Op jacht naar radicalen

Dijk, T. van

OP JACHT NAAR RADICALEN

EEN ALGORITME VOOR DE ENUMERATIE VAN ABC-DRIETALLEN

THIJS VAN DIJK

BACHELORSCRIPTIE
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.

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

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.

Zie appendix A.

1.1 HET ABC-VERMOEDEN

1.1.1 Radicaal

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

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

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

VOORBEELDEN

- $\mathrm{rad}(1.024) = \mathrm{rad}(1.048.576) = \mathrm{rad}(1.073.741.824) = 2$.
- Voor alle $p, q, r \in \mathbb{Z}_{>0}$ geldt: $\mathrm{rad}\left(2^p 3^q 5^r\right) = 30$.
- 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:

$$1 \leqslant \mathrm{rad}(n) \leqslant n,$$

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:

$$\mathrm{rad}(pq) \leqslant \mathrm{rad}(p)\,\mathrm{rad}(q),$$

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:

i $0 < a < b < c$
ii $a + b = c$
iii a, b, en c zijn copriem
iv $\mathrm{rad}(abc) < c$.

VOORBEELDEN

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

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 $3^{2n} \equiv 1 \bmod 8$, 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, $\operatorname{rad}(c) = 3$, en

$$\operatorname{rad}(b) \leqslant \operatorname{rad}(8) \operatorname{rad}\left(\frac{b}{8}\right) \leqslant \frac{b}{4}.$$

Dus geldt,

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

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

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

1.1.3 Kwaliteit

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

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

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 “$\lightning$” 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 rekentijd 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

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

Bij het maken van een rekentijdanalyse 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 rekentijd tot op de dichtstbijzijnde orde af te schatten.

Het gebruik van ordenotaties kan ook inzicht bieden in hoe de rekentijden 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.

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

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** $a \in \{1, \ldots, \frac{1}{2}N\}$ **doe**
2:   **voor** $b \in \{a+1, \ldots, N-a\}$ **doe**
3:     Zet $c := a + b$.
4:     **als** a, b, c copriem en $\mathrm{rad}(abc) < c$ **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 $(= \mathcal{O}\left(N^3\right))$. De factorisatie van een getal van orde $\mathcal{O}\left(N^3\right)$ kan worden uitgevoerd met met complexiteit (in tijd) $\mathcal{O}\left(N^{\frac{3}{2}}\right)$, dus de rekentijd van algoritme is $\mathcal{O}\left(N^{3\frac{1}{2}}\right)$.

Zie sectie 2.1.1

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-drietal* als geldt: $z \in \{x + y, |x - y|\}$, en

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

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

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

en

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

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 $\mathrm{ggd}(r_x, r_y) = 1$ **doe**
3:     **voor** $x \in \{0, \dots, N\}$, met $\mathrm{rad}(x) = r_x$ **doe**
4:       **voor** $y \in \{0, \dots, N\}$, met $\mathrm{rad}(y) = r_y$ **doe**
5:         **voor** $z \in \{x + y, |x - y|\}$ **doe**
6:           Controleer of je een 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:

$$\#\left\{ (x,y) \;\middle|\; \begin{array}{c} x, y \text{ copriem} \\ \mathrm{rad}(x)\,\mathrm{rad}(y)^2 < N \end{array} \right\} = \mathcal{O}(N^{\frac{2}{3}+\varepsilon}).$$

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:

$$\int_1^{\pi(\sqrt{z})} \frac{\log z}{\log p} dp < \log z \int_1^{\pi(\sqrt{z})} 1 \, dp = \log z \cdot \pi\left(\sqrt{z}\right)$$

$$= \log z \cdot \mathcal{O}\left(\frac{\sqrt{z}}{\log \sqrt{z}}\right) = \mathcal{O}\left(\log z \cdot \frac{z^{\frac{1}{2}}}{\frac{1}{2}\log z}\right) = \mathcal{O}\left(z^{\frac{1}{2}}\right).$$

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.

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

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).

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 $r \in [m, M)$.

Laat $L := (1, 1, \ldots, 1)$ een lijst van $M - m$ enen zijn.
Laat $I := \{0, \ldots, M - m - 1\}$.
voor $p \in P$ **doe**
  voor $i \in I$ met $p^2 \mid (m + i)$ **doe**
    Zet $L_i := 0$.
  einde voor
einde voor
resultaat $\{(m + i) \, \forall i \in I : L_i \neq 0\}$

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 $m \leqslant r < M$ 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.)

Laat $L := (1, 1, \ldots, 1)$ een lijst van $M - m$ enen zijn, en $R := (\emptyset, \emptyset, \ldots, \emptyset)$ een lijst van $M - m$ (vooralsnog lege) verzamelingen.
Laat $I := \{0, \ldots, M - m - 1\}$.
voor $p \in P$ **doe**
  voor $i \in I$ met $p \mid (m + i)$ **doe**
    als $L_i \neq 0$ **dan**
      Zet $L_i := p \cdot L_i$.
      Voeg p toe aan R_i.
    einde als
  einde voor
  voor $i \in I$ met $p^2 \mid (m + i)$ **doe**
    Zet $L_i := 0$.
  einde voor
einde voor
voor $i \in I$ met $L_i \neq 0$ **doe**
  als $L_i \neq m + i$ **dan**
    Voeg $\frac{m+i}{L_i}$ toe aan R_i.
  einde als
einde voor
resultaat $\{(m + i, R_i) \, \forall i \in I : L_i \neq 0\}$

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 $r, M \in Z$, 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 $r > M$. $\lightning$

Dus $\frac{r}{L}$ is priem. $\square$

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 rekentijd 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.)

Laat $N := \{r\}$ een lijst zijn. Laat $i := 0$.
zolang $i < \#N$ **doe**
  voor $p \in P$ **doe**
    als $p \cdot N_i < M$ **dan**
      Plak $p \cdot N_i$ achteraan N.
    einde als
  einde voor
  Zet $i := i + 1$.
einde zolang
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

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

in andere woorden,

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

We hebben dus een goedgedefinieerd interval waar $\mathrm{rad}(z)$ in moet vallen om samen met x en y een ABC-drietal te maken. (We wisten immers nog dat $\mathrm{rad}(y) < \mathrm{rad}(z)$.) 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 $\mathrm{rad}(n) > p^2$.
BEWIJS Van het tegendeel (n is een samengesteld p-ruw getal met een radicaal kleiner dan p^2, en *geen* priemmacht) is makklijk 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 $n \geqslant q_1 q_2^2 > p^3$. ↯
Dus n is een priemkwadraat, of n is kwadraatvrij. □

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 priemen 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*.)

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

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

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.)

