



Universiteit
Leiden
The Netherlands

Op jacht naar radicalen

Dijk, T. van

Citation

Dijk, T. van. (2011). *Op jacht naar radicalen*.

Version: Not Applicable (or Unknown)

License: [License to inclusion and publication of a Bachelor or Master thesis in the Leiden University Student Repository](#)

Downloaded from: <https://hdl.handle.net/1887/3596702>

Note: To cite this publication please use the final published version (if applicable).

OP JACHT NAAR RADICALEN
EEN ALGORITME VOOR DE ENUMERATIE VAN ABC-DRIETALLEN
THIJS VAN DIJK



BACHELORSRIPTIE
Mathematisch Instituut, Universiteit Leiden
Scriptiebegeleider: dr. Bart de Smit
9 september 2011

ABSTRACT

The ABC conjecture: the final frontier. These are the methods used by the ABC@HOME project. Its continuing mission: to explore strange new Diophantine equations; to seek out new ABC triples; to boldly go where no man has gone before.

INHOUDSOPGAVE

ABSTRACT III

INHOUDSOPGAVE V

1	INLEIDING	1
1.1	Het ABC-vermoeden	1
1.1.1	Radicaal	1
1.1.2	ABC-drietal	2
1.1.3	Kwaliteit	3
1.2	Conventies	4
1.2.1	Ordesymbool van Landau	5
2	RADICALEN GENEREREN	7
2.1	Rekentijd	8
2.1.1	Factorisaties	9
2.2	Kwadraatvrije getallen opschrijven	9
2.2.1	Veelvouden met hetzelfde radicaal	11
3	TRIAL DIVISION	13
4	SUB-BLOKKEN FILTEREN	17
4.1	Workunits	19
5	GEVOLGEN	21
	BIBLIOGRAFIE	23
	APPENDIX	25
A	PROGRAMMACODE	27
A.1	abc_sieve_standalone.cpp	27
A.2	abc_sieve_util.cpp	29
A.3	abc_sieve.cpp	37
A.4	Header files	58
A.4.1	abc_common.h	58
A.4.2	abc_sieve.h	59
A.4.3	abc_sieve_util.h	60

1 | INLEIDING

“ Het ABC-vermoeden (zie onder —red.) is momenteel zo’n beetje de heilige graal van de Diophantische wiskunde. ”

— Bart de Smit, 27 januari 2011

Om een inzicht te krijgen in het eerste staartje van dit vermoeden, is in 2005 het ABC@HOME-project gestart, dat als doel heeft een complete lijst op te stellen van alle ABC-drietallen (zie onder) tot een bepaalde bovengrens.

Hierbij wordt gebruik gemaakt van een computerprogramma dat is ontwikkeld door Hendrik Verhoek, Willem-Jan Palenstijn, en Alyssa Milburn. Dit programma kunnen we middels BOINC gedistribueerd uitvoeren over het internet. Op het moment van schrijven rekenen er zo’n 12.000 computers mee aan dit project.

Onlangs heeft het project een mijlpaal bereikt met het afmaken van de complete lijst tot 10^{18} . Na zo’n gigantische inspanning is het natuurlijk interessant om te weten of het programma dat is gebruikt überhaupt wel correct werkt. Met dit doel is in deze scriptie een beschrijving gemaakt voor een wiskundig publiek van de werking van het programma. De correctheid wordt aannemelijk gemaakt, en er wordt een korte analyse gemaakt van de benodigde rekentijd.

Ter referentie is bovendien in de appendix de complete programmacode bijgevoegd.

Berkeley Open Infrastructure for Network Computing. Zie [5], of sectie 4.1

Zie appendix A.

1.1 HET ABC-VERMOEDEN

1.1.1 Radicaal

DEFINITIE Voor $n \in \mathbb{Z} > 0$ wordt het *radicaal* $\text{rad}(n)$ gegeven door:

$$\text{rad}(n) = \prod_{\substack{p|n \\ p \text{ priem}}} p$$

Als geldt $\text{rad}(n) = n$, dan wordt n ook wel *kwadraatvrij* genoemd.

VOORBEELDEN

- $\text{rad}(1.024) = \text{rad}(1.048.576) = \text{rad}(1.073.741.824) = 2.$
- Voor alle $p, q, r \in \mathbb{Z}_{>0}$ geldt: $\text{rad}(2^p 3^q 5^r) = 30.$
- $210 (= 2 \cdot 3 \cdot 5 \cdot 7)$ is kwadraatvrij.
- Ieder priemgetal is kwadraatvrij.

Voor iedere gehele $n > 0$ is het radicaal van n een deler van n , en daarmee in het bijzonder begrensd:

$$1 \leq \text{rad}(n) \leq n,$$

met gelijkheid links dan en slechts dan als $n = 1$.

Ook is het radicaal in zekere mate multiplicatief: voor $p, q \in \mathbb{Z}_{>0}$ geldt:

$$\text{rad}(pq) \leq \text{rad}(p) \text{rad}(q),$$

met gelijkheid dan en slechts dan als p en q onderling ondeelbaar zijn.

1.1.2 ABC-drietal

DEFINITIE Drie getallen $a, b, c \in \mathbb{Z}$ vormen samen een *ABC-drietal* als aan de volgende eisen wordt voldaan:

- $0 < a < b < c$
- $a + b = c$
- a, b , en c zijn copriem
- $\text{rad}(abc) < c.$

VOORBEELDEN

- Neem $a = 1, b = 8, c = 9$. Eisen 1 t/m 3 zijn snel gecontroleerd, en $\text{rad}(1 \cdot 8 \cdot 9) = 2 \cdot 3 = 6 < 9$.
- Neem $(a, b, c) = (5, 27, 32)$. Er geldt, $\text{rad}(5 \cdot 27 \cdot 32) = 5 \cdot 3 \cdot 2 = 30 < 32$.
- Neem $(a, b, c) = (3, 125, 128)$. Er geldt, $\text{rad}(3 \cdot 125 \cdot 128) = 3 \cdot 5 \cdot 2 = 30 < 128$.

Het eerste ABC-drietal uit dit lijstje maakt deel uit van een rijtje, dat ik in het bewijs van de volgende stelling gebruik.

STELLING Er zijn oneindig veel ABC-drietallen.

BEWIJS Kies $n \in \mathbb{Z}_{>0}$. Omdat $3^{2n} \equiv 1 \pmod{8}$, geldt $8 \mid (3^{2n} - 1)$.

Neem nu $a = 1$, $b = 3^{2n} - 1$, $c = 3^{2n}$. Merk op, a, b, c zijn copriem. We zien, $\text{rad}(c) = 3$, en

$$\text{rad}(b) \leq \text{rad}(8) \text{rad}\left(\frac{b}{8}\right) \leq \frac{b}{4}.$$

Dus geldt,

$$\text{rad}(abc) = \text{rad}(a) \text{rad}(b) \text{rad}(c) \leq \frac{3}{4}b < c.$$

Dus (a, b, c) is een ABC-drietel.

Aangezien dit voor iedere $n \in \mathbb{Z}_{>0}$ kan, zijn er dus oneindig veel ABC-drietallen. \square

1.1.3 Kwaliteit

DEFINITIE Voor positieve coprieme a, b, c , met $c > 1$ wordt de *kwaliteit* $q(a, b, c)$ gegeven door:

$$q(a, b, c) = \frac{\log c}{\log \text{rad}(abc)}.$$

We zien meteen een alternatieve definitie voor ABC-drietallen ontstaan: eis 4 kan vervangen worden door de equivalente eis

$$q(a, b, c) > 1.$$

Het zogeheten *ABC-vermoeden*, geformuleerd door Joseph Oesterlé en David Masser doet een uitspraak over deze kwaliteit. Het luidt:

Zie [1].

Voor alle $\varepsilon > 0$ is het aantal ABC-drietallen met kwaliteit minstens $1 + \varepsilon$ eindig.

1.2 CONVENTIES

In deze scriptie is een aantal niet-conventionele notatiekeuzes gemaakt, met name bij het opschrijven van algoritmes. Hier wordt gepoogd die van tevoren samen te vatten.

Allereerst, in ieder bewijs uit het ongerijmde wordt het symbool “ \perp ” gebruikt om het bereiken van een tegenspraak aan te geven.

Met een ‘lijst’ wordt een eindige, indexeerbare verzameling bedoeld, wiens index — tenzij anders vermeld — bij een 0 begint. Typisch is dit in de programmacode geïmplementeerd met een `std::vector<T>`.

Verder wordt in de algoritmes regelmatig de iteratiestap in de vorm “**voor** $i \in I$ **doe** ... **einde voor**” gebruikt, waar I een verzameling is. (Een variant hierop is “**voor** $i \in I : R$ **doe** ... **einde voor**”, waar I een verzameling is, en R een conditie.)

Het zo efficiënt mogelijk enumereren van de verzameling I (of de deelverzameling van I die voldoet aan R) is — tenzij expliciet vermeld — een oefening voor de lezer.

Zie appendix A.

Ter referentie is er overigens altijd nog de c++-implementatie in de appendix.

Als laatste wordt een terminologie voor het “in tweeën knippen” van priemontbindingen gehanteerd.

DEFINITIE Laat $n, p \in \mathbb{Z}_{>0}$. We noemen n ook wel *p-glad* als alle priemdelers van n kleiner dan, of gelijk aan p zijn. De grootste *p-gladde* deler van n heet het *p-gladde deel* van n .

Omgekeerd heet n ook wel *p-ruw* als alle priemdelers van n strikt groter zijn dan p . De grootste *p-ruwe* deler van n heet dan het *p-ruwe deel* van n .

VOORBEELDEN

- Het getal 6 is 5-glad, 4-glad, 3-glad, maar niet 2-glad.
- De getallen 2, 16, 128, en 1.048.576 zijn allen 3-glad; 125 is 3-ruw.
- Het 3-gladde deel van 210 ($= 2 \cdot 3 \cdot 5 \cdot 7$) is $2 \cdot 3$; het 3-ruwe deel is $5 \cdot 7$.
- Het 7-ruwe deel van 5 is 1, evenals het 7-ruwe deel van 7.
- Ieder positief geheel getal is 1-ruw.
- Het getal 77 is noch 9-ruw, noch 9-glad. Het enige getal dat zowel 9-ruw als 9-glad is, is 1.

1.2.1 Ordesymbool van Landau

Bij de analyses van de rekestijd en het geheugengebruik wordt het ordesymbool van Landau gebruikt om afschattingen aan te geven. Dit is een veelgebruikte methode, maar ter volledigheid volgt hier de definitie.

DEFINITIE Laat $f(x)$ en $g(x)$ twee functies op \mathbb{R} zijn. We zeggen dat f *van orde* g (notatie: $f(x) = \mathcal{O}(g(x))$) is, als er een $k \in \mathbb{R}$ bestaat, zodanig dat

$$\forall t > 0 : |f(t)| < k \cdot g(t).$$

De epsilon heeft binnen een ordeteken (althans, in deze scriptie) een speciale betekenis. Als $h(x, \varepsilon)$ een \mathbb{R} -waardige functie is, dan wil de opmerking " $f(x) = \mathcal{O}(h)$ " zeggen,

$$\forall \varepsilon > 0 : f(x) = \mathcal{O}(h(\cdot, \varepsilon)).$$

VOORBEELDEN

- $4.294.967.296 = \mathcal{O}(1)$.
- $f(x) = x^3 + x^2 = \mathcal{O}(x^3)$
- $f(x) = \frac{e^x}{x} = \mathcal{O}(e^x)$.
- $f(x) = x \log x = \mathcal{O}(x^{1+\varepsilon})$.

Bij het maken van een rekestijdanalyse wordt meestal aangenomen dat er een functie $f(x)$ bestaat die de grootte x van de opdracht omzet naar een of andere tijdseenheid. Als f van orde g is, zeggen we ook wel dat het algoritme *van complexiteit (in tijd)* $\mathcal{O}(g)$ is. Het vaststellen van zo'n functie is typisch gekkenwerk, maar dikwijls is het best mogelijk om de rekestijd tot op de dichtstbijzijnde orde af te schatten.

Het gebruik van ordenotaties kan ook inzicht bieden in hoe de rekestijden van verschillende opdrachten zich verhouden.

