factor (exponent of 2)
          int     g             ; // root of unity (exponent of 2)
          int     i             ; // i,j and m are indices
          int     j             ; // used in Gentleman Sande and
          int     m             ; // Cooley-Tukey loops
          int     n             ; // number of elements in container
          int     t             ; // exponent for conversion div in mul
//--------------------------------------------------------------------------
//        Preparation of input
//--------------------------------------------------------------------------
          Mes (31,0)            ; // write separation line in log
          Mes (53,0)            ; // write "Action 6" in log
          Mes (59,0)            ; // write "Test SSA C++ acyclic" in log
          ReqSize = 464         ; // requested size of input in bits
          Prng (input)          ; // fill input with random bits
          Nosb (n,input)        ; // number of significant bits in n
          while (n>ReqSize)   {   // as long as n greater than ReqSize
             Shr (input,1)      ; // divide input by two
             Nosb (n,input) ; } ; // and calculate Nosb again
//--------------------------------------------------------------------------
//        Preparation FFT variables (Container)
//--------------------------------------------------------------------------
          k = 5                 ; // exp. of 2 to define # elements
          Sel = 64              ; // size of one element in # bits
          D = 1                 ; // calculate D (# of elements
          for (i=0; i<k; i=i+1)   // in container)
              D = D*2 ;           // D = 2^k
          g = 2*Sel/D           ; // calculate # bits for root of unity
          Load (modulus,0)      ; // set modulus to 0
          Bts  (modulus,Sel)    ; // calculate:
          Inc  (modulus)        ; // modulus = (2^Sel) + 1
          Content = (Sel-k)/2   ; // calculate start-content
          Load (ContLba,0)      ; // calculate: multiplier in
          Bts (ContLba,Content) ; // Lba field for Content
//--------------------------------------------------------------------------
//        Report SSA & FFT variables
//--------------------------------------------------------------------------
          Mes    (31,0)         ; // write separation line via message
          Nosb   (n,input)      ; // determine number of significant
          Mes    (64,n)         ; // bits in input and report
          Mes    (65,k)         ; // report k
          Mes    (66,D)         ; // report # elements in container
          Mes    (69,Sel)       ; // report size of element
          Mes    (70,g)         ; // report root of unity
          Mes    (71,Content)   ; // start-content # bits per element
```

```
          Mes    (31,0)         ; // write separation line via message
          Report (modulus)      ; // report modulus
          Mes    (31,0)         ; // write separation line via message
          Report (ContLba)      ; // report start-content in Lba form
          Mes    (31,0)         ; // write separation line via message
//----------------------------------------------------------------------
//        Calculate square-product in classic way and report
//----------------------------------------------------------------------
          Report (input)                 ; // report value of input
          Mes    (31,0)                  ; // write separation line
          Smul   (RC,SquareClassic,input); // square multiply input
          Report (SquareClassic)         ; // and report result
          Mes    (31,0)                  ; // write separation line
//----------------------------------------------------------------------
//        Load elements of container with input
//----------------------------------------------------------------------
          for (i=0;i<D/2;i=i+1) { // for first half elements of
            p = X[i]            ; // container load address from
            Mov (wrk,input)     ; // address-array X in p; copy input to
            Mod (wrk,ContLba)   ; // workfield, extract content # bits
            Mov (p,wrk)         ; // and copy extract to element p;
            Shr (input,Content) ; // divide input by shifting bits right
               }                ; // Zero-padding:
          for (i=D/2;i<D;i=i+1) { // for second half elements of
            p = X[i]            ; // container load address from
            Load (p,0)          ; // address-array X in p; load zero
               }                ; // in element p
//----------------------------------------------------------------------
//        Forward FFT - Gentleman-Sande
//        See "Prime Numbers" 2d Edition p.480, ISBN-13 978-0387-25282-7
//----------------------------------------------------------------------