Zet $r := 1, l := z$.
voor $p \in P$ **doe** — *In volgorde.*
  als $p \mid l$ **dan**
    Zet $l := \frac{z}{p}$, $r := p \cdot r$.
    zolang $p \mid l$ **doe**
      Zet $l := \frac{l}{p}$.
    einde zolang
  einde als
  als $r \cdot p^2 > M$ **dan**
    Er is voldaan aan de stopconditie uit stelling A. Rest ons alleen te kiezen uit één van de twee gevallen.
    Voer een machtstest uit op l.
    als $\exists j, k \in \mathbb{Z} : l = k^j$ **dan**
      Merk op, k *is priem, dus* $\text{rad}(z) = r \cdot k$.
      resultaat $(r_y < rk < M)$.
    else
      Merk op, $\text{rad}(z) > r \cdot p^2$.
      resultaat NEEN.
    einde als
  einde als
  als $p^3 > l$ **dan**
    Er is voldaan aan de stopconditie uit stelling B. Wederom hoeven we alleen te kiezen uit één van de twee gevallen.
    Voer een kwadraattest uit op l.
    als $\exists k \in \mathbb{Z} : l = k^2$ **dan**
      Merk op, k *is priem, dus* $\text{rad}(z) = r \cdot k$.
      resultaat $(r_y < rk < M)$.
    else
      Merk op, l *is kwadraatvrij, dus* $\text{rad}(z) = r \cdot l$.
      resultaat $(r_y < rl < M)$.
    einde als
  einde als
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 priemen 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 rekentijd dure grap. In veel gevallen zal voor lage p het p-gladde deel van $\mathrm{rad}(z)$ de bovengrens al overstijgen.

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

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 $i \in \{0, \ldots, \#N^x - 1\}$ geldt $\mathrm{rad}(N^x_i) = R^x_i$; 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 $\mathrm{rad}(z)$ kleiner is dan $\frac{\max\{x,y,z\}}{\mathrm{rad}(x)\,\mathrm{rad}(y)}$.

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

Laat $I^x := \{0, \ldots, \#N^x - 1\}$, en $I^y := \{0, \ldots, \#N^y - 1\}$.
Laat Z een $(\#N^x) \times (\#N^y)$–matrix zijn.
voor $p \in P$ **doe**
 Laat $Q := \{\emptyset, \emptyset, \ldots, \emptyset\}$ een lijst van p (lege) lijsten zijn.
 voor $i_x \in I^x$ **doe**
  Noem $x := N^x_{i_x}$.
  Noem $r := x \mod p$.
  als $r = 0$ **dan** ga door met de volgende i_x.
  Voeg i_x toe aan Q_r.
 einde voor
 voor $i_y \in I^y$ **doe**
  Noem $y := N^y_{i_y}$.
  Noem $r := y \mod p$.
  als $r = 0$ **dan** ga door met de volgende i_y.
  voor $i_x \in Q_{p-r}$ **doe**
   Noem $x := N^x_{i_x}$.
   Zet $Z_{i_x,i_y} := pZ_{i_x,i_y}$.
  einde voor
 einde voor
einde voor
voor $i_x, i_y \in I^x \times I^y$ **doe**
 Noem $x := N^x_{i_x}$, $y := N^y_{i_y}$.
 Noem $r_x := R^x_{i_x}$, $r_y := R^y_{i_y}$.
 als $r_x > r_y$ **dan** ga door met het volgende paar i_x, i_y.
 als $\mathrm{ggd}(r_x, r_y) \neq 1$ **dan** ga door met de volgende i_x, i_y.
 als $r_y < Z_{i_x,i_y} < \frac{x+y}{r_x r_y}$ **dan**
  print (x, y, z)
 einde als
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 overtollige 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 (zogeheten *"work units"*) op te delen die:

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

Het hergebruiken van de eerder genoemde subblokken voldoet uiteraard niet aan eis 2: de "centrale" computer (die alles coördineert) zou eerst in $\mathcal{O}\left(N^{\frac{2}{3}+\varepsilon}\right)$ 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 $\mathrm{rad}(x)$ resp. $\mathrm{rad}(y)$ 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 $\mathrm{ggd}(x, y) = 1$) 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 $\mathrm{ggd}(x, y) = 1$. 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 $\mathrm{rad}(abc) = \mathrm{rad}(xyz) < c^{\frac{1}{2}+\frac{1}{3}+\frac{1}{7}} = c^{\frac{41}{42}} < c$. Dus (a, b, c) is een ABC-drietal.

Voor de kwaliteit is bovendien een ondergrens:

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

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

```
#include <iostream>
#include <time.h>
#include <vector>
#include <cstdlib>
#include <cstdio>

#include "abc_common.h"
#include "abc_sieve.h"

#ifdef _MSC_VER
#define atoll _atoi64
#include "str_util.h"
#endif

#define VERBOSE

typedef unsigned long long intt;

intt META_hits;
intt META_trialdivcount;
intt META_triples_total;


void callback(std::vector<wudata>& d, intt triples_done, intt triples_total, intt &hits, intt trialdivcount)
{
    hits += d.size();

#ifdef VERBOSE
#if 0
    for (size_t i = 0; i < d.size(); ++i)
```

Zie pagina 58

```
        printf(FINTT " + " FINTT " = " FINTT "\n", d[i].a, d[i].b, d[i].a+d[i].b);
#endif
    printf("progress: %f\n", double(triples_done)/double(triples_total));
#endif

    d.clear();

    META_hits = hits;
    META_trialdivcount = trialdivcount;
    META_triples_total = triples_total;
}

void run_workunit(abc_sieve_header h)
{
    intt triples_total = 0;
    intt triples_done = 0;
    intt hits = 0;
    intt trialdivcount = 0;

    find_triples(h, triples_done, triples_total, hits, trialdivcount, &callback);
    printf("finished with hits=" FINTT "\n", META_hits);
}

int main(int argc, char **argv)
{
    abc_sieve_header h;

    if (argc < 10 || argc > 11) {
        fprintf(stderr, "bad command line: expected\nlower_rx upper_rx lower_ry upper_ry gcd_with gcd_rx lower_c upper_c blocksize [sievebound]");
        return 1;
    }
    h.id = 0;
    h.lower_rx = atoll(argv[1]);
    h.upper_rx = atoll(argv[2]);
    h.lower_ry = atoll(argv[3]);
    h.upper_ry = atoll(argv[4]);
    h.gcdwith = atoll(argv[5]);
    h.gcdx = atoll(argv[6]);
    h.lower_c_LO = atoll(argv[7]);
    h.lower_c_HI = 0;
    h.upper_c_LO = atoll(argv[8]);
    h.upper_c_HI = 0;
```

```
    h.blocksize = atoi(argv[9]);
    if (argc < 11)
        h.sievebound = estimate_sieve_bound(h.lower_rx, h.lower_ry);
    else
        h.sievebound = atoi(argv[10]);

    h.minQ = 0x01000000UL;

    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",
           h.id, h.lower_rx, h.upper_rx, h.lower_ry, h.upper_ry, h.gcdwith,
           h.gcdx, h.lower_c_LO, h.upper_c_LO, h.blocksize, h.sievebound);

    run_workunit(h);
    return 0;
}
```