Stel, een algoritme van complexiteit (in tijd) $\mathcal{O}(N^2)$ kan onder optimale omstandigheden op een gestandaardiseerde computer een opdracht van grootte M in één uur verwerken.

Dankzij de sterk versimpelde ordenotatie is het makkelijk om in te zien dat een opdracht van grootte $2M$ hooguit vier uur gaat duren. Evenzo, als het algoritme efficiënter geïmplementeerd wordt zodat het twee keer zo snel is, kan de computer in datzelfde uur een opdracht van grootte minstens $\sqrt{2}M$ verwerken.

2 | RADICALEN GENEREREN

We hebben ons de taak voorgezet om een complete lijst te maken van alle ABC-drietallen a, b, c met $c < N$, waar N een van tevoren afgesproken bovengrens is.

Een voor de hand liggende aanpak voor het genereren van een lijst tot een zekere bovengrens N is natuurlijk de volgende.

Algoritme 1 Een voor de hand liggende manier om ABC-drietallen op te schrijven

```
1: voor  $a \in \{1, \dots, \frac{1}{2}N\}$  doe
2:   voor  $b \in \{a + 1, \dots, N - a\}$  doe
3:     Zet  $c := a + b$ .
4:     als  $a, b, c$  copriem en  $\text{rad}(abc) < c$  dan
5:       print  $(a, b, c)$ 
6:     einde als
7:   einde voor
8: einde voor
```

Algoritme 1 is volkomen functioneel, maar vanwege de uitermate hoge rekentijd niet erg praktisch. Voor a en voor b zijn er immers $\mathcal{O}(N)$ mogelijkheden, en men moet vervolgens telkens op regel 4 een factorisatie uitvoeren van abc ($= \mathcal{O}(N^3)$). De factorisatie van een getal van orde $\mathcal{O}(N^3)$ kan worden uitgevoerd met met complexiteit (in tijd) $\mathcal{O}(N^{\frac{3}{2}})$, dus de rekentijd van algoritme is $\mathcal{O}(N^{3\frac{1}{2}})$.

Algoritme 1 laat dus duidelijk ruimte over voor verbetering. Hiertoe definiëren we eerst een (zwakkere) variant op ABC-drietallen.

DEFINITIE Coprieme getallen $x, y, z \in \mathbb{Z}_{>0}$ vormen samen een *XYZ-driet* als geldt: $z \in \{x + y, |x - y|\}$, en

$$\text{rad}(x) < \text{rad}(y) < \text{rad}(z).$$

Merk op dat een ABC-driet op unieke wijze te herordenen is naar een XYZ-driet. Voor ieder omgeschreven ABC-driet geldt vervolgens

$$\text{rad}(y)^2 < \text{rad}(y) \text{rad}(z)$$

en

$$\text{rad}(x) \text{rad}(y)^2 < \text{rad}(xyz) < c (= \max\{x, y, z\}) < N. \quad (2.1)$$

Toen het project werd gestart hanteerden we $N = 10^{18}$, maar onlangs [6] is besloten het project te verlengen. Inmiddels geldt $N = 2^{63}$.

Zie sectie 2.1.1

Deze ongelijkheden komen goed van pas bij de constructie van een verbeterd algoritme (algoritme 2), ontwikkeld door Hendrik Verhoek, Willem-Jan Palenstijn en Alyssa Milburn, geïnspireerd door Hendrik Lenstra en Bart de Smit.

Algoritme 2 Een verbeterd algoritme

```

1: voor  $r_x \in \{0, \dots, \sqrt[3]{N}\}$ , met  $r_x$  kwadraatvrij doe
2:   voor  $r_y \in \{r_x, \dots, \sqrt{\frac{N}{r_x}}\}$ , met  $r_y$  kwadraatvrij, en
       $\text{ggd}(r_x, r_y) = 1$  doe
3:     voor  $x \in \{0, \dots, N\}$ , met  $\text{rad}(x) = r_x$  doe
4:       voor  $y \in \{0, \dots, N\}$ , met  $\text{rad}(y) = r_y$  doe
5:         voor  $z \in \{x + y, |x - y|\}$  doe
6:           Controleer of je een ABC-drietal krijgt als je
              $(x, y, z)$  sorteert.
7:         einde voor
8:       einde voor
9:     einde voor
10:   einde voor
11: einde voor

```

Om de correctheid van algoritme 2 in te zien is het voldoende op te merken dat algoritme 2 ieder XYZ-drietal controleert, en dus in het bijzonder ieder ABC-drietal.

Algoritme 2 heeft een aantal niet-evidente stappen. Allereerst is het in regels 1 en 2 noodzakelijk om een lijst kwadraatvrije getallen tussen bepaalde grenzen te kunnen maken. Omgekeerd is het ook nodig om bij een gegeven kwadraatvrije r de lijst getallen (tot N) te produceren wiens radicaal gelijk is aan r . Dit wordt in sectie 2.2 resp. 2.2.1 behandeld.

Tenslotte is er de ABC-heidstest op regel 6, welke in hoofdstukken 3 en 4 uitvoerig wordt besproken.

De grootste truc van dit verbeterde algoritme is dat we niet langer blindelings a en b itereren, maar slim gebruik maken van de in formule (2.1) gevonden ongelijkheden. In sectie 2.1 wordt hier dieper op ingegaan.

2.1 REKENTIJD

Om een inzicht te krijgen van de rekentijd van algoritme 2, hebben we de (overigens niet gepubliceerde) stelling van Granville nodig. Die luidt als volgt.

STELLING (GRANVILLE) Zij $N > 1$. Dan geldt:

$$\# \left\{ (x, y) \mid \begin{array}{l} x, y \text{ copriem} \\ \text{rad}(x) \text{rad}(y)^2 < N \end{array} \right\} = \mathcal{O}(N^{\frac{2}{3} + \epsilon}).$$

Deze stelling is een speciaal geval van de *methode van Rankin*. Het bewijs van deze stelling wordt in dit artikel niet gegeven.

Zie [2, p. 117].

We zien meteen dat deze stelling boekdelen spreekt over de grootte van de zoekruimte van algoritme 2: aangezien er maar 2 mogelijkheden zijn voor de keuze van z , is de zoekruimte dus van orde $\mathcal{O}\left(N^{\frac{2}{3}+\varepsilon}\right)$.

2.1.1 Factorisaties

Met de orde van de zoekruimte is het verhaal echter nog niet af. Immers moet er voor ieder gevonden drietal het radicaal van xyz berekend worden, waar nog altijd een factorisatie van $z (= \mathcal{O}(N))$ voor nodig is.

Van de naïeve manier om dat te doen (door door elke priem tot \sqrt{z} zo vaak mogelijk te delen) zullen we de rekentijd afschatten. (Er zijn methodes die in sommige gevallen beter werken, maar die worden in dit project niet gebruikt.)

Allereerst merken we op dat voor $n > 1$ geldt

Zie [2, p. 10].

$$\pi(n) := \#\{p < n : p \text{ priem}\} = \mathcal{O}\left(\frac{n}{\log n}\right).$$

We kunnen vervolgens de rekentijd die nodig is om z te factoriseren afschatten met een integraal, door op te merken dat door elke priem p hoogstens $\log_p z$ keer gedeeld kan worden:

$$\begin{aligned} \int_1^{\pi(\sqrt{z})} \frac{\log z}{\log p} dp &< \log z \int_1^{\pi(\sqrt{z})} 1 dp = \log z \cdot \pi(\sqrt{z}) \\ &= \log z \cdot \mathcal{O}\left(\frac{\sqrt{z}}{\log \sqrt{z}}\right) = \mathcal{O}\left(\log z \cdot \frac{z^{\frac{1}{2}}}{\frac{1}{2} \log z}\right) = \mathcal{O}\left(z^{\frac{1}{2}}\right). \end{aligned}$$

De complexiteit (in tijd) van algoritme 2 is dus $\mathcal{O}\left(N^{\frac{2}{3}+\frac{1}{2}+\varepsilon}\right)$.

2.2 KWADRAATVRIJE GETALLEN OPSCHRIJVEN

Om een lijst van kwadraatvrije getallen in een interval $[m, M]$ te genereren, wordt een variant op de zeef van Eratosthenes gebruikt.

Algoritme 3 (pagina 10) ‘streept’ alle veelvouden van kwadraten ‘weg’, dus blijven alleen de kwadraatvrije getallen over.

Met dit algoritme kunnen we dus makkelijk (in weinig rekenstappen) alle kwadraatvrije getallen in een interval opschrijven. We zijn er echter nog niet. Als we alvast vooruitblikken op sectie 2.2.1, zullen we zien dat het wenselijk is om van elk kwadraatvrij getal de priemontbinding te onthouden. Deze (overigens weinig ingrijpende) toevoeging is beschreven in algoritme 4 (pagina 10).

Ter volledigheid, alleen kwadraten van priemgetallen. Dit is echter voldoende.

Algoritme 3 Een algoritme om radicalen te genereren in een interval $[m, M)$.

Invoer: Een interval $[m, M)$; de lijst P van alle priemgetallen kleiner dan \sqrt{M} .

Uitvoer: Alle kwadraatvrije $r \in [m, M)$.

```
1: Laat  $L := (1, 1, \dots, 1)$  een lijst van  $M - m$  enen zijn.
2: Laat  $I := \{0, \dots, M - m - 1\}$ .
3: voor  $p \in P$  doe
4:   voor  $i \in I$  met  $p^2 \mid (m + i)$  doe
5:     Zet  $L_i := 0$ .
6:   einde voor
7: einde voor
8: resultaat  $\{(m + i) \mid i \in I : L_i \neq 0\}$ 
```

Algoritme 4 Een algoritme om radicalen te genereren in een interval $[m, M)$.

Invoer: Een interval $[m, M)$; de lijst P van alle priemgetallen kleiner dan \sqrt{M} .

Uitvoer: Een lijst van alle tupels (r, R) , waar $m \leq r < M$ kwadraatvrij is, en R de verzameling is van alle priemdelers van r .

Implementatie: de functie *sieveradicals_interval* in het bestand *abc_sieve_util.cpp*. (Pagina 31.)

```
1: Laat  $L := (1, 1, \dots, 1)$  een lijst van  $M - m$  enen zijn, en
    $R := (\emptyset, \emptyset, \dots, \emptyset)$  een lijst van  $M - m$  (vooralsnog lege) verzamelingen.
2: Laat  $I := \{0, \dots, M - m - 1\}$ .
3: voor  $p \in P$  doe
4:   voor  $i \in I$  met  $p \mid (m + i)$  doe
5:     als  $L_i \neq 0$  dan
6:       Zet  $L_i := p \cdot L_i$ .
7:       Voeg  $p$  toe aan  $R_i$ .
8:     einde als
9:   einde voor
10:  voor  $i \in I$  met  $p^2 \mid (m + i)$  doe
11:    Zet  $L_i := 0$ .
12:  einde voor
13: einde voor
14: voor  $i \in I$  met  $L_i \neq 0$  doe
15:   als  $L_i \neq m + i$  dan
16:     Voeg  $\frac{m+i}{L_i}$  toe aan  $R_i$ .
17:   einde als
18: einde voor
19: resultaat  $\{(m + i, R_i) \mid i \in I : L_i \neq 0\}$ 
```

Allereerst, het opslaan van alle gevonden priemfactoren in een verzameling op regel 7 spreekt voor zich. Deze methode is alleen nog niet waterdicht: er zullen vast kwadraatvrije getallen zijn in het interval met een priemfactor groter dan \sqrt{M} .

Bij het vinden van de 'laatste' priemdelers gebruiken we de volgende stelling.

STELLING Zij $r, M \in \mathbb{Z}$, met $0 < r < M$ en r kwadraatvrij. Laat L het \sqrt{M} -gladde deel zijn van r . Dan geldt: $r \neq L \Rightarrow \frac{r}{L}$ is priem.

BEWIJS Stel niet. Dan heeft $\frac{r}{L}$ minstens twee priemfactoren, beide groter dan \sqrt{M} . Dus $\frac{r}{L} > M$. Dus $r > M$. ζ

Dus $\frac{r}{L}$ is priem. □

In regel 6 wordt de deler L opgebouwd, en in regel 14 t/m 18 wordt gekeken of er nog een deler mist.