          n = D ;                              // n # elements container
          for ( m = n/2 ; m >= 1 ; m = m/2 )   // start with m = half n
              {                                // dividing m by two until
              for ( j = 0; j < m; j = j + 1 )  // m = 1; start with j =0
                  {                            // increasing j by 1 until
                  a = (g*j*n) / (2*m)        ; // j>=m; calculate twiddle
                  for (i=j; i < n; i=i+2*m )   // factor; place indices
                      {                        // to elements in
                         p = X[i] ;            // container in p
                         q = X[i+m] ;          // and q
//
//        AddSub
//
          Mov (wrk,p)             ; // copy element p in workfield
          Add (p,q)               ; // add element q to element p
          Mod (p,modulus)         ; // apply modulus on p
          while (LessThan (wrk,q))  // if workfield (original p) less
                Add (wrk,modulus) ; // than q, add modulus to workfield
          Sub (wrk,q)             ; // subtract q from workfield
          Mov (q,wrk)             ; // copy workfield in q
//
//        MulMod
//
          if (a != 0)           {   // do only if Twiddle-Factor not
            Shl  (q,a)            ; // is zero: multiply q by Twiddle
            Mod  (q,modulus)      ; // Factor (by shifting left) and
                                } ; // apply modulus on q
                         } ; } ;
                                } ; // end of foreward FFT
//        End of Gentleman-Sande FFT
```

```
//-----------------------------------------------------------------------
//         Dyadic stage - Square multiplication of elements
//-----------------------------------------------------------------------
          for (i=0; i<D; i=i+1)   // for every element
          {                       // in container load address of
            p = X[i]         ;    // element from address-array X in p
            Mov  (wrk,p)     ;    // calculate square of
            Smul (RC,p,wrk)  ;    // element p in p
            Mod  (p,modulus) ;    // and apply modulus
          }                  ;    // on element p
//-----------------------------------------------------------------------
//         Inverse FFT - Cooley-Tukey
//         See "Prime Numbers" 2d Edition p.480, ISBN-13 978-0387-25282-7
//-----------------------------------------------------------------------
          for ( m = 1 ; m < n ; m = 2*m )      // start with m = 1,multi-
              {                                // plying m by two until
              for ( j=0; j < m; j=j+1)         // m not < n; strt with j=0
                  {                            // increasing j by 1 until
                  a = (g*j*n) / (2*m) ;        // j not < m; calculate
                  for (i=j; i < n; i=i+2*m)    // twiddle factor;
                      {                        // place addresses to
                         p = X[i] ;            // elements in container
                         q = X[i+m] ;          // in p and q
//
//         MulMod
//
          if (a != 0)             // do only if Twiddle Factor not is
              {                   // zero: multiply q by Twiddle Factor
              Shl  (q,a)        ; // (by shifting left) and apply
              Mod  (q,modulus)  ; // modulus on element q
              }                 ; //
//
//         AddSub
//
          while (LessThan (p,q))  // if p < q, then
                Add (p,modulus) ; // add modulus to p
          Mov (wrk,p) ;           // copy p to workfield
          Add (p,q)   ;           // add q to p
          Mod (p,modulus) ;       // apply modulus to p
          Sub (wrk,q) ;           // subtract q from workfield
          Mov (q,wrk) ;           // copy workfield to q
            } ; } ; } ;           // end of backward FFT
//
//         End of Cooley-Tukey FFT
//-----------------------------------------------------------------------
//         Divide every element by 2^k (mod modulus)
//-----------------------------------------------------------------------
          t = 2*Sel - k         ; // convert division into multiplication
          for (i=0; i<D; i=i+1) { // multiply every element by 2^t
            p = X [i]           ; // by shifting t bits
            Shl (p,t)           ; // to the left
            Mod (p,modulus) ; } ; // apply modulus
          X[D] = X[0]           ; // place index first element
                                  // as last in table
//-----------------------------------------------------------------------
//         Composition of final result and report
//--------------------------------------------------------------------
          Load (SquareSSA,0)         ; // init SquareSSA to zero
          for (i=1; i<=D ; i=i+1)      // process all elements of