Zie pagina 42

Zie pagina 28

A.2 ABC_SIEVE_UTIL.CPP

```
#include "abc_sieve_util.h"
#include <cassert>

const num_t ZERO = { 0, 0 };
const num_t ONE = { 1, 1 };

#if 0
bool operator<(const rad_t& r1, const rad_t& r2)
{
    return r1.rad < r2.rad;
}

bool operator<(const num_t& r1, const num_t& r2)
{
    return r1.num < r2.num;
}
#endif


// Find primes up to primelimit. (Fast.)
void sieveprimes(unsigned int primelimit,
                 std::vector<unsigned int>& lowprimes)
```

```
{
    bool* s = new bool[primelimit];
    unsigned int primecount = 0;
    for (unsigned int i = 0; i < primelimit; ++i)
        s[i] = true;

    for (unsigned int i = 2; i < primelimit; ++i) {
        if (s[i]) {
            primecount++;
            for (unsigned int j = i+i; j < primelimit; j += i)
                s[j] = false;
        }
    }

    lowprimes.resize(primecount);

    primecount = 0;
    for (unsigned int i = 2; i < primelimit; ++i)
        if (s[i])
            lowprimes[primecount++] = i;

    delete[] s;
}

#if 0
// Find squarefrees up to radlimit. (Fast, but O~(radlimit) memory )
void sieveradicals(unsigned int radlimit, std::vector<rad_t>& rads)
{
    rad_t* r = new rad_t[radlimit];
    unsigned int radcount = 0;
    for (unsigned int i = 0; i < radlimit; ++i)
        r[i].rad = 1;

    for (unsigned int i = 2; i < radlimit; ++i) {
        if (r[i].rad == 1) {
            for (unsigned int j = i; j < radlimit; j += i) {
                r[j].rad *= i;
                r[j].primes.push_back(i);
            }
        }
        if (r[i].rad == i) radcount++;
    }
    rads.resize(radcount+1);
    radcount = 0;
    rads[radcount++] = r[1];
```

```
    for (unsigned int i = 2; i < radlimit; ++i)
        if (r[i].rad == i) {
            rads[radcount++] = r[i];
        }
    delete[] r;
}
#endif

// Find squarefrees in interval. Fairly fast, memory linear in interval length
// Needs primes up to sqrt(start+length) in lowprimes
void sieveradicals_interval(intt start, unsigned int length,
                std::vector<rad_t>& rads,
                std::vector<unsigned int>& lowprimes)
{
    rad_t* r = new rad_t[length];
    unsigned int radcount = 0;
    for (unsigned int i = 0; i < length; ++i)
        r[i].rad = 1;
    if (start == 0)
        r[0].rad = 0;

    unsigned int i = 0;
    unsigned int p = lowprimes[i];
    while (((intt)p)*p <= start+length) {
        intt q = 1;

        intt plim = (start+length)/p;

        while (q <= plim && q <= p) {
            q *= p;
            intt rem = start % q;
            if (rem == 0) rem = q;

            for (intt j = q - rem; j < length; j += q) {
                if (p == q) {
                    r[j].primes.push_back(p);
                    r[j].rad *= p;
                } else {
                    r[j].rad = 0;
                }
            }
        }
        p = lowprimes[++i];
    }

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

Hij ontkent overigens alles.

```
            if (r[i].rad != 0) radcount++;
        }

        rads.resize(radcount);
        radcount = 0;
        for (i = 0; i < length; ++i) {
            if (r[i].rad != 0) {
                assert( (start+i) % r[i].rad == 0);
                intt lastp = (start + i) / r[i].rad;
                if (lastp != 1)
                    r[i].primes.push_back((start + i) / r[i].rad);
                r[i].rad = start + i;
                rads[radcount++] = r[i];
            }
        }

        delete[] r;
}

// Check if two squarefree integers are coprime.
// Based on code by Arjen Lenstra <arjen.lenstra@epfl.ch>, supplied by
// Alexander Petric <alexandre.petric@epfl.ch> and
// Bos Joppe <joppe.bos@epfl.ch>.
bool AreRadCoprime(unsigned int u, unsigned int v)
{
    bool u_2 = false;
    bool v_2 = false;

    // get rid of factor 2
    if (!(u & 1)) {
        u_2 = true;
        u >>= 1;
    }
    if (!(v & 1)) {
        v_2 = true;
        v >>= 1;
    }

    // if both are even
    if (u_2 & v_2)
        return false;

    if (v == 1 || u == 1)
        return true;

    // now both u and v are odd
    for (;;) {
```

```
        if (u == v)
            return (u == 1);

        if (u > v) {
            u -= v;
            while (!(u&3)) u >>= 2;
            if (!(u & 1))  u >>= 1;
        } else {
            v -= u;
            while (!(v&3)) v >>= 2;
            if (!(v & 1))  v >>= 1;
        }
    }
}


// Search all numbers with radical with index from index_from (inclusive)
// and radical size up to rad_limit (inclusive) and size up to num_limit (inc.)
// and coprime to coprime_to.
// Radicals are taken from the rads[] vector.
// The nums[] argument is used for temporary storage.
// Returns the number of radicals considered.
intt ForEachRad(ForEachRadFunc f,
        unsigned int index_from, unsigned int rad_limit,
        std::vector<rad_t>& rads,
        intt num_limit, unsigned int coprime_to,
        std::vector<num_t>& nums,
        num_t data)
{
    nums.resize(NUMPERRAD_LIMIT);// max. number of ints with given radical

    assert(num_limit >= rad_limit);

    unsigned int radsize = rads.size();
    intt doneradcount = 0;

    for (unsigned int i = index_from; i < radsize; ++i) {
        rad_t& r = rads[i];

        if (r.rad > rad_limit) break;

        if (coprime_to > 1 && !AreRadCoprime(coprime_to, r.rad)) continue;

```

Zie pagina 32

```
        ++doneradcount;

        num_t n;
        n.num = r.rad;
        n.rad = r.rad;

        bool ret = (*f)(n, i, data);
        if (!ret) continue;

        unsigned int count = 0;
        nums[count++] = n;

        std::list<unsigned int>::iterator piter;

        bool breakloop = false;

        for (piter = r.primes.begin(); piter != r.primes.end() && !breakloop; ++piter) {
            unsigned int p = *piter;
            intt plim = num_limit / p;
            for (unsigned int k = 0; k < count; ++k) {
                intt a = nums[k].num;
                if (a <= plim) {
                    a *= p;
                    n.num = a;
                    nums[count++] = n;
                    assert(count < NUMPERRAD_LIMIT);
                    assert(nums.size() == NUMPERRAD_LIMIT);

                    ret = (*f)(n, i, data);
                    // stop processing this radical
                    // if requested by f
                    if (!ret) { breakloop = true; break; }
                }
            }
        }
    }

    return doneradcount;
}


// find square root of number if it is a perfect square
// if not a square, returns square root rounded down
// TODO: this can probably be done much faster
bool issquare(intt n, unsigned int& r)
{
```