Om de rekestijd van dit algoritme in te zien, merken we op dat het algoritme voor iedere priem $p < \sqrt{M}$ nog $\frac{M}{p}$ stappen moet afleggen. Aangezien

$$\int_1^{\sqrt{M}} \frac{M}{x} dx = M \log \sqrt{M},$$

is de complexiteit van algoritme 4 dus $\mathcal{O}(M \log M)$.

2.2.1 Veelvouden met hetzelfde radicaal

Nu we van elk gegenereerd kwadraatvrij getal r de priemontbinding hebben onthouden, is het vrij eenvoudig om getallen te verzinnen die r als radicaal hebben.

Algoritme 5 Een algoritme dat getallen met een gegeven radicaal genereert.

Invoer: Een kwadraatvrije r ; de verzameling P van alle priemen die r delen. Een bovengrens $M \in \mathbb{Z} > r$ voor de uitvoer.

Uitvoer: Alle $n \in \{1, \dots, M\}$ met $\text{rad}(n) = r$.

Implementatie: de functie *ForEachRad* in het bestand *abc_sieve_util.cpp*. (Pagina 33.)

```

1: Laat  $N := \{r\}$  een lijst zijn. Laat  $i := 0$ .
2: zolang  $i < \#N$  doe
3:   voor  $p \in P$  doe
4:     als  $p \cdot N_i < M$  dan
5:       Plak  $p \cdot N_i$  achteraan  $N$ .
6:     einde als
7:   einde voor
8:   Zet  $i := i + 1$ .
9: einde zolang
10: resultaat  $N$ .
```

Merk op dat de lijst N steeds groter wordt, en het algoritme dus pas stopt als alle voldoende kleine veelvouden zijn geprobeerd.

De opbouw van de c++-implementatie verschilt overigens een klein beetje van algoritme 5, in die zin dat de c++-versie een *lijstje* kwadraatvrije getallen controleert, in plaats van slechts één. Bovendien wordt niet de volledige lijst N teruggegeven, maar wordt er telkens een routine aangeroepen voor ieder gevonden getal. De keuze van die routine is één van de parameters van de functie.

3 | TRIAL DIVISION

De laatste schakel in het algoritme is een feilloze ABC-heidstest. We zagen in hoofdstuk 2 dat een XYZ-drietal (x, y, z) een (op permutatie na) ABC-drietal is dan en slechts dan als geldt

$$\text{rad}(xyz) < \max\{x, y, z\},$$

in andere woorden,

$$\text{rad}(z) < \frac{\max\{x, y, z\}}{\text{rad}(x) \text{rad}(y)}.$$

We hebben dus een goedgedefinieerd interval waar $\text{rad}(z)$ in moet vallen om samen met x en y een ABC-drietal te maken. (We wisten immers nog dat $\text{rad}(y) < \text{rad}(z)$.) De kunst is om dit zo snel mogelijk met zekerheid te kunnen bevestigen of ontkrachten.

Hierbij maken we gebruik van de volgende twee stellingen.

STELLING A Zij $p > 2$. Zij $n \in \mathbb{Z}_{>1}$ p -ruw. Dan geldt: of n is een priemmacht, of $\text{rad}(n) > p^2$.

BEWIJS Van het tegendeel (n is een samengesteld p -ruw getal met een radicaal kleiner dan p^2 , en *geen* priemmacht) is makkelijk in te zien dat het onwaar is.

STELLING B Zij $p > 2$. Zij $n \in \mathbb{Z}_{>1}$ p -ruw, en $n < p^3$. Dan geldt, of n is een priemkwadraat, of n is kwadraatvrij.

BEWIJS Stel niet. Dan zijn er verschillende priemdelers q_1, q_2 van n , groter dan p , met $q_2^2 \mid n$. Dus $n \geq q_1 q_2^2 > p^3$. ζ

Dus n is een priemkwadraat, of n is kwadraatvrij. \square

Deze stellingen komen goed van pas bij algoritme 6. Dit algoritme is in principe een naïeve manier om het radicaal van z te bepalen, maar van de stellingen hierboven krijgen we twee extra stopcondities cadeau.

De c++-implementatie komt vrij sterk overeen met algoritme 6, op twee details na. Ten eerste worden er nergens delingen uitgevoerd: aangezien we met binaire computers werken, is iedere rekenoperatie in principe een operatie op $\mathbb{Z}/2^M\mathbb{Z}$, waar M het aantal bits is (veelal 64). Een belangrijk gevolg daarvan is dat alle priemmen behalve 2 een multiplicatieve inverse hebben. Vermenigvuldiging met die inverse is vele malen efficiënter dan een deling. (Delen door 2 kan overigens snel middels *bitshiften*.)

Ten tweede worden de twee stopcondities niet gecontroleerd voor elke priem in het lijstje, maar slechts om de 16 priemmen.

Zie [7, p. 192],
of [4]

Algoritme 6 Een abc-heidstest.

Invoer: x, y, z , met x, y, z copriem, $z \in \{x + y, |x - y|\}$; $r_x := \text{rad}(x)$, $r_y := \text{rad}(y)$; $M := \frac{\max\{x, y, z\}}{r_y r_x}$; een lijst P van priemgetallen kleiner dan $\sqrt{\max\{x, y, z\}}$.

Uitvoer: x, y, z is al dan niet een ABC-drietal.

Implementatie: de functie `ABCSieve::trial_div` in het bestand `abc_sieve.cpp`. (Pagina 45.)

```
1: Zet  $r := 1, l := z$ .
2: voor  $p \in P$  doe — In volgorde.
3:   als  $p \mid l$  dan
4:     Zet  $l := \frac{z}{p}, r := p \cdot r$ .
5:     zolang  $p \mid l$  doe
6:       Zet  $l := \frac{l}{p}$ .
7:     einde zolang
8:   einde als
9:   als  $r \cdot p^2 > M$  dan
10:    Er is voldaan aan de stopconditie uit stelling A. Rest ons alleen te
    kiezen uit één van de twee gevallen.
11:    Voer een machtstest uit op  $l$ .
12:    als  $\exists j, k \in \mathbb{Z} : l = k^j$  dan
13:      Merk op,  $k$  is priem, dus  $\text{rad}(z) = r \cdot k$ .
14:      resultaat  $(r_y < rk < M)$ .
15:    else
16:      Merk op,  $\text{rad}(z) > r \cdot p^2$ .
17:      resultaat NEEN.
18:    einde als
19:  einde als
20:  als  $p^3 > l$  dan
21:    Er is voldaan aan de stopconditie uit stelling B. Wederom hoeven
    we alleen te kiezen uit één van de twee gevallen.
22:    Voer een kwadraattest uit op  $l$ .
23:    als  $\exists k \in \mathbb{Z} : l = k^2$  dan
24:      Merk op,  $k$  is priem, dus  $\text{rad}(z) = r \cdot k$ .
25:      resultaat  $(r_y < rk < M)$ .
26:    else
27:      Merk op,  $l$  is kwadraatorij, dus  $\text{rad}(z) = r \cdot l$ .
28:      resultaat  $(r_y < rl < M)$ .
29:    einde als
30:  einde als
31: einde voor
```

Dit heeft een belangrijk gevolg voor de machtstest op regel 11, die waarschijnlijk het interessantste gedeelte is van dit hele algoritme.

Deze machtstest is geïmplementeerd in de functie *ABCSieve::rad_pp_gt_maxrad* in het bestand *abc_sieve.cpp* op pagina 43, en werkt iteratief. Eerst wordt gekeken of het invoergetal L een kwadraat is, door (op de Egyptische manier) de wortel te trekken. Als dit lukt, dan wordt de stap herhaald met \sqrt{L} . Dit procédé wordt herhaald met derde-, vijfde-, en zevendemachten.

Voor elfdemachten gebruiken we het feit dat de eerste 17 priemmen al zijn gecontroleerd, en dat L dus 59-ruw is. De kleinste elfdemacht zou dus 61^{11} zijn, en dat is groter dan onze beoogde bovengrens 2^{63} .

Op dezelfde manier worden meer hoge machten afgevangen.

4

SUB-BLOKKEN FILTEREN

De “trial division”-stap in het vorige hoofdstuk is een in rekestijd dure grap. In veel gevallen zal voor lage p het p -gladde deel van $\text{rad}(z)$ de bovengrens al overstijgen.

Het kan daarom de investering best waard zijn om een extra filterstap (of “zeefstap”) te maken voor de laagste paar priemmen.

Deze filterstap, beschreven in algoritme 7, sorteert voor elke lage priem p de potentiële x en y naar hun restklasse modulo p . Vervolgens geldt voor iedere potentiële x en y die elkaars tegengestelde zijn modulo p , dat het radicaal van $x + y$ een factor p heeft.

Analoog geldt voor iedere potentiële x en y die equivalent zijn modulo p , dat het radicaal van $|x - y|$ een factor p heeft.

Op deze manier is het voor een gegeven p makkelijk om het radicaal van het p -gladde deel van iedere mogelijke z te bepalen, zonder daarvoor heel veel “dure” delingen uit te hoeven voeren.

In de beschrijving van algoritme 7 nemen we — overigens zonder beperking der algemeenheid — aan dat in ieder drietal geldt $z = x + y$. De rekenstappen die in het geval $z = |x - y|$ anders verlopen, zijn gemarkeerd met een rode stip (•).

Er zitten twee grote vraagstukken, en één leukigheidje in dit algoritme.

Het leukigheidje zit verborgen in de controle of de twee radicalen copriem zijn, op regel 25. Uiteraard volstaat het hier om het algoritme van Euclides te gebruiken. Echter, in de code wordt een versie gebruikt die toegespitst is op kwadraatvrije getallen. Deze variant voert geen delingen met rest uit, maar maakt geheel gebruik van bitshifts. Dit is de functie *AreRadCoprime* in *abc_sieve_util.cpp* (pagina 137).

Zie [3, p. 646]. Ook wel toegeschreven aan Arjen Lenstra.

Het eerste grote vraagstuk betreft de motivatie voor deze zeefstap. Wanneer is het rendabel om deze stap uit te voeren? Is er een punt waarop deze extra zeefstap meer tijd kost dan het oplevert? Blijft het rendabel om te zeven als we de zoekruimte met nog een orde vergroten?

Dit is, anticlimactisch genoeg, een beter onderwerp voor een volgende publicatie.

De tweede vraag is van een veel praktischer aard: hoe veel geheugenruimte is er nodig om de matrix van regel 2 op te slaan?

Het antwoord vinden we in sectie 2.1: aangezien we de *complete* zoekruimte opslaan, is de geheugencomplexiteit van dezelfde orde: $\mathcal{O}\left(N^{\frac{2}{3}+\varepsilon}\right)$.

Algoritme 7 De ‘zeefstap’ die veel potentiële drietallen filtert.

Invoer: Een lijst P van alle priemgetallen tot een grens \bar{p} ; een lijst N^x met de bijbehorende lijst radicalen R^x zodanig dat voor iedere $i \in \{0, \dots, \#N^x - 1\}$ geldt $\text{rad}(N_i^x) = R_i^x$; analoog de lijsten N^y en R^y .

Uitvoer: Alle potentiële (x, y, z) die een ABC-drietal kunnen vormen. Dat wil zeggen, alle drietallen (x, y, z) met $x \in N^x$, $y \in N^y$, $z = x + y$ •, en waarvan het \bar{p} -vrije deel van $\text{rad}(z)$ kleiner is dan $\frac{\max\{x, y, z\}}{\text{rad}(x)\text{rad}(y)}$.