          {                            // container, from first to last
             p = X[i]                ; // load address to element in p
```

```
            Shl (SquareSSA,Content) ; // multiply p by 2^Content (by
            Add (SquareSSA,p)       ; // shifting content bits left)
         }                          ; // add element p to result
         Report (SquareSSA)         ; // report result
         Mes (31,0)                 ; // write separation line
//-----------------------------------------------------------------------
//       End of SSA algorithm - acyclic convolution
//=======================================================================
```

```
//=========================================================================
//        SSA Algorithm - negacyclic convolution
//-------------------------------------------------------------------------
//        Definition of SSA Integers
//-------------------------------------------------------------------------
          int     ReqSize       ; // required number of bits in input
          int     Sel           ; // size of one element in # bits
          int     k             ; // exponent for two for
          int     D             ; // # elements in container
          int     Content       ; // start-content of one element # bits
//-------------------------------------------------------------------------
//        Definition of FFT and DWT variables
//-------------------------------------------------------------------------
          void*   p             ; // address-pointers to elements
          void*   q             ; // of container
          int     a             ; // twiddle factor (exponent of 2)
          int     g             ; // root of unity (exponent of 2)
          int     i             ; // i,j and m are indices
          int     j             ; // used in Gentleman Sande and
          int     m             ; // Cooley-Tukey loops
          int     n             ; // number of elements in container
          int     t             ; // exponent for conversion div in mul
          int     Th            ; // (Theta) general weight factor
          int     Wf            ; // weight factor for element
          int     Sn            ; // workfield for sequnce number element
//-------------------------------------------------------------------------
//        Preparation of input
//-------------------------------------------------------------------------
          Mes (31,0)            ; // write separation line in log
          Mes (62,0)            ; // write "Action 7" in log; write
          Mes (82,0)            ; // "Test SSA C++ negacyclic" in log
          ReqSize = 928         ; // requested size of input in bits
          Prng (input)          ; // fill input with random bits
          Nosb (n,input)        ; // number of significant bits in n
          while (n>ReqSize)   {   // as long as n greater than ReqSize
             Shr (input,1)      ; // divide input by two
             Nosb (n,input) ; } ; // and calculate Nosb again
//-------------------------------------------------------------------------
//        Preparation FFT variables (Container)
//-------------------------------------------------------------------------
          k = 5                 ; // exp. of 2 to define # elements
          Sel = 64              ; // size of one element in # bits
          D = 1                 ; // calculate D (# of elements
          for (i=0; i<k; i=i+1)   // in container)
              D = D*2 ;           // D = 2^k
          g = 2*Sel/D           ; // calculate # bits for root of unity
          Content = (Sel-k)/2   ; // calculate start-content
          Load (ContLba,0)      ; // calculate: multiplier in
          Bts (ContLba,Content) ; // Lba field for Content
          n = Content*D         ; // calculate capacity container in bits
          Load (modulusN,0)     ; // set modulusN to 0
          Bts  (modulusN,n)     ; // calculate modulusN :
          Inc  (modulusN)       ; // modulusN = (2^n + 1)
          Load (modulus,0)      ; // set modulus for element to 0
          Bts  (modulus,Sel)    ; // calculate modulus for element
          Inc  (modulus)        ; // modulus = (2^Sel + 1)
//-------------------------------------------------------------------------
//        Report SSA & FFT variables
//-------------------------------------------------------------------------
          Mes    (31,0)         ; // write separation line via message
          Nosb   (n,input)      ; // determine number of significant
```

```
          Mes    (64,n)         ; // bits in input and report
          Mes    (65,k)         ; // report k
          Mes    (66,D)         ; // report # elements in container
          Mes    (69,Sel)       ; // report size of element