```
    unsigned int lo = 0;
    unsigned int hi = 4294967295ULL; // 2^32-1
    //unsigned int hi = 1000000000; // 10^9
    while (lo + 1 < hi) {
        unsigned int v = lo/2 + hi/2 + ((lo&1)+(hi&1))/2;
        if (((intt)v)*v < n)
            lo = v;
        else
            hi = v;
    }

    if (((intt)lo)*lo == n) {
        r = lo;
        return true;
    }

    if (((intt)hi)*hi == n) {
        r = hi;
        return true;
    }

    r = hi-1;

    return false;
}
bool iscube(intt n, unsigned int& r)
{
    unsigned int lo = 0;
    unsigned hi = 2642245; // floor((2^64-1)^(1/3))
    //unsigned hi = 1000000; // 10^6
    while (lo + 1 < hi) {
        unsigned int v = (lo+hi)/2;
        if (((intt)v)*v*v < n)
            lo = v;
        else
            hi = v;
    }

    if (((intt)lo)*lo*lo == n) {
        r = lo;
        return true;
    }

    if (((intt)hi)*hi*hi == n) {
        r = hi;
        return true;
    }

```

```
    r = hi-1;

    return false;
}

struct ppower_t {
  unsigned int p;
  intt pk;
};

bool is_fifth_ppower(intt n, unsigned int& p)
{
#include "5thpp.h"
  int l = 0;
  int h = sizeof(cand)/sizeof(cand[0])-1;
  while (l+1 < h) {
    unsigned int v = (l+h)/2;
    if (cand[v].pk < n)
      l = v;
    else
      h = v;
  }

  if (cand[l].pk == n) {
    p = cand[l].p;
    return true;
  }
  if (cand[h].pk == n) {
    p = cand[h].p;
    return true;
  }

  return false;
}

bool is_seventh_ppower(intt n, unsigned int& p)
{
#include "7thpp.h"
  int l = 0;
  int h = sizeof(cand)/sizeof(cand[0])-1;

  while (l+1 < h) {
    unsigned int v = (l+h)/2;
    if (cand[v].pk < n)
      l = v;
    else
      h = v;
  }

```

```
  if (cand[l].pk == n) {
    p = cand[l].p;
    return true;
  }
  if (cand[h].pk == n) {
    p = cand[h].p;
    return true;
  }

  return false;
}
```

A.3 ABC_SIEVE.CPP

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <cstdio>
#ifndef _MSC_VER
#include <sys/time.h>
#endif
#include "abc_common.h"
#include "abc_sieve_util.h"

#ifdef __x86_64__
#define ASM64
#endif
#ifdef WIN64
#define ASM64
#endif
#define STATS
#define NDEBUG

const unsigned int NX_LIMIT = 400000;
const unsigned int NY_LIMIT = 1600000;
const unsigned int P_LIMIT = 200000;
const unsigned int BLOCK_LIMIT = 10000;

#include <cassert>
#include <cmath>

struct Index {
    intt num;
    unsigned int rad;
    bool minus;
```

```
    bool operator< (const Index& other) const { return num < other.num; }
};

struct Entry {
    intt left;
    intt rad;
};


class ABCSieve {
public:
    ABCSieve(abc_sieve_header& h, ScanCallback cb);
    void printStats();
    void find_triples(intt skiptriples);
    void setStats(intt totalcount, intt donecounter, intt foundcounter, intt trialdivcounter);


private:
    bool trial_div(intt n, intt left, intt rad, intt maxrad, intt minrad, unsigned int p_index);
    void prepare_buckets(unsigned int start, unsigned int n, unsigned int ip);
    void sieve_primes_bucket(unsigned int startX, unsigned int startY,
                             unsigned int nx, unsigned int ny,
                             unsigned int limit, unsigned int& index);
    intt get_num(unsigned int ix, unsigned int iy) const {
        if (listX[ix].minus)
            return listY[iy].num + listX[ix].num;
        if (listX[ix].num >= listY[iy].num)
            return listX[ix].num - listY[iy].num;
        else
            return listY[iy].num - listX[ix].num;
    }
    void do_block(int blocksize, int sieve_bound, bool count, intt& skip_triples);
    void finish_block(unsigned int startX, unsigned int startY,
                      unsigned int nx, unsigned int ny, unsigned int p_index);

    bool rad_pp_gt_maxrad(unsigned int p, intt pp, intt rad, intt left, intt minrad, intt maxrad);

```

Zie pagina 40
Zie pagina 41
Zie pagina 55
Zie pagina 58
Zie pagina 45
Zie pagina 48
Zie pagina 49
Zie pagina 52
Zie pagina 50
Zie pagina 43

```
    ScanCallback callback;
    abc_sieve_header header;
    unsigned int minrx, minry, maxrx, maxry;

    // Storage
    std::vector<unsigned int> lowprimes;
    std::vector<rad_t> lowrads;
    std::vector<rad_t> highrads;
    std::vector<num_t> nums_tmp;
    std::vector<Entry> block;
    std::vector<int> bucketX;
    std::vector<int> nodesx;

    std::vector<Index> listX;
    std::vector<Index> listY;

#ifndef _MSC_VER
    // Timing
    struct timeval nowT, startT;
#endif

    // Output
    std::vector<wudata> OUT_foundtriples;
    unsigned long long OUT_totalcount;
    unsigned long long OUT_donecounter;
    unsigned long long OUT_foundcounter;
    unsigned long long OUT_trialdivcounter;


#ifdef STATS
    intt stat_totalblocksize;
    intt stat_doneblocksize;
    intt stat_sieve_bucket_marks;
    intt stat_sieve_modY;
    intt stat_sieve_modX;
    intt stat_sieve_marks;
    intt stat_notcoprime;
    intt stat_notcoprime_repeat;
    intt stat_rxryrytoolarge;
    intt stat_rxryrztoolarge;
    intt stat_trialdiv_reached;
    intt stat_trialdiv_start;
    intt stat_trialdiv_smooth;
    intt stat_trialdiv_end;
    intt stat_trialdiv_smallescape;
    intt stat_trialdiv_escape;
    intt stat_trialdiv_power;
    intt stat_trialdiv_factors;
    intt stat_trialdiv_rounds;
```

```
    intt stat_trialdiv_endsquare;
#endif
};

ABCSieve::ABCSieve(abc_sieve_header& h, ScanCallback cb) {
    header = h;
    callback = cb;

    OUT_totalcount = 0;
    OUT_donecounter = 0;
    OUT_foundcounter = 0;
    OUT_trialdivcounter = 0;

#ifdef STATS
    stat_totalblocksize = 0;
    stat_doneblocksize = 0;
    stat_sieve_bucket_marks = 0;
    stat_sieve_modY = 0;
    stat_sieve_modX = 0;
    stat_sieve_marks = 0;
    stat_notcoprime = 0;
    stat_notcoprime_repeat = 0;
    stat_rxryrytoolarge = 0;
    stat_rxryrztoolarge = 0;
    stat_trialdiv_reached = 0;
    stat_trialdiv_start = 0;
    stat_trialdiv_smooth = 0;
    stat_trialdiv_end = 0;
    stat_trialdiv_smallescape = 0;
    stat_trialdiv_escape = 0;
    stat_trialdiv_power = 0;
    stat_trialdiv_factors = 0;
    stat_trialdiv_rounds = 0;
    stat_trialdiv_endsquare = 0;
#endif

    sieveprimes(2105000, lowprimes); // 2^21 + a bit
//  init_Two64divp(lowprimes.size());
//  init_prime_inv(lowprimes.size());

//  sieveradicals(200000, lowrads);
//  sieveradicals(200000, highrads);
//  sieveradicals_interval(80000000ULL, 10000, highrads, lowprimes);

    nums_tmp.reserve(80000);

    bucketX.resize(P_LIMIT);
```