Implementatie: de functie `ABCSieve::do_block` in het bestand `abc_sieve.cpp`. (Pagina 52.)

```
1: Laat  $I^x := \{0, \dots, \#N^x - 1\}$ , en  $I^y := \{0, \dots, \#N^y - 1\}$ .
2: Laat  $Z$  een  $(\#N^x) \times (\#N^y)$ -matrix zijn.
3: voor  $p \in P$  doe
4:   Laat  $Q := \{\emptyset, \emptyset, \dots, \emptyset\}$  een lijst van  $p$  (lege) lijsten zijn.
5:   voor  $i_x \in I^x$  doe
6:     Noem  $x := N_{i_x}^x$ .
7:     Noem  $r := x \bmod p$ .
8:     als  $r = 0$  dan ga door met de volgende  $i_x$ .
9:     Voeg  $i_x$  toe aan  $Q_r$ .
10:  einde voor
11:  voor  $i_y \in I^y$  doe
12:    Noem  $y := N_{i_y}^y$ .
13:    Noem  $r := y \bmod p$ .
14:    als  $r = 0$  dan ga door met de volgende  $i_y$ .
15:    voor  $i_x \in Q_{p-r}$  • doe
16:      Noem  $x := N_{i_x}^x$ .
17:      Zet  $Z_{i_x, i_y} := pZ_{i_x, i_y}$ .
18:    einde voor
19:  einde voor
20:  einde voor
21:  voor  $i_x, i_y \in I^x \times I^y$  doe
22:    Noem  $x := N_{i_x}^x$ ,  $y := N_{i_y}^y$ .
23:    Noem  $r_x := R_{i_x}^x$ ,  $r_y := R_{i_y}^y$ .
24:    als  $r_x > r_y$  dan ga door met het volgende paar  $i_x, i_y$ .
25:    als  $\text{ggd}(r_x, r_y) \neq 1$  dan ga door met de volgende  $i_x, i_y$ .
26:    als  $r_y < Z_{i_x, i_y} < \frac{x+y}{r_x r_y}$  • dan
27:      print  $(x, y, z)$ 
28:    einde als
29:  einde voor
```

Helaas komt dit met onze nieuwe bovengrens ($N = 2^{63}$) neer op zo'n 128 terabyte. Dit is uiteraard volkomen onacceptabel.

Om het geheugengebruik enigszins binnen de perken te houden, wordt het werk daarom opgedeeld in kleine stukjes, of "sub-blokken". De grootte van deze subblokken hangt af van de hoeveelheid beschikbaar geheugen.

Dit wordt in de code bewerkstelligd door een partitie P^x te maken van N^x , en een partitie P^y van N^y , en voor iedere $(U, V) \in P^x \times P^y$ algoritme 7 aan te roepen met $N^x := U$ en $N^y := V$. De partities zijn zo gekozen, dat de matrix op regel 2 een bescheiden deel van het beschikbare geheugen in beslag neemt.

4.1 WORKUNITS

In de inleiding werd al gehint naar het gedistribueerd uitvoeren van het ABC@HOME-algoritme. Middels BOINC kan iedereen die dat wil onze software downloaden, en de overvloedige processortikken van zijn of haar computer inzetten om mee te rekenen aan ons project.

Zie [5].

Hoewel dit in principe een uitstekend idee lijkt voor *ieder* programma, leidt het in de praktijk vaak tot veel hoofdgekrab. Immers is het noodzakelijk om een methode te vinden om het werk in stukjes (zogenoeten "work units") op te delen die:

- werkt; (de complete zoekruimte overdekt)
- niet al te veel bandbreedte gebruikt in verhouding tot processorkracht;
- genoeg fourtherstellend vermogen heeft om het plotseling uitvallen van een aantal computers op te vangen.

Het hergebruiken van de eerder genoemde subblokken vol- doet uiteraard niet aan eis 2: de "centrale" computer (die alles coördineert) zou eerst in $\mathcal{O}(N^{\frac{2}{3}+\epsilon})$ tijd de complete zoekruimte moeten uitrekenen, en die vervolgens geheel (128 TB) verzenden.

Voor dit project:
mara.math.
leidenuniv.nl

Deze getallen liggen niet geheel buiten het bereik der redelijkheid (nu nog minder dan in 2005), maar ruimte voor verbetering is er zeker.

De workunits die wij gebruiken bevatten daarom slechts een handjevol parameters:

- de intervallen waar $\text{rad}(x)$ resp. $\text{rad}(y)$ in mogen vallen;
- de ggd van x met 210;
- het interval waarin c mag vallen.

De parameters zijn zo gekozen dat iedere workunit in ongeveer 4 uur is uitgerekend (onder optimale omstandigheden).

Om te kunnen voldoen aan eis 3 (en om fraude te voorkomen) wordt iedere workunit twee keer uitgevoerd. De vraag of de

complete zoekruimte bereikt is met de verzameling workunits die we nu hebben, is wederom materiaal voor een ander artikel.

5 | GEVOLGEN

Inmiddels hebben we gezien hoe het algoritme achter het ABC@HOME-project werkt, en hebben we kunnen in zien dat de werking correct is. Het is ook interessant om te weten welk hoger doel dit project dient — zeker nu het zo'n immense investering van enkele millennia processortijd moet kunnen verantwoorden.

Naar het antwoord op deze vraag werd in de inleiding al gehint: het ABC-vermoeden is een belangrijke speler in de Diophantische wiskunde. In [1] wordt in veel detail uitgelegd dat het ABC-vermoeden, als het klopt, impliceert dat een diophantische vergelijking van de vorm

$$x^p + y^q = z^r$$

(met gehele p, q, r, x, y, z , en $\text{ggd}(x, y) = 1$) slechts eindig veel oplossingen heeft als $\frac{1}{p} + \frac{1}{q} + \frac{1}{r} < 1$.

Hoewel Granville en Tucker dit op vrij elegante wijze laten zien, leent dit principe zich goed voor een voorbeeld. We beschouwen daarom de vergelijking

Zie [1].

$$x^2 + y^3 = z^7, \tag{5.1}$$

en zoeken naar geheeltallige oplossingen met $\text{ggd}(x, y) = 1$. Merk op, $\frac{1}{2} + \frac{1}{3} + \frac{1}{7} < 1$.

Stel, we vinden een oplossing x, y, z . Als we dan $a = x^2$, $b = y^3$, en $c = z^7$ nemen, dan zien we dat $a + b = c$, en $\text{rad}(abc) = \text{rad}(xyz) < c^{\frac{1}{2} + \frac{1}{3} + \frac{1}{7}} = c^{\frac{41}{42}} < c$. Dus (a, b, c) is een ABC-drietal.

Voor de kwaliteit is bovendien een ondergrens:

$$q(a, b, c) = \frac{\log c}{\log \text{rad}(abc)} > \frac{\log c}{\log c^{\frac{41}{42}}} = \frac{42}{41}.$$

Het ABC-vermoeden zegt nu dat er maar eindig veel ABC-drietallen zijn met kwaliteit minstens $\frac{42}{41}$. Dus zijn er maar eindig veel oplossingen voor vergelijking 5.1.

Het subtiele andere gevolg hiervan is dat als $c < 10^{18}$, dan staat de oplossing van 5.1 in onze lijst. Aangezien deze lijst beschikbaar is als SAGE-pakket, is dit een veel snellere methode om (kleine) oplossingen te vinden voor Diophantische vergelijkingen.

*Te downloaden vanaf
<http://abcathome.com/data>*

BIBLIOGRAFIE

- [1] Andrew Granville and Thomas J. Tucker, *It's As Easy As abc*. Notices of the AMS, Volume 49, Number 10, 2002.
- [2] Gérald Tenenbaum, *Introduction to Analytic and Probabilistic Number Theory*. Cambridge University Press, 2nd Edition, 2004.
- [3] Donald Knuth, *The Art of Computer Programming*, Volume 2: Seminumerical Algorithms. Addison Wesley, Massachusetts, 3rd Edition, 1997.
- [4] Torbjörn Granlund and Peter L. Montgomery, *Division By Invariant Integers Using Multiplication*. PLDI '94 Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation ACM New York, 1994.
- [5] *The BOINC project*, <http://boinc.berkeley.edu/>.
- [6] *Project Completion*, http://abcathome.com/forum_thread.php?id=535. Geraadpleegd op 21 juli 2011.
- [7] *AMD Optimization Guide*. http://support.amd.com/us/Processor_TechDocs/25112.PDF. Geraadpleegd op 21 juli 2011.

APPENDIX

A

PROGRAMMACODE

De broncode van het ABC@HOME-algoritme is als .tar.gz-bestand bijgevoegd in dit PDF-document.



Ten behoeve van de papieren versie zijn bovendien de belangrijkste delen in verbatim opgeschreven.

A.1 ABC_SIEVE_STANDALONE.CPP

```
1 #include <iostream>
2 #include <time.h>
3 #include <vector>
4 #include <cstdlib>
5 #include <cstdio>
6
7 #include "abc_common.h"
8 #include "abc_sieve.h"
9
10 #ifdef _MSC_VER
11 #define atoll _atoi64
12 #include "str_util.h"
13 #endif
14
15 #define VERBOSE
16
17 typedef unsigned long long intt;
18
19 intt META_hits;
20 intt META_trialdivcount;
21 intt META_triples_total;
22
23
24 void callback(std::vector<wudata>& d, intt triples_done
25             , intt triples_total, intt &hits, intt trialdivcount
26             )
27 {
28     hits += d.size();
29
30     #ifdef VERBOSE
31     #if 0
32     for (size_t i = 0; i < d.size(); ++i)
```

```

31         printf(FINTT " + " FINTT " = " FINTT "\n", d[i
           ].a, d[i].b, d[i].a+d[i].b);
32     #endif
33     printf("progress: %f\n", double(triples_done)/
           double(triples_total));
34 #endif
35
36     d.clear();
37
38     META_hits = hits;
39     META_trialdivcount = trialdivcount;
40     META_triples_total = triples_total;
41 }
42
43 void run_workunit(abc_sieve_header h)
44 {
45     intt triples_total = 0;
46     intt triples_done = 0;
47     intt hits = 0;
48     intt trialdivcount = 0;
49
Zie pagina 58 50     find_triples(h, triples_done, triples_total, hits,
           trialdivcount, &callback);
51     printf("finished with hits=" FINTT "\n", META_hits)
           ;
52 }
53
54 int main(int argc, char **argv)
55 {
56     abc_sieve_header h;
57
58     if (argc < 10 || argc > 11) {
59         fprintf(stderr, "bad command line: expected\
           nlower_rx upper_rx lower_ry upper_ry
           gcd_with gcd_rx lower_c upper_c blocksize [
           sievebound]");
60         return 1;
61     }
62     h.id = 0;
63     h.lower_rx = atoll(argv[1]);
64     h.upper_rx = atoll(argv[2]);
65     h.lower_ry = atoll(argv[3]);
66     h.upper_ry = atoll(argv[4]);
67     h.gcdwith = atoll(argv[5]);
68     h.gcdx = atoll(argv[6]);
69     h.lower_c_L0 = atoll(argv[7]);
70     h.lower_c_HI = 0;
71     h.upper_c_L0 = atoll(argv[8]);
72     h.upper_c_HI = 0;

```

```

73     h.blocksize = atoi(argv[9]);
74     if (argc < 11)
75         h.sievebound = estimate_sieve_bound(h.lower_rx,
76                                             h.lower_ry);
77     else
78         h.sievebound = atoi(argv[10]);
79     h.minQ = 0x01000000UL;
80
81     printf("Parameters: id = \"FINTT\", rx in [\"FINTT\", \"
      FINTT\"), ry in [\"FINTT\", \"FINTT\"), gcd(x, \"FINTT\")
      = \"FINTT\", c in [\"FINTT\", \"FINTT\"], bs %d, sb %d
      \n",
82          h.id, h.lower_rx, h.upper_rx, h.lower_ry, h.
      upper_ry, h.gcdwith,
83          h.gcdx, h.lower_c_L0, h.upper_c_L0, h.
      blocksize, h.sievebound);
84
85     run_workunit(h);
86     return 0;
87 }

```

Zie pagina 42

Zie pagina 28

A.2 ABC_SIEVE_UTIL.CPP

```

1  #include "abc_sieve_util.h"
2  #include <cassert>
3
4  const num_t ZERO = { 0, 0 };
5  const num_t ONE = { 1, 1 };
6
7  #if 0
8  bool operator<(const rad_t& r1, const rad_t& r2)
9  {
10     return r1.rad < r2.rad;
11 }
12
13 bool operator<(const num_t& r1, const num_t& r2)
14 {
15     return r1.num < r2.num;
16 }
17 #endif
18
19
20 // Find primes up to primelimit. (Fast.)
21 void sieveprimes(unsigned int primelimit,
22                 std::vector<unsigned int>& lowprimes)