          Mes    (70,g)         ; // report root of unity
          Mes    (71,Content)   ; // start-content # bits per element
          Mes    (31,0)         ; // write separation line
          Report (modulus)      ; // report modulus
          Mes    (31,0)         ; // write separation line
          Report (modulusN)     ; // report modulusN
          Mes    (31,0)         ; // write separation line
          Report (ContLba)      ; // report start-content of element
          Mes    (31,0)         ; // in Lba form; write separation line
//---------------------------------------------------------------------------
//        Calculate square-product in classic way and report
//---------------------------------------------------------------------------
          Report (input)                 ; // report value of input
          Mes    (31,0)                  ; // write separation line
          Smul   (RC,SquareClassic,input); // square multiply input
          Report (SquareClassic)         ; // and report result
          Mes    (31,0)                  ; // write separation line
//---------------------------------------------------------------------------
//        Load elements of container with input
//---------------------------------------------------------------------------
          for (i=0;i<D;i=i+1) {   // for all elements of
            p = X[i]            ; // container load address from
            Mov (wrk,input)     ; // address-array X in p; copy input to
            Mod (wrk,ContLba)   ; // workfield, extract content # bits
            Mov (p,wrk)         ; // and copy extract to element p;
            Shr (input,Content) ; // divide input by shifting bits right
               }                ; //
//---------------------------------------------------------------------------
//        Apply weight factor
//---------------------------------------------------------------------------
          Th = Sel/D          ; // calculate general weight factor
          for (i=0;i<D;i=i+1) { // for all elements of container
             Wf = i*Th        ; // calculate weight factor for element
             if (Wf != 0)   {   // do only if weight Factor not zero
             p = X[i]         ; // load address from address array
             Shl  (p,Wf)      ; // multiply p by Weight factor
             Mod  (p,modulus) ; // (by shifting left) and
                      } ; } ; // apply modulus on p
//---------------------------------------------------------------------------
//        Forward FFT - Gentleman-Sande
//        See "Prime Numbers" 2d Edition p.480, ISBN-13 978-0387-25282-7
//---------------------------------------------------------------------------
          n = D ;                              // n # elements container
          for ( m = n/2 ; m >= 1 ; m = m/2 )   // start with m = half n
              {                                // dividing m by two until
              for ( j = 0; j < m; j = j + 1 )  // m = 1; start with j =0
                  {                            // increasing j by 1 until
                  a = (g*j*n) / (2*m)        ; // j>=m; calculate twiddle
                  for (i=j; i < n; i=i+2*m )   // factor; place indices
                      {                        // to elements in
                         p = X[i] ;            // container in p
                         q = X[i+m] ;          // and q
//
//        AddSub
//
          Mov (wrk,p)             ; // copy element p in workfield
          Add (p,q)               ; // add element q to element p
```

```
          Mod (p,modulus)          ; // apply modulus on p
          while (LessThan (wrk,q))  // if workfield (original p) less
                Add (wrk,modulus) ; // than q, add modulus to workfield
          Sub (wrk,q)              ; // subtract q from workfield
          Mov (q,wrk)              ; // copy workfield in q
//
//        MulMod
//
          if (a != 0)          {    // do only if Twiddle-Factor not
            Shl  (q,a)             ; // is zero: multiply q by Twiddle
            Mod  (q,modulus)       ; // factor (by shifting left) and
                               } ; // apply modulus on q
                      } ; } ; } ; // end of forward FFT

//        End of Gentleman-Sande FFT
//--------------------------------------------------------------------------
//        Dyadic Stage - Square multiplication of elements
//--------------------------------------------------------------------------
          for (i=0; i<D; i=i+1)   // for every element
          {                       // in container load address of
            p = X[i]        ;     // element from address-array X in p
            Mov  (wrk,p)    ;     // calculate square of
            Smul (RC,p,wrk) ;     // element p in p
            Mod  (p,modulus) ;    // and apply modulus