Zie pagina 29

```
    nodesx.resize(BLOCK_LIMIT);
}

void ABCSieve::printStats() {
#ifdef STATS
    printf("total blocksize:%14lld\n",
        stat_totalblocksize);
    printf("processed size:%15lld\n",
        stat_doneblocksize);
    printf("sieve b. marked:%14lld\n",
        stat_sieve_bucket_marks);
    printf("sieve b. X mod p:%13lld\n", stat_sieve_modX
        );
    printf("sieve b. Y mod p:%13lld\n", stat_sieve_modY
        );
    printf("sieve marked:    %13lld\n",
        stat_sieve_marks);
    printf("rx ry^2 > c:     %13lld\n",
        stat_rxryrytoolarge);
    printf("rx ry rz' > c:   %13lld\n",
        stat_rxryrztoolarge);
    printf("not coprime:     %13lld\n", stat_notcoprime
        );
    printf("not coprime rep: %13lld\n",
        stat_notcoprime_repeat);

    printf("reached trialdiv:%13lld\n",
        stat_trialdiv_reached);
    printf("start of trialdiv:%12lld\n",
        stat_trialdiv_start);
    printf("smooth trialdiv: %13lld\n",
        stat_trialdiv_smooth);
    printf("early exit trdiv:%13lld\n",
        stat_trialdiv_escape);
    printf("early exit power:%13lld\n",
        stat_trialdiv_power);
    printf("early exit small:%13lld\n",
        stat_trialdiv_smallescape);
    printf("trialdiv factors:%13lld\n",
        stat_trialdiv_factors);
    printf("trialdiv rounds:%14lld\n",
        stat_trialdiv_rounds);
    printf("end of trialdiv: %13lld\n",
        stat_trialdiv_end);
    printf("end square:      %13lld\n",
        stat_trialdiv_endsquare);
#endif
}

```

```
#include "prime_inv.h"
#include "prime_inv2.h"
//#include "tables.h"

#include "asm.h"

#ifdef ASM64
#include "invH.h"
#endif

#ifndef _MSC_VER
#define STARTTIME do { gettimeofday(&startT, NULL); } while(0)
#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)
#else
#define STARTTIME
#define ENDTIME
#endif
#ifdef STATS
#define PRINTTIME ENDTIME
#define INC_STAT(s) do { (s)++; } while(0)
#else
#define PRINTTIME
#define INC_STAT(s)
#endif



unsigned int estimate_sieve_bound(unsigned int Lx, unsigned int Ly)
{
    intt r = Lx;
    r *= Ly;
    unsigned int S;

    if (r < 268435456) {
        S = 61000 + (unsigned int)(15000 * (28 - log((double)r) / log(2.0) ));
    } else {
        S = (unsigned int)sqrt((double)(1000000000000000000ULL / r));
    }
    if (S > 200000) S = 200000;
    return S;
}

std::vector<Index>* store_target;
```

```
bool storeRad(num_t n, unsigned int, num_t data)
{
    if (n.rad % data.rad) return false;
    Index a;
    a.num = n.num;
    a.rad = n.rad;
    a.minus = false;
    store_target->push_back(a);
    return true;
}

// Find index of smallest element in a sorted rad_t vector with value >= target
static int binsearch(const std::vector<rad_t>& a, intt target)
{
    int l = 0;
    int h = a.size()-1;
    while (l <= h) {
        int t = (l + h) / 2;
        if (a[t].rad == target) return t;
        if (a[t].rad > target) {
            h = t-1;
        } else {
            l = t+1;
        }
    }
    // not found, so return smallest index with value above target
    return l;
}


// Input: (Implicit:) n - number to factor
//         left - the factor of n coprime to all primes < p
//         rad  - the radical of (n/left)
//         pp   - p*p
// Assumption: rad * p * p > maxrad
// Assumption: p > 59
// Output: rad(n) <= maxrad && rad(n) > minrad

bool ABCSieve::rad_pp_gt_maxrad(unsigned int p, intt /*pp*/, intt rad, intt left, intt minrad, intt maxrad)
{
    // now either:
    // A. left has one prime divisor:
    //    left is a prime power and rad(n) is known
```

```
    // B. left has >=2 prime divisors:
    //    r(left) > p*p and rad(n) > maxrad

    if (p < 300)
        INC_STAT(stat_trialdiv_smallescape);


    // Check if left is a prime-power:
    // left is not divisible by primes <= 59
    // (since we just tested the first 17)
    // 61^10 < 10^18 < 61^11
    // So: only check 2nd, 3rd, 5th, 7th powers
    unsigned int cuberoot, root;
    unsigned int m63 = left % 63;
    if ((m63 == 1 || m63 == 8 || m63 == 55 || m63 ==
        62) && iscube(left, cuberoot)) {
        INC_STAT(stat_trialdiv_power);
        // third power; maybe 6th, 9th?
        left = cuberoot;
        m63 = left % 63;
        if ((m63 == 1 || m63 == 8 || m63 == 55 || m63
            == 62) && iscube(left, cuberoot)) {
            left = cuberoot; // 9th power
        } else if (((left & 7) == 1) && (m63 == 1 ||
            m63 == 4 || m63 == 16 || m63 == 22 || m63 ==
             25 || m63 == 37 || m63 == 43 || m63 == 46
            || m63 == 58) && issquare(left, root)) {
            left = root; // 6th power
        }
        // Now: left is a prime or
        // left >= rad(left) > p^2
        return (rad*left <= maxrad && rad*left > minrad
            );
    }
    if (((left & 7) == 1) && (m63 == 1 || m63 == 4 ||
        m63 == 16 || m63 == 22 || m63 == 25 || m63 == 37
         || m63 == 43 || m63 == 46 || m63 == 58) &&
        issquare(left, root)) {
        INC_STAT(stat_trialdiv_power);
        left = root;
        // square; maybe 4th, 8th, 10th
        if (issquare(left, root)) {
            left = root; // 4th power
            if (issquare(left, root)) {
                left = root; // 8th power
            }
        } else if (left == 844596301) {
            // the only possible 10th powers are
                {61,67,71,73}^10
```

Zie pagina 35
Zie pagina 35
Zie pagina 34
Zie pagina 34
Zie pagina 34
Zie pagina 34

```
            left = 61;
        } else if (left == 1350125107) {
            left = 67;
        } else if (left == 1804229351) {
            left = 71;
        } else if (left == 2073071593) {
            left = 73;
        }
        // Now: left is a prime or
        // left >= rad(left) > p^2
        return (rad*left <= maxrad && rad*left > minrad
           );
    }
    unsigned int m11 = left % 11;
    if ((m11 == 1 || m11 == 10) && is_fifth_ppower(left
        ,root)) {
        INC_STAT(stat_trialdiv_power);
        return (rad*root <= maxrad && rad*root > minrad
           );
    }
    unsigned int m29 = left % 29;
    if ((m29 == 1 || m29 == 12 || m29 == 17 || m29 ==
        28) && is_seventh_ppower(left, root)) {
        INC_STAT(stat_trialdiv_power);
        return (rad*root <= maxrad && rad*root > minrad
           );
    }

    INC_STAT(stat_trialdiv_escape);
    return (rad*left <= maxrad && rad*left > minrad);
}

// Do a trial division by primes from lowprimes,
    starting at index p_index
// Input: n - number to factor
//        left - n divided by the p-smooth part of n (p
     = lowprimes[p_index-1]
//        rad  - radical of the p-smooth part of n
// Output: rad(n) <= maxrad && rad(n) > minrad

bool ABCSieve::trial_div(intt n, intt left, intt rad,
    intt maxrad, intt minrad, unsigned int p_index)
{
    INC_STAT(stat_trialdiv_reached);

#ifndef NDEBUG
    intt _n = n;
    intt _left = left;
    intt _rad = rad;
```