```

```

23 {
24     bool* s = new bool[primelimit];
25     unsigned int primecount = 0;
26     for (unsigned int i = 0; i < primelimit; ++i)
27         s[i] = true;
28
29     for (unsigned int i = 2; i < primelimit; ++i) {
30         if (s[i]) {
31             primecount++;
32             for (unsigned int j = i+i; j < primelimit;
33                 j += i)
34                 s[j] = false;
35         }
36     }
37     lowprimes.resize(primecount);
38
39     primecount = 0;
40     for (unsigned int i = 2; i < primelimit; ++i)
41         if (s[i])
42             lowprimes[primecount++] = i;
43
44     delete[] s;
45 }
46
47 #if 0
48 // Find squarefrees up to radlimit. (Fast, but 0~(
49 //   radlimit) memory )
49 void sieveradicals(unsigned int radlimit, std::vector<
50     rad_t>& rads)
51 {
52     rad_t* r = new rad_t[radlimit];
53     unsigned int radcount = 0;
54     for (unsigned int i = 0; i < radlimit; ++i)
55         r[i].rad = 1;
56
57     for (unsigned int i = 2; i < radlimit; ++i) {
58         if (r[i].rad == 1) {
59             for (unsigned int j = i; j < radlimit; j +=
60                 i) {
61                 r[j].rad *= i;
62                 r[j].primes.push_back(i);
63             }
64         }
65         if (r[i].rad == i) radcount++;
66     }
67     rads.resize(radcount+1);
68     radcount = 0;
69     rads[radcount++] = r[1];

```

```

68     for (unsigned int i = 2; i < radlimit; ++i)
69         if (r[i].rad == i) {
70             rads[radcount++] = r[i];
71         }
72     delete[] r;
73 }
74 #endif
75
76 // Find squarefrees in interval. Fairly fast, memory
77 // linear in interval length
78 // Needs primes up to sqrt(start+length) in lowprimes
79 void sieveradicals_interval(intt start, unsigned int
80 length,
81 std::vector<rad_t>& rads,
82 std::vector<unsigned int>& lowprimes)
83 {
84     rad_t* r = new rad_t[length];
85     unsigned int radcount = 0;
86     for (unsigned int i = 0; i < length; ++i)
87         r[i].rad = 1;
88     if (start == 0)
89         r[0].rad = 0;
90
91     unsigned int i = 0;
92     unsigned int p = lowprimes[i];
93     while (((intt)p)*p <= start+length) {
94         intt q = 1;
95
96         intt plim = (start+length)/p;
97
98         while (q <= plim && q <= p) {
99             q *= p;
100             intt rem = start % q;
101             if (rem == 0) rem = q;
102
103             for (intt j = q - rem; j < length; j += q)
104                 {
105                     if (p == q) {
106                         r[j].primes.push_back(p);
107                         r[j].rad *= p;
108                     } else {
109                         r[j].rad = 0;
110                     }
111                 }
112             p = lowprimes[++i];
113     }
114
115     for (i = 0; i < length; ++i) {

```

```

114         if (r[i].rad != 0) radcount++;
115     }
116
117     rads.resize(radcount);
118     radcount = 0;
119     for (i = 0; i < length; ++i) {
120         if (r[i].rad != 0) {
121             assert( (start+i) % r[i].rad == 0);
122             intt lastp = (start + i) / r[i].rad;
123             if (lastp != 1)
124                 r[i].primes.push_back((start + i) / r[i]
125                                     ].rad);
126             r[i].rad = start + i;
127             rads[radcount++] = r[i];
128         }
129     }
130     delete[] r;
131 }
132
133 // Check if two squarefree integers are coprime.
134 // Based on code by Arjen Lenstra <arjen.lenstra@epfl.
Hij ontkent overigens alles. ch>, supplied by
135 // Alexander Petric <alexandre.petric@epfl.ch> and
136 // Bos Joppe <joppe.bos@epfl.ch>.
137 bool AreRadCoprime(unsigned int u, unsigned int v)
138 {
139     bool u_2 = false;
140     bool v_2 = false;
141
142     // get rid of factor 2
143     if (!(u & 1)) {
144         u_2 = true;
145         u >>= 1;
146     }
147     if (!(v & 1)) {
148         v_2 = true;
149         v >>= 1;
150     }
151
152     // if both are even
153     if (u_2 & v_2)
154         return false;
155
156     if (v == 1 || u == 1)
157         return true;
158
159     // now both u and v are odd
160     for (;;) {

```

```

161         if (u == v)
162             return (u == 1);
163
164         if (u > v) {
165             u -= v;
166             while (!(u&3)) u >>= 2;
167             if (!(u & 1)) u >>= 1;
168         } else {
169             v -= u;
170             while (!(v&3)) v >>= 2;
171             if (!(v & 1)) v >>= 1;
172         }
173     }
174 }
175
176
177 // Search all numbers with radical with index from
178 // index_from (inclusive)
179 // and radical size up to rad_limit (inclusive) and
180 // size up to num_limit (inc.)
181 // and coprime to coprime_to.
182 // Radicals are taken from the rads[] vector.
183 // The nums[] argument is used for temporary storage.
184 // Returns the number of radicals considered.
185 intt ForEachRad(ForEachRadFunc f,
186                unsigned int index_from, unsigned int rad_limit
187                ,
188                std::vector<rad_t>& rads,
189                intt num_limit, unsigned int coprime_to,
190                std::vector<num_t>& nums,
191                num_t data)
192 {
193     nums.resize(NUMPERRAD_LIMIT); // max. number of ints
194     // with given radical
195
196     assert(num_limit >= rad_limit);
197
198     unsigned int rads_size = rads.size();
199     intt done_radcount = 0;
200
201     for (unsigned int i = index_from; i < rads_size; ++i)
202     {
203         rad_t& r = rads[i];
204
205         if (r.rad > rad_limit) break;
206
207         if (coprime_to > 1 && !AreRadCoprime(coprime_to
208             , r.rad)) continue;
209     }

```

Zie pagina 32


```

204         ++doneradcount;
205
206         num_t n;
207         n.num = r.rad;
208         n.rad = r.rad;
209
210         bool ret = (*f)(n, i, data);
211         if (!ret) continue;
212
213         unsigned int count = 0;
214         nums[count++] = n;
215
216         std::list<unsigned int>::iterator piter;
217
218         bool breakloop = false;
219
220         for (piter = r.primes.begin(); piter != r.
                primes.end() && !breakloop; ++piter) {
221             unsigned int p = *piter;
222             intt plim = num_limit / p;
223             for (unsigned int k = 0; k < count; ++k) {
224                 intt a = nums[k].num;
225                 if (a <= plim) {
226                     a *= p;
227                     n.num = a;
228                     nums[count++] = n;
229                     assert(count < NUMPERRAD_LIMIT);
230                     assert(nums.size() ==
                            NUMPERRAD_LIMIT);
231
232                     ret = (*f)(n, i, data);
233                     // stop processing this radical
234                     // if requested by f
235                     if (!ret) { breakloop = true; break
                                ; }
236                 }
237             }
238         }
239     }
240
241     return doneradcount;
242 }
243
244
245 // find square root of number if it is a perfect square
246 // if not a square, returns square root rounded down
247 // TODO: this can probably be done much faster
248 bool issquare(intt n, unsigned int& r)
249 {

```

```

250     unsigned int lo = 0;
251     unsigned int hi = 4294967295ULL; // 2^32-1
252     //unsigned int hi = 1000000000; // 10^9
253     while (lo + 1 < hi) {
254         unsigned int v = lo/2 + hi/2 + ((lo&1)+(hi&1))
                /2;
255         if (((intt)v)*v < n)
256             lo = v;
257         else
258             hi = v;
259     }
260
261     if (((intt)lo)*lo == n) {
262         r = lo;
263         return true;
264     }
265
266     if (((intt)hi)*hi == n) {
267         r = hi;
268         return true;
269     }
270
271     r = hi-1;
272
273     return false;
274 }
275 bool iscube(intt n, unsigned int& r)
276 {
277     unsigned int lo = 0;
278     unsigned hi = 2642245; // floor((2^64-1)^(1/3))
279     //unsigned hi = 1000000; // 10^6
280     while (lo + 1 < hi) {
281         unsigned int v = (lo+hi)/2;
282         if (((intt)v)*v*v < n)
283             lo = v;
284         else
285             hi = v;
286     }
287
288     if (((intt)lo)*lo*lo == n) {
289         r = lo;
290         return true;
291     }
292
293     if (((intt)hi)*hi*hi == n) {
294         r = hi;
295         return true;
296     }
297

```

```

298     r = hi-1;
299
300     return false;
301 }
302
303 struct ppower_t {
304     unsigned int p;
305     intt pk;
306 };
307
308 bool is_fifth_ppower(intt n, unsigned int& p)
309 {
310     #include "5thpp.h"
311     int l = 0;
312     int h = sizeof(cand)/sizeof(cand[0])-1;
313     while (l+1 < h) {
314         unsigned int v = (l+h)/2;
315         if (cand[v].pk < n)
316             l = v;
317         else
318             h = v;
319     }
320
321     if (cand[l].pk == n) {
322         p = cand[l].p;
323         return true;
324     }
325     if (cand[h].pk == n) {
326         p = cand[h].p;
327         return true;
328     }
329
330     return false;
331 }
332
333 bool is_seventh_ppower(intt n, unsigned int& p)
334 {
335     #include "7thpp.h"
336     int l = 0;
337     int h = sizeof(cand)/sizeof(cand[0])-1;
338
339     while (l+1 < h) {
340         unsigned int v = (l+h)/2;
341         if (cand[v].pk < n)
342             l = v;
343         else
344             h = v;
345     }
346

```

```

347     if (cand[l].pk == n) {
348         p = cand[l].p;
349         return true;
350     }
351     if (cand[h].pk == n) {
352         p = cand[h].p;
353         return true;
354     }
355
356     return false;
357 }

```

A.3 ABC_SIEVE.CPP

```

1  #include <vector>
2  #include <iostream>
3  #include <algorithm>
4  #include <cstdio>
5  #ifndef _MSC_VER
6  #include <sys/time.h>
7  #endif
8  #include "abc_common.h"
9  #include "abc_sieve_util.h"
10
11 #ifdef __x86_64__
12 #define ASM64
13 #endif
14 #ifdef WIN64
15 #define ASM64
16 #endif
17 #define STATS
18 #define NDEBUG
19
20 const unsigned int NX_LIMIT = 400000;
21 const unsigned int NY_LIMIT = 1600000;
22 const unsigned int P_LIMIT = 200000;
23 const unsigned int BLOCK_LIMIT = 10000;
24
25 #include <cassert>
26 #include <cmath>
27
28 struct Index {
29     intt num;
30     unsigned int rad;
31     bool minus;

```

```

32     bool operator< (const Index& other) const { return
        num < other.num; }
33 };
34
35 struct Entry {
36     intt left;
37     intt rad;
38 };
39
40
41 class ABCSieve {
42 public:
Zie pagina 40 43     ABCSieve(abc_sieve_header& h, ScanCallback cb);
Zie pagina 41 44     void printStats();
Zie pagina 55 45     void find_triples(intt skiptriples);
Zie pagina 58 46     void setStats(intt totalcount, intt donecounter,
        intt foundcounter, intt trialdivcounter);
47
48
49 private:
Zie pagina 45 50     bool trial_div(intt n, intt left, intt rad, intt
        maxrad, intt minrad, unsigned int p_index);
Zie pagina 48 51     void prepare_buckets(unsigned int start, unsigned
        int n, unsigned int ip);
Zie pagina 49 52     void sieve_primes_bucket(unsigned int startX,
        unsigned int startY,
53         unsigned int nx, unsigned
        int ny,
54         unsigned int limit,
        unsigned int& index);
55     intt get_num(unsigned int ix, unsigned int iy)
        const {
56         if (listX[ix].minus)
57             return listY[iy].num + listX[ix].num;
58         if (listX[ix].num >= listY[iy].num)
59             return listX[ix].num - listY[iy].num;
60         else
61             return listY[iy].num - listX[ix].num;
62     }
Zie pagina 52 63     void do_block(int blocksize, int sieve_bound, bool
        count, intt& skip_triples);
Zie pagina 50 64     void finish_block(unsigned int startX, unsigned int
        startY,
65         unsigned int nx, unsigned int ny,
        unsigned int p_index);
66
Zie pagina 43 67     bool rad_pp_gt_maxrad(unsigned int p, intt pp, intt
        rad, intt left, intt minrad, intt maxrad);
68