          }                 ;     // on element p
//--------------------------------------------------------------------------
//        Inverse FFT - Cooley-Tukey
//        See "Prime Numbers" 2d Edition p.480, ISBN-13 978-0387-25282-7
//--------------------------------------------------------------------------
          for ( m = 1 ; m < n ; m = 2*m )      // start with m = 1,multi-
              {                                // plying m by two until
              for ( j=0; j < m; j=j+1)         // m not < n; strt with j=0
                  {                            // increasing j by 1 until
                  a = (g*j*n) / (2*m) ;        // jnot < m; calculate
                  for (i=j; i < n; i=i+2*m)    // twiddle factor;
                      {                        // place addresses to
                         p = X[i] ;            // elements in container
                         q = X[i+m] ;          // in p and q
//
//        MulMod
//
          if (a != 0)             // do only if Twiddle Factor not is
              {                   // zero: multiply q by Twiddle Factor
              Shl  (q,a)        ; // (by shifting left) and apply
              Mod  (q,modulus)  ; // modulus on element q
              }                 ; //
//
//        AddSub
//
          while (LessThan (p,q))  // if p < q, then
                Add (p,modulus) ; // add modulus to p
          Mov (wrk,p) ;           // copy p to workfield
          Add (p,q)   ;           // add q to p
          Mod (p,modulus) ;       // apply modulus to p
          Sub (wrk,q) ;           // subtract q from workfield
          Mov (q,wrk) ;           // copy workfield to q
            } ; } ; } ;           // end of backward FFT
//
//        End of Cooley-Tukey FFT
```

```
//---------------------------------------------------------------------------
//        Apply inverse weight factor & divide elements by k(mod modulus)
//---------------------------------------------------------------------------
          X[D] = X[0]          ; // place index first element as last in
                                 // in address table; for every element
          for (i=1;i<=D;i=i+1)  { // of container from 1 to D convert
            t = 2*Sel-(D-i)*Th-k; // division in multiplication by calcu-
            p = X [i]          ; // lating number of bits to shift
            Shl (p,t)          ; // to the left
            Mod (p,modulus) ; } ; // apply modulus
//---------------------------------------------------------------------------
//        Composition of final result
//---------------------------------------------------------------------------
          t = 2*Content              ; // prepare integer with 2*Content
          Mov (SquareSSA,modulusN)   ; // init SSA result with modulusN
          for (i=1; i<=D ; i=i+1)      // process all elements of
          {                            // container, from first to last
            p = X[i]                 ; // load address to element in p
            Shl (SquareSSA,Content)  ; // multiply result by 2^Content
            Add (SquareSSA,p)        ; // add p to result
            Sn=(D+1-i)               ; // calculate sequence number of
            Load (wrk2,Sn)           ; // element; load number in wrk
            Shl (wrk2,t)             ; // by shifting t bits left
            if (LessThan (wrk2,p)) {   // if p >= wrk, then correction:
              Sub (SquareSSA,modulus); // subtract modulus
                               } ; } ; // from result
//---------------------------------------------------------------------------
          Report (SquareSSA)         ; // report Square SSA without
          Mes (31,0)                 ; // modulusN
//------------------------------------------------------------------------
//        Calculate final SSA product mod modulusN and report
//------------------------------------------------------------------------
          Mov (SquareSSA_mod_modulusN,SquareSSA) ; // copy SquareSSA
          Mod (SquareSSA_mod_modulusN,modulusN)  ; // apply modulusN
          Report (SquareSSA_mod_modulusN)        ; // report SquareSSA
          Mes    (31,0)                          ; // write separation
//------------------------------------------------------------------------
//        End of SSA algorithm - negacyclic convolution
//------------------------------------------------------------------------
//        To check result of SSA product;
//        calculate Classic product mod modulusN and report
//------------------------------------------------------------------------
          Mov (SquareClassic_mod_modulusN,SquareClassic) ; // apply
          Mod (SquareClassic_mod_modulusN,modulusN)  ; // modulusN on
          Report (SquareClassic_mod_modulusN)        ; // classic
          Mes    (31,0)                              ; // write separ.