Zie pagina 36

Zie pagina 36

```
    intt _maxrad = maxrad;
#else
    (void)n; // n is used in debug mode only
#endif

    assert(left);

    if (rad > maxrad) {
        INC_STAT(stat_trialdiv_start);
        return false;
    }
    if (left == 1) {
        INC_STAT(stat_trialdiv_start);
        return (rad <= maxrad && rad > minrad);
    }
    unsigned int p = lowprimes[p_index];
#ifndef NDEBUG
    if (left < p) {
        printf("error: %llu < %d\n (%llu, %llu, %llu, %
            llu)", left, p, _n, _left, _rad, _maxrad);
        assert(left >= p);
    }
#endif
    if (p > maxrad) {
        INC_STAT(stat_trialdiv_start);
        return false;
    }

    // handle p == 2 separately
    if (p_index == 0) {
        assert(rad == 1);

        if (!(left & 1)) {
            rad = 2;
            do {
                left >>= 1;
            } while (!(left & 1));
        }
        p = lowprimes[++p_index];
    }

    OUT_trialdivcounter++;

    while (true) {
        for (int i = 0; i < 16; ++i) {
            INC_STAT(stat_trialdiv_rounds);
            intt t = left * prime_inv[p_index];
            if (t < Two64divp[p_index]) {
                INC_STAT(stat_trialdiv_factors);
```

```c
                rad *= p;
                do {
                    left = t;
                    t = t * prime_inv[p_index];
                } while (t < Two64divp[p_index]);
            }
            p = lowprimes[++p_index];
        }

        if (left == 1) {
            INC_STAT(stat_trialdiv_smooth);
            return (rad <= maxrad && rad > minrad);
        }
#ifndef NDEBUG
        if (left < p) {
            printf("error: %llu >= %d\n (%llu, %llu, %llu, %llu)", left, p, _n, _left, _rad, _maxrad);
            assert(left >= p);
        }
#endif

        intt pp = p;
        pp *= p;

        // p is large enough to reach a conclusion
        if (rad*pp > maxrad)
            return rad_pp_gt_maxrad(p,pp,rad,left, minrad,maxrad);

        if (pp*p > left) {
            INC_STAT(stat_trialdiv_end);
            // now either:
            // A. left is the square of a prime
            // B. left is squarefree

            unsigned int root;
            if (((left & 7) == 1) && issquare(left, root)) {
                INC_STAT(stat_trialdiv_endsquare);
                return (rad*root <= maxrad && rad*root > minrad);
            }

            return (rad*left <= maxrad && rad*left > minrad);
        }
    }

```

Zie pagina 43

```
    assert(false);
}


void ABCSieve::prepare_buckets(unsigned int start, unsigned int n, unsigned int ip)
{
    assert(n < BLOCK_LIMIT);

    // Sort listX[start] up to listX[start+n-1] into buckets mod p.

    unsigned long long p = lowprimes[ip];
#ifdef ASM64
    unsigned char s = InvT[ip].s;
    unsigned long long q = InvT[ip].q;
#endif


    // bucketX + nodesx form a set of linked lists.
    // bucketX[i] points to the head node of list i, or -1 if the list is empty.
    // nodesx[i] is the node following node i, or -1 for the tail of a list.

    assert(p <= P_LIMIT);
    for (unsigned i = 1; i < p; ++i)
        bucketX[i] = -1;
    for (unsigned int i = 0; i < n; ++i) {
        unsigned int modp;
        MOD(modp,listX[start+i].num,p,q,s);
//      std::cout << l[i].num << " % " << p << " == " << (l[i].num % p) << " != " << modp << " ( q = " << q << ", s = " << (unsigned int)s << " )" << std::endl;
        assert(modp == listX[start+i].num % p);
        INC_STAT(stat_sieve_modX);
        if (!modp) continue;
        if (listX[start+i].minus) modp = p - modp;
        nodesx[i] = bucketX[modp];
        assert(bucketX[modp] == -1 || (bucketX[modp] >= 0 && (unsigned int)bucketX[modp] < n));
        bucketX[modp] = i;
    }
}


// Sieve through block with x's in listX at indices [startX,startX+nx) and
```

```
// y's in listY at indices [startY,startY+ny)

// Divide by primes in lowprimes, starting at index, and continuing with all
// primes < limit. Afterwards the parameter index is the index of the first
// prime not yet handled.

// Result: for every entry e in the block of (x,y) pairs,
// e.left and e.rad are updated for all primes p above.

void ABCSieve::sieve_primes_bucket(unsigned int startX, unsigned int startY,
                                   unsigned int nx, unsigned int ny,
                                   unsigned int limit, unsigned int& index)
{
    unsigned int ip;
    for (ip = index; true; ++ip) {
        unsigned int p = lowprimes[ip];
        if (p > limit) break;
        intt pinv = prime_inv[ip];
        intt pmax = Two64divp[ip];
#ifdef ASM64
        unsigned char s = InvT[ip].s;
        unsigned long long q = InvT[ip].q;
#endif
        //std::cout << p << std::endl;
        prepare_buckets(startX, nx, ip);

        for (unsigned int iy = 0; iy < ny; ++iy) {
            unsigned int modp;
            MOD(modp,listY[startY+iy].num,p,q,s);
            assert(listY[startY+iy].num % p == modp);
            INC_STAT(stat_sieve_modY);
            if (!modp) continue;
            for (int ix = bucketX[modp]; ix >= 0; ix = nodesx[ix]) {
                assert((unsigned int)ix < nx);
                //printf("modp=%s%lld vs %d, p=%d\n", (listX[ix].minus ? "-" : ""), (listX[ix].num % p), modp, p);
                Entry& e = block[iy*nx+ix];
                if (!e.left) continue;
                e.rad *= p;
                INC_STAT(stat_sieve_bucket_marks);
```

Zie pagina 48

```
                intt t = e.left * pinv;
#ifndef NDEBUG
                if (t >= pmax) {
                    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);
                    assert(t < pmax);
                }
#endif
                do {
                    e.left = t;
                    t = t * pinv;
                } while (t < pmax);
            }
        }
    }
    index = ip;
}


void ABCSieve::finish_block(unsigned int startX, unsigned int startY,
                            unsigned int nx, unsigned int ny,
                            unsigned int p_index)
{
    for (unsigned int iy = 0; iy < ny; ++iy) {
        unsigned int ry = listY[startY+iy].rad;
        unsigned int last_rx = 0;
        bool not_coprime = false;
        for (unsigned int ix = 0; ix < nx; ++ix) {
            unsigned int rx = listX[startX+ix].rad;
            if (rx == last_rx && not_coprime) {
                INC_STAT(stat_notcoprime_repeat);
                continue;
            }
            if (rx >= ry) {
                // checking equality means this catches the 1 + 1 = 2 case too
                continue;
            }

            intt num = get_num(startX+ix,startY+iy);

            if (num == 0) continue;
            if (num > header.upper_c_LO) continue;