```

```

69     ScanCallback callback;
70     abc_sieve_header header;
71     unsigned int minrx, minry, maxrx, maxry;
72
73     // Storage
74     std::vector<unsigned int> lowprimes;
75     std::vector<rad_t> lowrads;
76     std::vector<rad_t> highrads;
77     std::vector<num_t> nums_tmp;
78     std::vector<Entry> block;
79     std::vector<int> bucketX;
80     std::vector<int> nodesx;
81
82     std::vector<Index> listX;
83     std::vector<Index> listY;
84
85 #ifndef _MSC_VER
86     // Timing
87     struct timeval nowT, startT;
88 #endif
89
90     // Output
91     std::vector<wudata> OUT_foundtriples;
92     unsigned long long OUT_totalcount;
93     unsigned long long OUT_donecounter;
94     unsigned long long OUT_foundcounter;
95     unsigned long long OUT_trialdivcounter;
96
97
98 #ifdef STATS
99     intt stat_totalblocksize;
100    intt stat_doneblocksize;
101    intt stat_sieve_bucket_marks;
102    intt stat_sieve_modY;
103    intt stat_sieve_modX;
104    intt stat_sieve_marks;
105    intt stat_notcoprime;
106    intt stat_notcoprime_repeat;
107    intt stat_rxryrytoolarge;
108    intt stat_rxryrztoolarge;
109    intt stat_trialdiv_reached;
110    intt stat_trialdiv_start;
111    intt stat_trialdiv_smooth;
112    intt stat_trialdiv_end;
113    intt stat_trialdiv_smallescape;
114    intt stat_trialdiv_escape;
115    intt stat_trialdiv_power;
116    intt stat_trialdiv_factors;
117    intt stat_trialdiv_rounds;

```

```

118     intt stat_trialdiv_endsquare;
119 #endif
120 };
121
122 ABCSieve::ABCSieve(abc_sieve_header& h, ScanCallback cb
123 ) {
124     header = h;
125     callback = cb;
126
127     OUT_totalcount = 0;
128     OUT_donecounter = 0;
129     OUT_foundcounter = 0;
130     OUT_trialdivcounter = 0;
131 #ifdef STATS
132     stat_totalblocksize = 0;
133     stat_doneblocksize = 0;
134     stat_sieve_bucket_marks = 0;
135     stat_sieve_modY = 0;
136     stat_sieve_modX = 0;
137     stat_sieve_marks = 0;
138     stat_notcoprime = 0;
139     stat_notcoprime_repeat = 0;
140     stat_rxyryrytoolarge = 0;
141     stat_rxyryrztoolarge = 0;
142     stat_trialdiv_reached = 0;
143     stat_trialdiv_start = 0;
144     stat_trialdiv_smooth = 0;
145     stat_trialdiv_end = 0;
146     stat_trialdiv_smallescape = 0;
147     stat_trialdiv_escape = 0;
148     stat_trialdiv_power = 0;
149     stat_trialdiv_factors = 0;
150     stat_trialdiv_rounds = 0;
151     stat_trialdiv_endsquare = 0;
152 #endif
153
154     sieveprimes(2105000, lowprimes); // 2^21 + a bit
155     // init_Two64divp(lowprimes.size());
156     // init_prime_inv(lowprimes.size());
157
158     // sieveradicals(200000, lowrads);
159     // sieveradicals(200000, highrads);
160     // sieveradicals_interval(80000000ULL, 10000, highrads
161     // , lowprimes);
162
163     nums_tmp.reserve(80000);
164
165     bucketX.resize(P_LIMIT);

```

```

165     nodesx.resize(BLOCK_LIMIT);
166 }
167
168 void ABCSieve::printStats() {
169 #ifdef STATS
170     printf("total blocksize:%14lld\n",
171           stat_totalblocksize);
172     printf("processed size:%15lld\n",
173           stat_doneblocksize);
174     printf("sieve b. marked:%14lld\n",
175           stat_sieve_bucket_marks);
176     printf("sieve b. X mod p:%13lld\n", stat_sieve_modX
177           );
178     printf("sieve b. Y mod p:%13lld\n", stat_sieve_modY
179           );
180     printf("sieve marked:   %13lld\n",
181           stat_sieve_marks);
182     printf("rx ry^2 > c:   %13lld\n",
183           stat_rxyrytoolarge);
184     printf("rx ry rz' > c:  %13lld\n",
185           stat_rxyztoolarge);
186     printf("not coprime:   %13lld\n", stat_notcoprime
187           );
188     printf("not coprime rep: %13lld\n",
189           stat_notcoprime_repeat);
190
191     printf("reached trialdiv:%13lld\n",
192           stat_trialdiv_reached);
193     printf("start of trialdiv:%12lld\n",
194           stat_trialdiv_start);
195     printf("smooth trialdiv: %13lld\n",
196           stat_trialdiv_smooth);
197     printf("early exit trdiv:%13lld\n",
198           stat_trialdiv_escape);
199     printf("early exit power:%13lld\n",
200           stat_trialdiv_power);
201     printf("early exit small:%13lld\n",
202           stat_trialdiv_smallescapes);
203     printf("trialdiv factors:%13lld\n",
204           stat_trialdiv_factors);
205     printf("trialdiv rounds:%14lld\n",
206           stat_trialdiv_rounds);
207     printf("end of trialdiv: %13lld\n",
208           stat_trialdiv_end);
209     printf("end square:     %13lld\n",
210           stat_trialdiv_endsquare);
211 #endif
212 }
213

```



```

194 #include "prime_inv.h"
195 #include "prime_inv2.h"
196 // #include "tables.h"
197
198 #include "asm.h"
199
200 #ifdef ASM64
201 #include "invH.h"
202 #endif
203
204 #ifndef _MSC_VER
205 #define STARTTIME do { gettimeofday(&startT, NULL); }
      while(0)
206 #define ENDTIME do { gettimeofday(&nowT, NULL); printf(
      "%ld\n", (nowT.tv_sec-startT.tv_sec)*1000+(nowT.
      tv_usec-startT.tv_usec)/1000); } while(0)
207 #else
208 #define STARTTIME
209 #define ENDTIME
210 #endif
211 #ifdef STATS
212 #define PRINTTIME ENDTIME
213 #define INC_STAT(s) do { (s)++; } while(0)
214 #else
215 #define PRINTTIME
216 #define INC_STAT(s)
217 #endif
218
219
220
221 unsigned int estimate_sieve_bound(unsigned int Lx,
      unsigned int Ly)
222 {
223     intt r = Lx;
224     r *= Ly;
225     unsigned int S;
226
227     if (r < 268435456) {
228         S = 61000 + (unsigned int)(15000 * (28 - log((
      double)r) / log(2.0)));
229     } else {
230         S = (unsigned int)sqrt((double)
      (1000000000000000000ULL / r));
231     }
232     if (S > 200000) S = 200000;
233     return S;
234 }
235
236 std::vector<Index>* store_target;

```

```

237
238 bool storeRad(num_t n, unsigned int, num_t data)
239 {
240     if (n.rad % data.rad) return false;
241     Index a;
242     a.num = n.num;
243     a.rad = n.rad;
244     a.minus = false;
245     store_target->push_back(a);
246     return true;
247 }
248
249 // Find index of smallest element in a sorted rad_t
    vector with value >= target
250 static int binsearch(const std::vector<rad_t>& a, intt
    target)
251 {
252     int l = 0;
253     int h = a.size()-1;
254     while (l <= h) {
255         int t = (l + h) / 2;
256         if (a[t].rad == target) return t;
257         if (a[t].rad > target) {
258             h = t-1;
259         } else {
260             l = t+1;
261         }
262     }
263     // not found, so return smallest index with value
        above target
264     return l;
265 }
266
267
268 // Input: (Implicit:) n - number to factor
269 //         left - the factor of n coprime to all primes
        < p
270 //         rad - the radical of (n/left)
271 //         pp - p*p
272 // Assumption: rad * p * p > maxrad
273 // Assumption: p > 59
274 // Output: rad(n) <= maxrad && rad(n) > minrad
275
276 bool ABCSieve::rad_pp_gt_maxrad(unsigned int p, intt /*
        pp*/, intt rad, intt left, intt minrad, intt maxrad)
277 {
278     // now either:
279     // A. left has one prime divisor:
280     //     left is a prime power and rad(n) is known

```

```

281 // B. left has >=2 prime divisors:
282 //   r(left) > p*p and rad(n) > maxrad
283
284 if (p < 300)
285     INC_STAT(stat_trialdiv_smallescape);
286
287
288 // Check if left is a prime-power:
289 // left is not divisible by primes <= 59
290 // (since we just tested the first 17)
291 // 61^10 < 10^18 < 61^11
292 // So: only check 2nd, 3rd, 5th, 7th powers
293 unsigned int cuberoot, root;
294 unsigned int m63 = left % 63;
295 if ((m63 == 1 || m63 == 8 || m63 == 55 || m63 ==
Zie pagina 35 62) && iscube(left, cuberoot)) {
296     INC_STAT(stat_trialdiv_power);
297     // third power; maybe 6th, 9th?
298     left = cuberoot;
299     m63 = left % 63;
300     if ((m63 == 1 || m63 == 8 || m63 == 55 || m63 ==
Zie pagina 35 == 62) && iscube(left, cuberoot)) {
301         left = cuberoot; // 9th power
302     } else if (((left & 7) == 1) && (m63 == 1 ||
303         m63 == 4 || m63 == 16 || m63 == 22 || m63 ==
304         25 || m63 == 37 || m63 == 43 || m63 == 46
Zie pagina 34 || m63 == 58) && issquare(left, root)) {
305         left = root; // 6th power
306     }
307     // Now: left is a prime or
308     // left >= rad(left) > p^2
309     return (rad*left <= maxrad && rad*left > minrad
310         );
311 }
312 if (((left & 7) == 1) && (m63 == 1 || m63 == 4 ||
313     m63 == 16 || m63 == 22 || m63 == 25 || m63 == 37
314     || m63 == 43 || m63 == 46 || m63 == 58) &&
Zie pagina 34 issquare(left, root)) {
315     INC_STAT(stat_trialdiv_power);
316     left = root;
317     // square; maybe 4th, 8th, 10th
318     if (issquare(left, root)) {
319         left = root; // 4th power
Zie pagina 34 if (issquare(left, root)) {
320             left = root; // 8th power
321         }
322     } else if (left == 844596301) {
323         // the only possible 10th powers are
324         {61,67,71,73}^10

```

```

320         left = 61;
321     } else if (left == 1350125107) {
322         left = 67;
323     } else if (left == 1804229351) {
324         left = 71;
325     } else if (left == 2073071593) {
326         left = 73;
327     }
328     // Now: left is a prime or
329     // left >= rad(left) > p^2
330     return (rad*left <= maxrad && rad*left > minrad
331         );
332 }
333 unsigned int m11 = left % 11;
334 if ((m11 == 1 || m11 == 10) && is_fifth_ppower(left
335     ,root)) {
336     INC_STAT(stat_trialdiv_power);
337     return (rad*root <= maxrad && rad*root > minrad
338         );
339 }
340 unsigned int m29 = left % 29;
341 if ((m29 == 1 || m29 == 12 || m29 == 17 || m29 ==
342     28) && is_seventh_ppower(left, root)) {
343     INC_STAT(stat_trialdiv_power);
344     return (rad*root <= maxrad && rad*root > minrad
345         );
346 }
347 INC_STAT(stat_trialdiv_escape);
348 return (rad*left <= maxrad && rad*left > minrad);
349 }
350 // Do a trial division by primes from lowprimes,
351 // starting at index p_index
352 // Input: n - number to factor
353 // left - n divided by the p-smooth part of n (p
354 // = lowprimes[p_index-1]
355 // rad - radical of the p-smooth part of n
356 // Output: rad(n) <= maxrad && rad(n) > minrad
357
358 bool ABCSieve::trial_div(intt n, intt left, intt rad,
359     intt maxrad, intt minrad, unsigned int p_index)
360 {
361     INC_STAT(stat_trialdiv_reached);
362
363 #ifndef NDEBUG
364     intt _n = n;
365     intt _left = left;
366     intt _rad = rad;