//========================================================================
```

Program Log

```
17:04:51 Storage logging enabeled
17:04:51 Disk logging enabeled
17:04:51 Set language to English
17:04:51 -------------------------------------------------------------------
17:04:51 Processor information:
17:04:51    Vendor ID =   GenuineIntel
17:04:51    Processor Type = Original processor
17:04:51    Processor Family =  06
17:04:51    Processor Model  =  05
17:04:51    Stepping ID =   05
17:04:53 Number_of_cycles_per_second =                        3,186,946,067
17:04:53 -------------------------------------------------------------------
17:04:53 SSA Test Program Version 10.0
17:04:53 Date                          01/18/2014
17:04:53 -------------------------------------------------------------------
17:04:56 Wijzig taal in Nederlands
17:05:01 -------------------------------------------------------------------
17:05:01 Actie 6:
17:05:01 Test C++ routine; aantal bits input:                                 0
17:05:01 -------------------------------------------------------------------
17:05:01 Aantal bits input:                                                 464
17:05:01 k =                                                                  5
17:05:01 D (aantal elementen in container)                                   32
17:05:01 Grootte van element (in aantal bits)                                64
17:05:01 Eenheidwortel g als exponent van twee                                4
17:05:01 Startcontent per element (aantal bits)                              29
17:05:01 -------------------------------------------------------------------
17:05:01 modulus =                                   18.446.744.073.709.551.617
17:05:01 -------------------------------------------------------------------
17:05:01 ContLba =                                                  536.870.912
17:05:01 -------------------------------------------------------------------
17:05:01 input =                  33.961.920.717.900.195.209.977.331.036.648
    .033.151.937.678.978.121.422.563.222.395.691.340.415.150.141.126.656.478
    .428.606.663.180.807.501.407.402.957.547.110.588.568.372.587.057.753.207
17:05:01 -------------------------------------------------------------------
17:05:01 SquareClassic =                                      1.153.412.058
    .848.938.510.791.956.919.483.873.977.795.740.438.440.405.115.868.108.827
    .211.821.711.402.143.010.622.552.664.476.490.716.707.480.762.671.824.132
    .760.897.720.734.033.994.089.442.520.523.936.856.487.818.249.969.913.418
    .713.495.134.056.453.067.758.591.706.324.914.557.986.164.905.740.157.095
    .342.131.855.478.579.651.555.610.689.800.034.680.343.608.450.918.784.849
17:05:01 -------------------------------------------------------------------
17:05:01 SquareSSA =                                          1.153.412.058
    .848.938.510.791.956.919.483.873.977.795.740.438.440.405.115.868.108.827
    .211.821.711.402.143.010.622.552.664.476.490.716.707.480.762.671.824.132
    .760.897.720.734.033.994.089.442.520.523.936.856.487.818.249.969.913.418
    .713.495.134.056.453.067.758.591.706.324.914.557.986.164.905.740.157.095
    .342.131.855.478.579.651.555.610.689.800.034.680.343.608.450.918.784.849
17:05:01 -------------------------------------------------------------------
17:05:05 -------------------------------------------------------------------
17:05:05 Actie 7:
17:05:05 Test C++ negacyclische convolutie
17:05:05 -------------------------------------------------------------------
17:05:05 Aantal bits input:                                                 928
17:05:05 k =                                                                  5
17:05:05 D (aantal elementen in container)                                   32
17:05:05 Grootte van element (in aantal bits)                                64
17:05:05 Eenheidwortel g als exponent van twee                                4
17:05:05 Startcontent per element (aantal bits)                              29
```

```
17:05:05 ---------------------------------------------------------------
17:05:05 modulus =                               18.446.744.073.709.551.617
17:05:05 ---------------------------------------------------------------
17:05:05 modulusN =                                           2.269.007.733
    .883.335.972.287.082.669.296.112.915.239.349.672.942.191.252.221.331.572
    .442.536.403.137.824.056.312.817.862.695.551.072.066.953.619.064.625.508
    .194.663.368.599.769.448.406.663.254.670.871.573.830.845.597.595.897.613
    .333.042.429.214.224.697.474.472.410.882.236.254.024.057.110.212.260.250