```

Zie pagina 38

```
intt left, rz;
if (p_index) {
    Entry& e =  block[iy*nx+ix];
    left = e.left;
    rz = e.rad;
} else {
    left = num;
    rz = 1;
}

intt c = num;
if (listX[startX+ix].num > c) c = listX[
    startX+ix].num;
if (listY[startY+iy].num > c) c = listY[
    startY+iy].num;

intt t = rx;
t *= ry;
// rx*ry*ry < rx*ry*rz < c for triples
if (t*ry >= c) {
    INC_STAT(stat_rxryrytoolarge);
    continue;
}
if (t*rz >= c) {
    INC_STAT(stat_rxryrztoolarge);
    continue;
}

if (last_rx != rx && !AreRadCoprime(rx,ry))
     {
```

Zie pagina 32

```
    last_rx = rx;
    INC_STAT(stat_notcoprime);
    not_coprime = true;
    continue;
}
last_rx = rx;
not_coprime = false;
intt B = (c / t);


if (trial_div(num, left, rz, B, ry, p_index
    )) {
```

Zie pagina 45

```
    // We found a triple.

    wudata triple;
    if (listX[startX+ix].minus) {
        // {x,y} = {a,b}
        if (listX[startX+ix].num > listY[
            startY+iy].num) {
```

```
                        triple.b = listX[startX+ix].num;
                        triple.a = listY[startY+iy].num;
                    } else {
                        triple.a = listX[startX+ix].num;
                        triple.b = listY[startY+iy].num;
                    }
                } else {
                    intt n1,n2;
                    n1 = num;
                    if (listX[startX+ix].num > listY[startY+iy].num)
                        n2 = listY[startY+iy].num;
                    else
                        n2 = listX[startX+ix].num;

                    if (n1 < n2) {
                        triple.a = n1;
                        triple.b = n2;
                    } else {
                        triple.b = n1;
                        triple.a = n2;
                    }
                }
                OUT_foundtriples.push_back(triple);
                // NB: do NOT increment OUT_foundcounter, as that
                // is handled by the callback
#ifdef STATS
                std::cout << "Triple: " << triple.a << " + " << triple.b << " = " << (triple.a + triple.b) << std::endl;
#endif
            }
        }
    }
}


void ABCSieve::do_block(int blocksize, int sieve_bound, bool count,
                        intt& skip_triples)
{
    unsigned int totx = listX.size();
    unsigned int toty = listY.size();

```

```
    std::sort(listX.begin(), listX.end());
    std::sort(listY.begin(), listY.end());

    std::cout << "totx = " << totx << ", toty = " << toty << std::endl;

    const unsigned int LIMIT_SOFT = blocksize;
    const unsigned int LIMIT_HARD = blocksize + 100;
    if (sieve_bound)
        block.resize(LIMIT_HARD * LIMIT_HARD);

    assert(totx <= NX_LIMIT);
    assert(toty <= NY_LIMIT);

    PRINTTIME;
    if (totx == 0 || toty == 0) return;

#ifdef STATS
    stat_totalblocksize += ((intt)totx)*toty;
#endif

    unsigned int nx = 0, ny = 0;
    for (unsigned int startX = 0; startX < totx; startX += nx) {
        nx = LIMIT_SOFT;
        if (totx - startX < LIMIT_HARD) {
            nx = totx - startX;
        }

        for (unsigned int startY = 0; startY < toty; startY += ny) {
            ny = LIMIT_SOFT;
            if (toty - startY < LIMIT_HARD) {
                ny = toty - startY;
            }

            std::cout << "subblock: " << startX << "," << startY
                      << " of size " << nx << "x" << ny;

            intt maxx = listX[startX+nx-1].num;
            intt maxy = listY[startY+ny-1].num;
            intt rxryry = (((intt)minrx)*minry)*minry;
            if (!listX[0].minus && maxy < rxryry && maxx < rxryry) {
                std::cout << ", skipping: x or y too small"
                          << std::endl;
```

```
                continue;
            }
            if (listX[0].minus && maxx+maxy < rxryry) {
                std::cout << ", skipping: x+y too small "
                          << std::endl;
                continue;
            }
            std::cout << std::endl;

            if (count) {
                skip_triples += nx*ny;
                continue;
            }


            if (skip_triples > 0) {
                if (skip_triples < nx*ny) {
                    std::cerr << "Inconsistency detected in checkpointed state. Aborting.";
                    exit(1);
                }
                skip_triples -= nx*ny;
                continue;
            }

#ifdef STATS
            stat_doneblocksize += nx*ny;
#endif
            unsigned int index = 0;
            if (sieve_bound) {
                for (unsigned int iy = 0; iy < ny; ++iy) {
                    for (unsigned int ix = 0; ix < nx; ++ix) {
                        intt left = get_num(startX+ix, startY+iy);
                        unsigned int rad = 1;
                        if (left && !(left & 1)) {
                            rad = 2;
                            do {
                                left >>= 1;
                            } while (!(left & 1));
                        }
                        block[iy*nx+ix].left = left;
                        block[iy*nx+ix].rad = rad;
                    }
                }
```

Zie pagina 38

```
                PRINTTIME;

                index = 1;
                sieve_primes_bucket(startX, startY, nx, ny, sieve_bound, index);
                PRINTTIME;
            }
            finish_block(startX, startY, nx, ny, index);
            PRINTTIME;
//typedef void (*ScanCallback) (std::vector<wudata>& d, intt scanned, intt total, intt& found, intt trialdivs);

            OUT_donecounter += nx*ny;
            (*callback)(OUT_foundtriples, OUT_donecounter, OUT_totalcount, OUT_foundcounter, OUT_trialdivcounter);
        }
    }
}

void ABCSieve::find_triples(intt skiptriples)
//int Lx, int Ux, int Ly, int Uy, int g, int sieve_bound)
{
    STARTTIME;

    minrx = header.lower_rx;
    minry = header.lower_ry;
    maxrx = header.upper_rx;
    maxry = header.upper_ry;

    if (header.lower_c_HI != 0 || header.lower_c_LO != 1) {
        std::cerr << "This client does not support minC != 1"
                  << std::endl;
        return;
    }

    if (header.upper_c_HI != 0 || (header.upper_c_LO & 0x8000000000000000ULL)) {
        std::cerr << "This client does not support maxC >= 2^63"
                  << std::endl;
        return;
    }
```

Zie pagina 49

Zie pagina 50

```
    if (header.upper_rx > 0xFFFFFFFFUL || header.upper_ry > 0xFFFFFFFFUL) {
        std::cerr << "This client does not support radicals >= 2^32"
                  << std::endl;
        return;
    }
    if (header.gcdwith > 0xFFFFFFFFUL) {
        std::cerr << "This client does not support gcdwith >= 2^32"
                  << std::endl;
        return;
    }
    if (header.minQ != 0x01000000UL) {
        std::cerr << "This client does not support minQ != 1"
                  << std::endl;
        return;
    }
    if (header.sievebound > 300000) {
        // This is due to the size of precomputed prime tables
        std::cerr << "This client does not support sievebound > 300000"
                  << std::endl;
        return;
    }
    if (header.sievebound > P_LIMIT) {
        // see prepare_buckets
        std::cerr << "This client does not support sievebound > " << P_LIMIT
                  << std::endl;
        return;
    }

    if (header.blocksize > 10000) {
        std::cerr << "This client does not support blocksize > 10000"
                  << std::endl;
        return;
    }

    const intt _limit = header.upper_c_LO;

    sieveradicals_interval(minrx,maxrx-minrx,lowrads,lowprimes);
    sieveradicals_interval(minry,maxry-minry,highrads,lowprimes);