```

Zie pagina 36

Zie pagina 36

```

361     intt _maxrad = maxrad;
362 #else
363     (void)n; // n is used in debug mode only
364 #endif
365
366     assert(left);
367
368     if (rad > maxrad) {
369         INC_STAT(stat_trialdiv_start);
370         return false;
371     }
372     if (left == 1) {
373         INC_STAT(stat_trialdiv_start);
374         return (rad <= maxrad && rad > minrad);
375     }
376     unsigned int p = lowprimes[p_index];
377 #ifndef NDEBUG
378     if (left < p) {
379         printf("error: %llu < %d\n (%llu, %llu, %llu, %llu)", left, p, _n, _left, _rad, _maxrad);
380         assert(left >= p);
381     }
382 #endif
383     if (p > maxrad) {
384         INC_STAT(stat_trialdiv_start);
385         return false;
386     }
387
388     // handle p == 2 separately
389     if (p_index == 0) {
390         assert(rad == 1);
391
392         if (!(left & 1)) {
393             rad = 2;
394             do {
395                 left >>= 1;
396             } while (!(left & 1));
397         }
398         p = lowprimes[++p_index];
399     }
400
401     OUT_trialdivcounter++;
402
403     while (true) {
404         for (int i = 0; i < 16; ++i) {
405             INC_STAT(stat_trialdiv_rounds);
406             intt t = left * prime_inv[p_index];
407             if (t < Two64divp[p_index]) {
408                 INC_STAT(stat_trialdiv_factors);

```

```

409         rad *= p;
410         do {
411             left = t;
412             t = t * prime_inv[p_index];
413         } while (t < Two64divp[p_index]);
414     }
415     p = lowprimes[++p_index];
416 }
417
418 if (left == 1) {
419     INC_STAT(stat_trialdiv_smooth);
420     return (rad <= maxrad && rad > minrad);
421 }
422 #ifndef NDEBUG
423 if (left < p) {
424     printf("error: %llu >= %d\n (%llu, %llu, %
425         llu, %llu)", left, p, _n, _left, _rad,
426         _maxrad);
427     assert(left >= p);
428 }
429 #endif
430
431 intt pp = p;
432 pp *= p;
433
434 // p is large enough to reach a conclusion
435 if (rad*pp > maxrad)
436     return rad_pp_gt_maxrad(p, pp, rad, left,
437         minrad, maxrad);
438
439 if (pp*p > left) {
440     INC_STAT(stat_trialdiv_end);
441     // now either:
442     // A. left is the square of a prime
443     // B. left is squarefree
444
445     unsigned int root;
446     if (((left & 7) == 1) && issquare(left,
447         root)) {
448         INC_STAT(stat_trialdiv_endsquare);
449         return (rad*root <= maxrad && rad*root
450             > minrad);
451     }
452
453     return (rad*left <= maxrad && rad*left >
454         minrad);
455 }
456 }

```

Zie pagina 43

```

452     assert(false);
453 }
454
455
456 void ABCSieve::prepare_buckets(unsigned int start,
    unsigned int n, unsigned int ip)
457 {
458     assert(n < BLOCK_LIMIT);
459
460     // Sort listX[start] up to listX[start+n-1] into
    buckets mod p.
461
462     unsigned long long p = lowprimes[ip];
463 #ifdef ASM64
464     unsigned char s = InvT[ip].s;
465     unsigned long long q = InvT[ip].q;
466 #endif
467
468
469     // bucketX + nodesx form a set of linked lists.
470     // bucketX[i] points to the head node of list i, or
    -1 if the list is empty.
471     // nodesx[i] is the node following node i, or -1
    for the tail of a list.
472
473     assert(p <= P_LIMIT);
474     for (unsigned i = 1; i < p; ++i)
475         bucketX[i] = -1;
476     for (unsigned int i = 0; i < n; ++i) {
477         unsigned int modp;
478         MOD(modp, listX[start+i].num, p, q, s);
479 //         std::cout << l[i].num << " % " << p << " == "
    << (l[i].num % p) << " != " << modp << " ( q = " <<
    q << ", s = " << (unsigned int)s << " )" << std::
    endl;
480         assert(modp == listX[start+i].num % p);
481         INC_STAT(stat_sieve_modX);
482         if (!modp) continue;
483         if (listX[start+i].minus) modp = p - modp;
484         nodesx[i] = bucketX[modp];
485         assert(bucketX[modp] == -1 || (bucketX[modp] >=
    0 && (unsigned int)bucketX[modp] < n));
486         bucketX[modp] = i;
487     }
488 }
489
490
491 // Sieve through block with x's in listX at indices [
    startX, startX+nx) and

```

```

492 // y's in listY at indices [startY,startY+ny)
493
494 // Divide by primes in lowprimes, starting at index,
    and continuing with all
495 // primes < limit. Afterwards the parameter index is
    the index of the first
496 // prime not yet handled.
497
498 // Result: for every entry e in the block of (x,y)
    pairs,
499 // e.left and e.rad are updated for all primes p above.
500
501 void ABCSieve::sieve_primes_bucket(unsigned int startX,
    unsigned int startY,
502                                     unsigned int nx,
    unsigned int ny,
503                                     unsigned int limit,
    unsigned int&
    index)
504 {
505     unsigned int ip;
506     for (ip = index; true; ++ip) {
507         unsigned int p = lowprimes[ip];
508         if (p > limit) break;
509         intt pinv = prime_inv[ip];
510         intt pmax = Two64divp[ip];
511 #ifdef ASM64
512         unsigned char s = InvT[ip].s;
513         unsigned long long q = InvT[ip].q;
514 #endif
515         //std::cout << p << std::endl;
516         prepare_buckets(startX, nx, ip);
517
518         for (unsigned int iy = 0; iy < ny; ++iy) {
519             unsigned int modp;
520             MOD(modp,listY[startY+iy].num,p,q,s);
521             assert(listY[startY+iy].num % p == modp);
522             INC_STAT(stat_sieve_modY);
523             if (!modp) continue;
524             for (int ix = bucketX[modp]; ix >= 0; ix =
                nodesx[ix]) {
525                 assert((unsigned int)ix < nx);
526                 //printf("modp=%s%lld vs %d, p=%d\n", (
                    listX[ix].minus ? "-" : ""), (listX[
                    ix].num % p), modp, p);
527                 Entry& e = block[iy*nx+ix];
528                 if (!e.left) continue;
529                 e.rad *= p;
530                 INC_STAT(stat_sieve_bucket_marks);

```

Zie pagina 48


```

531         intt t = e.left * pinv;
532 #ifndef NDEBUG
533         if (t >= pmax) {
534             printf("p=%d, x=%s%lld, y=%lld,
                    left=%lld\n", p, (listX[startX+
                    ix].minus ? "-" : ""), listX[
                    startX+ix].num, listY[startY+iy
                    ].num, e.left);
535             assert(t < pmax);
536         }
537 #endif
538         do {
539             e.left = t;
540             t = t * pinv;
541         } while (t < pmax);
542     }
543 }
544 }
545 index = ip;
546 }
547
548
549 void ABCSieve::finish_block(unsigned int startX,
                    unsigned int startY,
550                             unsigned int nx, unsigned
                    int ny,
551                             unsigned int p_index)
552 {
553     for (unsigned int iy = 0; iy < ny; ++iy) {
554         unsigned int ry = listY[startY+iy].rad;
555         unsigned int last_rx = 0;
556         bool not_coprime = false;
557         for (unsigned int ix = 0; ix < nx; ++ix) {
558             unsigned int rx = listX[startX+ix].rad;
559             if (rx == last_rx && not_coprime) {
560                 INC_STAT(stat_notcoprime_repeat);
561                 continue;
562             }
563             if (rx >= ry) {
564                 // checking equality means this catches
                    // the 1 + 1 = 2 case too
565                 continue;
566             }
567
568             intt num = get_num(startX+ix, startY+iy);
569
570             if (num == 0) continue;
571             if (num > header.upper_c_L0) continue;
572

```

Zie pagina 38

```

573     intt left, rz;
574     if (p_index) {
575         Entry& e = block[iy*nx+ix];
576         left = e.left;
577         rz = e.rad;
578     } else {
579         left = num;
580         rz = 1;
581     }
582
583     intt c = num;
584     if (listX[startX+ix].num > c) c = listX[
585         startX+ix].num;
586     if (listY[startY+iy].num > c) c = listY[
587         startY+iy].num;
588
589     intt t = rx;
590     t *= ry;
591     // rx*ry*ry < rx*ry*rz < c for triples
592     if (t*ry >= c) {
593         INC_STAT(stat_rxyrytoolarge);
594         continue;
595     }
596     if (t*rz >= c) {
597         INC_STAT(stat_rxyztoolarge);
598         continue;
599     }
600     if (last_rx != rx && !AreRadCoprime(rx, ry)) Zie pagina 32
601     {
602         last_rx = rx;
603         INC_STAT(stat_notcoprime);
604         not_coprime = true;
605         continue;
606     }
607     last_rx = rx;
608     not_coprime = false;
609     intt B = (c / t);
610
611     if (trial_div(num, left, rz, B, ry, p_index) Zie pagina 45
612         ) {
613         // We found a triple.
614
615         wodata triple;
616         if (listX[startX+ix].minus) {
617             // {x,y} = {a,b}
618             if (listX[startX+ix].num > listY[
619                 startY+iy].num) {

```

```

617             triple.b = listX[startX+ix].num
618             ;
619             triple.a = listY[startY+iy].num
620             ;
621         } else {
622             triple.a = listX[startX+ix].num
623             ;
624             triple.b = listY[startY+iy].num
625             ;
626         }
627     } else {
628         intt n1,n2;
629         n1 = num;
630         if (listX[startX+ix].num > listY[
631             startY+iy].num)
632             n2 = listY[startY+iy].num;
633         else
634             n2 = listX[startX+ix].num;
635
636         if (n1 < n2) {
637             triple.a = n1;
638             triple.b = n2;
639         } else {
640             triple.b = n1;
641             triple.a = n2;
642         }
643     }
644     OUT_foundtriples.push_back(triple);
645     // NB: do NOT increment
646     // OUT_foundcounter, as that
647     // is handled by the callback
648 #ifdef STATS
649     std::cout << "Triple: " << triple.a <<
650     " + " << triple.b << " = " << (
651     triple.a + triple.b) << std::endl;
652 #endif
653 }
654 }
655 }
656
657 void ABCSieve::do_block(int blocksize, int sieve_bound,
658     bool count,
659     intt& skip_triples)
660 {
661     unsigned int totx = listX.size();
662     unsigned int toty = listY.size();

```

```

657     std::sort(listX.begin(), listX.end());
658     std::sort(listY.begin(), listY.end());
659
660     std::cout << "totx = " << totx << ", toty = " <<
        toty << std::endl;
661
662     const unsigned int LIMIT_SOFT = blocksize;
663     const unsigned int LIMIT_HARD = blocksize + 100;
664     if (sieve_bound)
665         block.resize(LIMIT_HARD * LIMIT_HARD);
666
667     assert(totx <= NX_LIMIT);
668     assert(toty <= NY_LIMIT);
669
670     PRINTTIME;
671     if (totx == 0 || toty == 0) return;
672
673 #ifdef STATS
674     stat_totalblocksize += ((intt)totx)*toty;
675 #endif
676
677     unsigned int nx = 0, ny = 0;
678     for (unsigned int startX = 0; startX < totx; startX
        += nx) {
679         nx = LIMIT_SOFT;
680         if (totx - startX < LIMIT_HARD) {
681             nx = totx - startX;
682         }
683
684         for (unsigned int startY = 0; startY < toty;
            startY += ny) {
685             ny = LIMIT_SOFT;
686             if (toty - startY < LIMIT_HARD) {
687                 ny = toty - startY;
688             }
689
690             std::cout << "subblock: " << startX << ", "
                << startY
691                 << " of size " << nx << "x" << ny
                    ;
692
693             intt maxx = listX[startX+nx-1].num;
694             intt maxy = listY[startY+ny-1].num;
695             intt rxryry = (((intt)minrx)*minry)*minry;
696             if (!listX[0].minus && maxy < rxryry &&
                maxx < rxryry) {
697                 std::cout << ", skipping: x or y too
                    small"
698                     << std::endl;