    .671.521.235.807.709.272.244.389.361.641.091.086.035.023.229.622.419.457
17:05:05 ---------------------------------------------------------------
17:05:05 ContLba =                                              536.870.912
17:05:05 ---------------------------------------------------------------
17:05:05 input =                                              1.418.100.242
    .970.469.681.178.744.644.249.407.809.958.015.644.132.632.345.114.188.466
    .988.044.724.878.681.259.113.594.276.734.732.943.923.773.192.063.702.324
    .249.223.009.009.831.446.178.218.181.623.275.567.980.248.957.255.225.213
    .779.407.667.589.172.625.180.590.084.361.470.179.343.313.886.606.563.582
    .663.257.535.340.344.323.224.335.851.241.815.166.874.787.501.529.596.068
17:05:05 ---------------------------------------------------------------
17:05:05 SquareClassic =                          2.011.008.299.112.905.144
    .408.292.652.619.752.524.118.616.200.954.119.104.981.461.410.827.097.013
    .298.218.783.672.351.902.115.316.028.379.514.094.378.731.133.533.589.759
    .648.326.542.752.193.805.220.436.545.362.667.066.537.452.555.340.189.250
    .656.675.814.654.509.135.293.136.617.678.904.108.409.414.615.853.231.195
    .475.713.637.213.472.645.600.853.822.894.370.418.166.411.099.104.691.245
    .175.202.333.705.753.609.299.999.846.676.738.479.195.470.879.005.972.917
    .283.853.709.103.211.675.408.161.243.420.853.833.637.477.867.497.151.070
    .189.650.833.244.276.378.986.740.369.910.161.533.819.035.064.040.520.197
    .674.396.182.743.802.349.471.530.565.511.600.417.959.390.618.539.701.366
    .718.247.940.584.136.725.761.529.963.025.127.619.010.764.131.241.060.624
17:05:05 ---------------------------------------------------------------
17:05:05 SquareSSA =                              5.148.396.096.422.391.593
    .693.210.985.222.689.424.575.005.282.213.296.071.655.104.164.476.572.739
    .631.099.228.161.281.473.378.140.200.867.014.462.091.313.754.559.770.129
    .599.777.987.619.078.360.817.109.726.631.408.021.465.198.022.633.853.290
    .201.484.467.517.254.546.983.876.930.815.720.308.649.330.475.055.891.564
    .430.659.129.037.097.735.228.607.981.843.261.104.371.845.409.978.258.744
    .444.537.941.033.570.657.054.515.471.016.391.176.469.227.335.716.244.017
    .238.619.994.470.233.035.652.180.244.068.814.567.709.947.607.219.566.495
    .776.637.660.070.632.796.324.282.751.345.469.167.969.532.201.896.240.618
    .734.269.813.872.937.049.907.796.514.573.130.530.571.876.286.629.108.069
    .597.324.843.841.111.577.770.749.512.758.738.213.893.996.169.176.284.491
17:05:05 ---------------------------------------------------------------
17:05:05 SquareSSA_mod_modulusN =                             2.267.124.662
    .048.302.703.194.051.137.488.035.038.203.015.059.968.844.935.229.340.712
    .680.731.257.439.442.767.042.437.406.535.824.861.638.931.949.531.891.274
    .061.131.170.670.593.868.545.257.024.555.138.539.528.080.069.287.899.636
    .864.997.930.416.923.223.011.909.118.199.776.114.608.476.638.075.683.902
    .819.690.890.892.304.245.291.501.763.400.161.922.143.474.920.697.138.572
17:05:05 ---------------------------------------------------------------
17:05:05 SquareClassic_mod_modulusN =                         2.267.124.662
    .048.302.703.194.051.137.488.035.038.203.015.059.968.844.935.229.340.712
    .680.731.257.439.442.767.042.437.406.535.824.861.638.931.949.531.891.274
    .061.131.170.670.593.868.545.257.024.555.138.539.528.080.069.287.899.636
    .864.997.930.416.923.223.011.909.118.199.776.114.608.476.638.075.683.902
    .819.690.890.892.304.245.291.501.763.400.161.922.143.474.920.697.138.572
17:05:05 ---------------------------------------------------------------
17:05:21 Berekening afgebroken
17:05:21 Logging beeindigd
```

Lba (Long Binary Architecture)

Lba is een (software-)architectuur om berekeningen met grote getallen uit te voeren. In de computerwereld worden grote getallen meestal BigNum getallen genoemd en berekeningen met BigNum getallen noemt men Arbitrary Precision Arithmetic.
Lba (Long Binary Arithmetic) is een vorm van Arbitrary Precision Arithmetic en werkt alleen met natuurlijke getallen, d.w.z. gehele positieve getallen en nul. Er is wel een functie voor het van elkaar aftrekken van twee Lba-variabelen beschikbaar, maar indien het verschil negatief zou worden, dan is het resultaat onvoorspelbaar.
Lba beschrijft het formaat van Lba-variabelen en voorziet in een library met functies die op deze variabelen kunnen worden toegepast. Deze functies kunnen worden opgeroepen vanuit assemblerprogramma's en C^{++} programma's.