```
Zie pagina 31

```
    unsigned int lowradx = binsearch(lowrads, minrx);
    unsigned int lowrady = binsearch(highrads, minry);

    unsigned int t = header.gcdwith;
    unsigned int g = header.gcdx;

    if (skiptriples)
        std::cout << "Resuming ";
    else
        std::cout << "Starting ";
    std::cout << "block with gcd(x," << t << ") = " << g << std::endl;
    std::cout << "rx in [" << minrx << "," << maxrx << "), "
              << "ry in [" << minry << "," << maxry << ")" << std::endl;
    std::cout << "sieve bound = " << header.sievebound << std::endl;


    num_t dg = { g, g };
    listX.clear(); listY.clear();
    listX.reserve(NX_LIMIT); listY.reserve(NY_LIMIT);
    store_target = &listX;
    ForEachRad(&storeRad, lowradx, maxrx-1, lowrads, _limit, t/g, nums_tmp, dg);
    if (listX.size() == 0) {
        std::cout << "nx = 0" << std::endl;
        ENDTIME;
        std::cout << "done" << std::endl;
    }

    store_target = &listY;
    ForEachRad(&storeRad, lowrady, maxry-1, highrads, _limit, g, nums_tmp, ONE);
    PRINTTIME;

    if (skiptriples == 0) {
        std::cout << "COUNTING" << std::endl;
        // We first quickly compute the total amount of work
        do_block(header.blocksize, header.sievebound, true, skiptriples);
        for (unsigned int i = 0; i < listX.size(); ++i)
            listX[i].minus = true;
        do_block(header.blocksize, header.sievebound, true, skiptriples);
        for (unsigned int i = 0; i < listX.size(); ++i)
            listX[i].minus = false;
```

Zie pagina 43
Zie pagina 33
Zie pagina 33
Zie pagina 52
Zie pagina 52

```
        OUT_totalcount = skiptriples;
        OUT_donecounter = 0;
        skiptriples = 0;
    }

    std::cout << "SIEVING" << std::endl;
```
Zie pagina 52
```
    do_block(header.blocksize, header.sievebound, false
        , skiptriples);
    for (unsigned int i = 0; i < listX.size(); ++i)
        listX[i].minus = true;
    do_block(header.blocksize, header.sievebound, false
        , skiptriples);
    ENDTIME;
    std::cout << "done" << std::endl;
}

void ABCSieve::setStats(intt totalcount, intt
    donecounter, intt foundcounter, intt trialdivcounter
    )
{
    OUT_totalcount = totalcount;
    OUT_donecounter = donecounter;
    OUT_foundcounter = foundcounter;
    OUT_trialdivcounter = trialdivcounter;
}

void find_triples(abc_sieve_header& DATA, intt
    skiptriples, intt triples_total, intt hits, intt
    trialdivcount, ScanCallback callback)
{
    ABCSieve sieve(DATA, callback);
```
Zie pagina 58
```
    sieve.setStats(triples_total, skiptriples, hits,
        trialdivcount);
```
Zie pagina 55
```
    sieve.find_triples(skiptriples);
```
Zie pagina 41
```
    sieve.printStats();
}
```

A.4 HEADER FILES

A.4.1 abc_common.h

```
#ifndef ABC_COMMON
#define ABC_COMMON

#include <vector>

```

```
#define CODE_VERSION 400

#ifdef _WIN64
  #define FINTT "%I64u"
  typedef unsigned __int64 intt;
#else
  #define FINTT "%llu"
  typedef unsigned long long intt;
#endif

struct wudata {
  intt a;
  intt b;
};

struct abc_header {
    intt alpha;
    intt beta;
    intt interval;
    intt limit;
};

struct abc_sieve_header {
    intt id;
    intt lower_rx, upper_rx;
    intt lower_ry, upper_ry;
    intt gcdwith, gcdx;
    intt lower_c_LO, lower_c_HI;
    intt upper_c_LO, upper_c_HI;
    unsigned int blocksize, sievebound;
    unsigned int minQ;
};


typedef void (*ScanCallback) (std::vector<wudata>& d, intt scanned, intt total, intt& found, intt trialdivs);






#endif
```

A.4.2 abc_sieve.h

```
#ifndef ABC_SIEVE_H
#define ABC_SIEVE_H

void find_triples(abc_sieve_header& DATA, intt
    skiptriples, intt triples_total, intt hits, intt
    trialdivcount, ScanCallback callback);
unsigned int estimate_sieve_bound(unsigned int Lx,
    unsigned int Ly);

#endif
```

A.4.3 abc_sieve_util.h

```
#ifndef ABCALGO_UTIL_H
#define ABCALGO_UTIL_H

#include <list>
#include <vector>

#include "abc_common.h"

const unsigned int NUMPERRAD_LIMIT = 300000;

struct rad_t {
    unsigned int rad;
    std::list<unsigned int> primes;
};

struct num_t {
    intt num;
    unsigned int rad;
    bool operator==(const num_t& b) const { return num
        == b.num; }
};

extern const num_t ZERO;
extern const num_t ONE;

bool operator<(const rad_t& r1, const rad_t& r2);
bool operator<(const num_t& r1, const num_t& r2);



void sieveprimes(unsigned int primelimit,
        std::vector<unsigned int>& lowprimes);
```

```
void sieveradicals(unsigned int radlimit, std::vector<rad_t>& rads);

// Needs primes up to sqrt(start+length) in lowprimes
void sieveradicals_interval(intt start, unsigned int length,
              std::vector<rad_t>& rads,
              std::vector<unsigned int>& lowprimes);

// Check if two squarefree integers are coprime
bool AreRadCoprime(unsigned int u, unsigned int v);

#if 0
inline static int binsearch(num_t* a, int l, int h, intt target)
{
    while (l <= h) {
        int t = (l + h) / 2;
        if (a[t].num == target) return t;
        if (a[t].num > target) {
            h = t-1;
        } else {
            l = t+1;
        }
    }
    return -1;
}
#endif

// find square root of number if it is a perfect square
// if not a square, returns square root rounded down
bool issquare(intt n, unsigned int& r);
// same, for cube root
bool iscube(intt n, unsigned int& r);

// determine if n is a fifth power of a prime. If true, return prime
bool is_fifth_ppower(intt n, unsigned int& p);
bool is_seventh_ppower(intt n, unsigned int& p);

typedef bool (*ForEachRadFunc) (num_t n, unsigned int rad_index, num_t data);

// Search all numbers with radical with index from index_from (inclusive)
// and radical size up to rad_limit (inclusive) and size up to num_limit (inc.)
// and coprime to coprime_to.
// Radicals are taken from the rads[] vector.
```

```
// The nums[] argument is used for temporary storage.
// If the ForEachRadFunc returns false, the current radical is skipped.
// Returns the number of radicals considered.
intt ForEachRad(ForEachRadFunc f,
        unsigned int index_from, unsigned int rad_limit,
        std::vector<rad_t>& rads,
        intt num_limit, unsigned int coprime_to,
        std::vector<num_t>& nums,
        num_t data);


typedef void (*TripleCallback) (intt a, intt b);
typedef void (*ScanCallback) (std::vector<wudata>& d, intt scanned, intt total, intt& found, intt trialdivs);


#endif


#ifdef _WIN32
typedef struct { long long quot, rem; } lldiv_t;
#endif
```