```

```

699         continue;
700     }
701     if (listX[0].minus && maxx+maxy < rxryry) {
702         std::cout << ", skipping: x+y too small
703         "
704         << std::endl;
705         continue;
706     }
707     std::cout << std::endl;
708     if (count) {
709         skip_triples += nx*ny;
710         continue;
711     }
712
713     if (skip_triples > 0) {
714         if (skip_triples < nx*ny) {
715             std::cerr << "Inconsistency
716             detected in checkpointed state.
717             Aborting.";
718             exit(1);
719         }
720         skip_triples -= nx*ny;
721         continue;
722     }
723     #ifdef STATS
724     stat_doneblocksize += nx*ny;
725     #endif
726     unsigned int index = 0;
727     if (sieve_bound) {
728         for (unsigned int iy = 0; iy < ny; ++iy
729         ) {
730             for (unsigned int ix = 0; ix < nx;
731             ++ix) {
732                 intt left = get_num(startX+ix,
733                 startY+iy);
734                 unsigned int rad = 1;
735                 if (left && !(left & 1)) {
736                     rad = 2;
737                     do {
738                         left >>= 1;
739                     } while (!(left & 1));
740                 }
741                 block[iy*nx+ix].left = left;
742                 block[iy*nx+ix].rad = rad;
743             }
744         }

```

Zie pagina 38

```

742
743         PRINTTIME;
744
745         index = 1;
746         sieve_primes_bucket(startX, startY, nx,           Zie pagina 49
            ny, sieve_bound, index);
747         PRINTTIME;
748     }
749     finish_block(startX, startY, nx, ny, index)         Zie pagina 50
        ;
750     PRINTTIME;
751     //typedef void (*ScanCallback) (std::vector<wudata>& d,
        intt scanned, intt total, intt& found, intt
        trialdivs);
752
753         OUT_donecounter += nx*ny;
754         (*callback)(OUT_foundtriples,
            OUT_donecounter, OUT_totalcount,
            OUT_foundcounter, OUT_trialdivcounter);
755     }
756 }
757 }
758
759 void ABCSieve::find_triples(intt skiptriples)
760 //int Lx, int Ux, int Ly, int Uy, int g, int
    sieve_bound)
761 {
762     STARTTIME;
763
764     minrx = header.lower_rx;
765     minry = header.lower_ry;
766     maxrx = header.upper_rx;
767     maxry = header.upper_ry;
768
769     if (header.lower_c_HI != 0 || header.lower_c_LO !=
        1) {
770         std::cerr << "This client does not support minC
            != 1"
771             << std::endl;
772         return;
773     }
774
775     if (header.upper_c_HI != 0 || (header.upper_c_LO &
        0x8000000000000000ULL)) {
776         std::cerr << "This client does not support maxC
            >= 2^63"
777             << std::endl;
778         return;
779     }

```

```

780     if (header.upper_rx > 0xFFFFFFFFUL || header.
781         upper_ry > 0xFFFFFFFFUL) {
782         std::cerr << "This client does not support
783             radicals >= 2^32"
784             << std::endl;
785         return;
786     }
787     if (header.gcdwith > 0xFFFFFFFFUL) {
788         std::cerr << "This client does not support
789             gcdwith >= 2^32"
790             << std::endl;
791         return;
792     }
793     if (header.minQ != 0x01000000UL) {
794         std::cerr << "This client does not support minQ
795             != 1"
796             << std::endl;
797         return;
798     }
799     if (header.sievebound > 300000) {
800         // This is due to the size of precomputed prime
801         // tables
802         std::cerr << "This client does not support
803             sievebound > 300000"
804             << std::endl;
805         return;
806     }
807     if (header.sievebound > P_LIMIT) {
808         // see prepare_buckets
809         std::cerr << "This client does not support
810             sievebound > " << P_LIMIT
811             << std::endl;
812         return;
813     }
814     if (header.blocksize > 10000) {
815         std::cerr << "This client does not support
816             blocksize > 10000"
817             << std::endl;
818         return;
819     }
820     const intt _limit = header.upper_c_L0;
821     sieveradicals_interval(minrx,maxrx-minrx,lowrads,
822         lowprimes);
823     sieveradicals_interval(minry,maxry-minry,highrads,
824         lowprimes);

```

Zie pagina 31

```

819 unsigned int lowradx = binsearch(lowrads, minrx);
820 unsigned int lowrady = binsearch(highrads, minry);
821
822 unsigned int t = header.gcdwith;
823 unsigned int g = header.gcdx;
824
825 if (skiptriples)
826     std::cout << "Resuming ";
827 else
828     std::cout << "Starting ";
829 std::cout << "block with gcd(x," << t << ") = " <<
    g << std::endl;
830 std::cout << "rx in [" << minrx << "," << maxrx <<
    ")," <<
831     << "ry in [" << minry << "," << maxry <<
    "]" << std::endl;
832 std::cout << "sieve bound = " << header.sievebound
    << std::endl;
833
834
835 num_t dg = { g, g };
836 listX.clear(); listY.clear();
837 listX.reserve(NX_LIMIT); listY.reserve(NY_LIMIT);
838 store_target = &listX;
839 ForEachRad(&storeRad, lowradx, maxrx-1, lowrads,
    _limit, t/g, nums_tmp, dg);
840 if (listX.size() == 0) {
841     std::cout << "nx = 0" << std::endl;
842     ENDTIME;
843     std::cout << "done" << std::endl;
844 }
845
846 store_target = &listY;
847 ForEachRad(&storeRad, lowrady, maxry-1, highrads,
    _limit, g, nums_tmp, ONE);
848 PRINTTIME;
849
850 if (skiptriples == 0) {
851     std::cout << "COUNTING" << std::endl;
852     // We first quickly compute the total amount of
    work
853     do_block(header.blocksize, header.sievebound,
    true, skiptriples);
854     for (unsigned int i = 0; i < listX.size(); ++i)
855         listX[i].minus = true;
856     do_block(header.blocksize, header.sievebound,
    true, skiptriples);
857     for (unsigned int i = 0; i < listX.size(); ++i)
858         listX[i].minus = false;

```

Zie pagina 43

Zie pagina 33

Zie pagina 33

Zie pagina 52

Zie pagina 52


```

859         OUT_totalcount = skiptriples;
860         OUT_donecounter = 0;
861         skiptriples = 0;
862     }
863
864     std::cout << "SIEVING" << std::endl;
Zie pagina 52 865     do_block(header.blocksize, header.sievebound, false
        , skiptriples);
866     for (unsigned int i = 0; i < listX.size(); ++i)
867         listX[i].minus = true;
868     do_block(header.blocksize, header.sievebound, false
        , skiptriples);
869     ENDTIME;
870     std::cout << "done" << std::endl;
871 }
872
873 void ABCSieve::setStats(intt totalcount, intt
        donecounter, intt foundcounter, intt trialdivcounter
        )
874 {
875     OUT_totalcount = totalcount;
876     OUT_donecounter = donecounter;
877     OUT_foundcounter = foundcounter;
878     OUT_trialdivcounter = trialdivcounter;
879 }
880
881 void find_triples(abc_sieve_header& DATA, intt
        skiptriples, intt triples_total, intt hits, intt
        trialdivcount, ScanCallback callback)
882 {
883     ABCSieve sieve(DATA, callback);
Zie pagina 58 884     sieve.setStats(triples_total, skiptriples, hits,
        trialdivcount);
Zie pagina 55 885     sieve.find_triples(skiptriples);
Zie pagina 41 886     sieve.printStats();
887 }

```

A.4 HEADER FILES

A.4.1 abc_common.h

```

1 #ifndef ABC_COMMON
2 #define ABC_COMMON
3
4 #include <vector>
5

```

```

6 #define CODE_VERSION 400
7
8 #ifdef _WIN64
9     #define FINTT "%I64u"
10    typedef unsigned __int64 intt;
11 #else
12    #define FINTT "%llu"
13    typedef unsigned long long intt;
14 #endif
15
16 struct wudata {
17     intt a;
18     intt b;
19 };
20
21 struct abc_header {
22     intt alpha;
23     intt beta;
24     intt interval;
25     intt limit;
26 };
27
28 struct abc_sieve_header {
29     intt id;
30     intt lower_rx, upper_rx;
31     intt lower_ry, upper_ry;
32     intt gcdwith, gcdx;
33     intt lower_c_LO, lower_c_HI;
34     intt upper_c_LO, upper_c_HI;
35     unsigned int blocksize, sievebound;
36     unsigned int minQ;
37 };
38
39
40 typedef void (*ScanCallback) (std::vector<wudata>& d,
41     intt scanned, intt total, intt& found, intt
42     trialdivs);
43
44
45
46 #endif

```

A.4.2 abc_sieve.h

```

1 #ifndef ABC_SIEVE_H
2 #define ABC_SIEVE_H
3
4 void find_triples(abc_sieve_header& DATA, intt
    skiptriples, intt triples_total, intt hits, intt
    trialdivcount, ScanCallback callback);
5 unsigned int estimate_sieve_bound(unsigned int Lx,
    unsigned int Ly);
6
7 #endif

```

A.4.3 abc_sieve_util.h

```

1 #ifndef ABCALGO_UTIL_H
2 #define ABCALGO_UTIL_H
3
4 #include <list>
5 #include <vector>
6
7 #include "abc_common.h"
8
9 const unsigned int NUMPERRAD_LIMIT = 300000;
10
11 struct rad_t {
12     unsigned int rad;
13     std::list<unsigned int> primes;
14 };
15
16 struct num_t {
17     intt num;
18     unsigned int rad;
19     bool operator==(const num_t& b) const { return num
    == b.num; }
20 };
21
22 extern const num_t ZERO;
23 extern const num_t ONE;
24
25 bool operator<(const rad_t& r1, const rad_t& r2);
26 bool operator<(const num_t& r1, const num_t& r2);
27
28
29
30 void sieveprimes(unsigned int primelimit,
31     std::vector<unsigned int>& lowprimes);

```

```

32 void sieveradicals(unsigned int radlimit, std::vector<
    rad_t>& rads);
33
34 // Needs primes up to sqrt(start+length) in lowprimes
35 void sieveradicals_interval(intt start, unsigned int
    length,
36     std::vector<rad_t>& rads,
37     std::vector<unsigned int>& lowprimes);
38
39 // Check if two squarefree integers are coprime
40 bool AreRadCoprime(unsigned int u, unsigned int v);
41
42 #if 0
43 inline static int binsearch(num_t* a, int l, int h,
    intt target)
44 {
45     while (l <= h) {
46         int t = (l + h) / 2;
47         if (a[t].num == target) return t;
48         if (a[t].num > target) {
49             h = t-1;
50         } else {
51             l = t+1;
52         }
53     }
54     return -1;
55 }
56 #endif
57
58 // find square root of number if it is a perfect square
59 // if not a square, returns square root rounded down
60 bool issquare(intt n, unsigned int& r);
61 // same, for cube root
62 bool iscube(intt n, unsigned int& r);
63
64 // determine if n is a fifth power of a prime. If true,
    return prime
65 bool is_fifth_ppower(intt n, unsigned int& p);
66 bool is_seventh_ppower(intt n, unsigned int& p);
67
68 typedef bool (*ForEachRadFunc) (num_t n, unsigned int
    rad_index, num_t data);
69
70 // Search all numbers with radical with index from
    index_from (inclusive)
71 // and radical size up to rad_limit (inclusive) and
    size up to num_limit (inc.)
72 // and coprime to coprime_to.
73 // Radicals are taken from the rads[] vector.

```

```

74 // The nums[] argument is used for temporary storage.
75 // If the ForEachRadFunc returns false, the current
    radical is skipped.
76 // Returns the number of radicals considered.
77 intt ForEachRad(ForEachRadFunc f,
78               unsigned int index_from, unsigned int rad_limit
    ,
79               std::vector<rad_t>& rads,
80               intt num_limit, unsigned int coprime_to,
81               std::vector<num_t>& nums,
82               num_t data);
83
84
85 typedef void (*TripleCallback) (intt a, intt b);
86 typedef void (*ScanCallback) (std::vector<wudata>& d,
    intt scanned, intt total, intt& found, intt
    trialdivs);
87
88
89 #endif
90
91
92 #ifdef _WIN32
93 typedef struct { long long quot, rem; } lldiv_t;
94 #endif

```