Lba is beschikbaar voor een 32-bitsWindows omgeving werkend op Intel x86 architectuur.
Voor een groot aantal rekenkundige en logische functies, waarvoor een computer-instructie beschikbaar is in de Intel-x86 architectuur, is ook een functie in de Lba architectuur aanwezig. Voorbeelden zijn:
Add, Sub, Mul, Div, Inc, Dec, Bts, Bsf.

In het uitgewerkte voorbeeld van het SSA algoritme in C^{++} code worden de volgende Lba-functies gebruikt:

```
Mov   (p1,p2)     Kopieer de inhoud van p2 naar p1
Load  (p1,int)    Zet p1 op 0 en kopieer int in de 1e limb van p1
Add   (p1,p2)     Tel p2 op bij p1
Sub   (p1,p2)     Trek p2 af van p1
Mod   (p1,p2)     Deel p1 door p2 en plaats de rest in p1
Shr   (p1,int)    Shift p1 het aantal bits in int naar rechts
Shl   (p1,int)    Shift p1 het aantal bits in int naar links
Bts   (p1,int)    Zet een bit in p1 op 1, bitnummer staat in int
Inc   (p1)        (increase) p1 wordt met één verhoogd
Dec   (p1)        (decrease) p1 wordt met één verlaagd
Nosb  (int,p1)    Het aantal significante bits van p1 wordt in int
                  geplaatst
LessThan (p1,p2)  Indien p1 < p2 voer dan de instructies uit die
                  hierna volgen
Smul (RC,p1,p2)   Bereken het kwadraat van p2 en plaats dit in p1.
                  Deze berekening geschiedt op de klassieke methode
Report (p1)       Rapporteer een Lba-variabele in de Program Log
Mes   (n,int)     Schrijf een message in de Program Log, waarbij n
                  het message-nummer is en int een 32 bits integer
                  waarvan de waarde achter aan de message wordt
                  weggeschreven
Prng  (p1)        Pseudo Random Number Generator; p1 wordt gevuld
                  met random binaire ene en nullen.
```

```
p1 en p2 zijn pointers naar Lba variabelen.
int is een 32 bits integer (dword).
RC is de result code; RC = 0 is de normale situatie.
```

References

1. Richard Crandall & Carl Pomerance. *Prime Numbers – A Computational Perspective*. Second Edition, Springer, 2005. Chapter 9: Fast Algorithms for large-integer arithmetic, pages 480, 502, 507.
2. Schönhage-Strassen algorithm (http://en.wikipedia.org/wiki/Schönhage-Strassen_algorithm).
3. Pierrick Gaudry, Alexander Kruppa, Paul Zimmerman. *A GMP-based Implementation of Schönhage-Strassen's Large Integer Multiplication Algorithm* (http://www.loria.fr/~gaudry/publis/issac07.pdf). Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation.
4. Sergey Chernenko. *Fast Fourier transform – FFT* (http://www.librow.com/articles/article-10).
5. *Number-theoretic transform.* (http://en.wikipedia.org/wiki/Number-theoretic_transform)
6. *Grote-O-notatie. (http://nl.wikipedia.org/wiki/Grote-O-notatie)*
7. *Root of unity modulo n. (http://en.wikipedia.org/wiki/Root_of_unity_modulo_n)*
8. *Eenheidswortel. (http://nl.wikipedia.org/wiki/Eenheidswortel)*
9. *Convolutie. (http://nl.wikipedia.org/wiki/Convolutie)*
10. *Karatsuba-algorithm. (http://en.wikipedia.org/wiki/Karatsuba_algorithm)*
11. *Toom-Cook multiplication. (http://en.wikipedia.org/wiki/Toom-Cook_multiplication)*
12. *Divide&Conquer. (http//:en.wikipedia.org/wiki/Divide_and_conquer_algorithms)*
13. Richard Brent and Paul Zimmermann. *Modern Computer Arithmetic.* University Press, Cambridge. Section 1.3.6 Squaring, page 11 and Section 2.3.3 The Schönhage-Strassen algorithm, page 55-57.

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