Segmentierung und Optimierung von Algorithmen zu Problemen aus der Zahlentheorie

Inaugural-Dissertation

zur
Erlangung des Doktorgrades
der Naturwissenschaftlichen Fachbereiche
(Fachbereich Mathematik)

der Justus-Liebig-Universität Gießen

vorgelegt von
Jörg Richstein
aus Hagen/Westfalen

Gießen, 1999

D 26

Dekan:	Prof. Dr. Winfried Stute
I. Berichterstatter:	Prof. Dr. Henner Kröger
II. Berichterstatter:	Prof. Dr. Sigbert Jaenisch
Tag der mündlichen Prüfung:	6. August 1999

Für meine Eltern

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

1 Einleitung

Noch wenige Jahre vor Entstehung dieser Arbeit war die Rechnerwelt von großen, zentralistisch orientierten Systemen geprägt. Erst zu Beginn der achtziger Jahre begann mit der Entwicklung von Workstations und Personalcomputern eine Wende hin zu kleinen, dezentralen Rechnerarchitekturen. Nicht zuletzt durch die Unterstützung immer leistungsfähigerer Netze wurden in den letzten zehn Jahren fast überall Großrechenanlagen durch Verbunde kleinerer Maschinen ersetzt.

Während dieser rasanten Veränderung entstanden jedoch zunehmend Probleme, die aus der wesentlich langsameren Entwicklung von verteilten, auf die neuen Voraussetzungen zugeschnittenen Algorithmen und ihrer Implementierungen resultierten. Dies führte zum Teil zu einer erheblichen Verzögerung des eigentlich sehr sinnvollen und natürlichen Konzeptes zur Parallelisierung und Verteilung von Problemen. Vor allem in der ersten Hälfte der neunziger Jahre scheiterten viele, teilweise euphorisch begonnene Projekte einer Umstellung auf die neuen, populären Client/Server-Rechnerumgebungen an fehlenden Verfahren zur optimalen Verteilung von bisher sequentiell bearbeiteten Problemen. Dies ist nicht auf spezielle Bereiche beschränkt. Die Entwicklung und Implementierung verteilter, paralleler Algorithmen stellt bis heute eine der wichtigsten Aufgaben der Informatik in allen Disziplinen dar.

Dabei sind Probleme aus der Mathematik keine Ausnahme. Insbesondere in der Zahlentheorie sind zum Teil Jahrhunderte alte Fragen noch immer ungelöst. Der Ursprung von Vermutungen zu möglichen Antworten ist meist das Ergebnis von Proberechnungen, die zunächst von Hand, später mit mechanischen und schließlich elektronischen Rechenanlagen durchgeführt wurden. Das Interesse gilt dabei einerseits der Bekräftigung der Vermutung selbst, andererseits aber auch der Ermittlung zusätzlichen, problemverwandten Datenmaterials, das wiederum Wege zur theoretischen Lösung aufzeigen könnte. Damit sind die Möglichkeiten des Rechnereinsatzes keineswegs erschöpft. Selbst Beweislücken konnten bereits geschlossen werden, wie sich bei der Lösung des Vierfarbenproblems aus der Graphentheorie zeigte: Im Jahre 1852 versuchte *Francis Guthrie*, die Landkarte Englands derart zu färben, daß keine zwei angrenzenden Grafschaften dieselbe Farbe aufwiesen. Er bemerkte, daß dazu vier Farben genügten. Es stellte sich die Frage, ob dies ein Spezialfall war oder ob diese Aussage allgemeingültig ist (siehe auch [Cay78]). Erst 1969 konnte durch *H. Heesch* ([Hee69]) ein Beweisansatz gefunden werden, der allerdings Lücken enthielt. Es verblieben nur endlich viele Fälle, die mit der gefundenen Methode nicht erfaßbar waren. Schließlich gelang es *K. Appel* und *W. Haken* im Jahre 1976, diese fehlenden Fälle durch den Einsatz massiver Rechnerleistung auszuschließen ([App77]).

Ein analoges Beispiel aus der Zahlentheorie ist die Widerlegung der *Mertensschen Vermutung* über die Beschränkung des Betrages der summatorischen Funktion der Möbiusfunktion. *Mertens* vermutete 1897, daß mit $M(x) = \sum_{n \leq x} \mu(n)$ für alle $x > 1$ gilt: $|M(x)| < \sqrt{x}$ ([Mer97]). Die Richtigkeit dieser Vermutung hätte weitreichende Folgen gehabt. Sie konnte erst 1985, etwa neunzig Jahre später – wiederum durch Rechnereinsatz – von *A. Odlyzko* und *H.J.J. te Riele* widerlegt werden (interessanterweise ohne die explizite Angabe eines Ausnahmefalles) ([Odl85] und [Rie85]).

Die vorliegende Arbeit beschäftigt sich mit der Entwicklung und Optimierung verteilter Algorithmen zu Problemen aus der Zahlentheorie. Das grundlegende Prinzip der Aufspaltung des Gesamtproblems und die sich anschließende, parallele Lösung der entstandenen Teilaufgaben existiert dabei nicht etwa erst seit Beginn der Entwicklung vernetzter, kleinerer Rechnerarchitekturen: Bereits im Jahre 1878 führte *J.W.L. Glaisher* [Gla78] eine Zählung von Primzahlzwillingen durch. Die Berechnung von Teilintervallen wurde damals von mehreren Mitarbeitern ausgeführt. Die simultane Nutzung „menschlicher Rechenleistung“ wird dabei heute durch den parallelen Einsatz einer Vielzahl verschiedener Computer an möglicherweise völlig unterschiedlichen Standorten ersetzt. Besonders letzteres wird durch das sich in den letzten zwanzig Jahren in der Anzahl der beteiligten Rechner exponentiell wachsende Internet begünstigt. So findet beispielsweise seit nunmehr dreieinhalb Jahren eine von *G. Woltman* initiierte, verteilte Suche nach *Mersenneschen Primzahlen* im Internet statt, wobei die ansonsten brachliegende Rechenleistung mehrerer tausend Maschinen parallel genutzt wird (siehe z.B. [Wol99]). Weitere Beispiele sind etwa die Berechnung von Dezimalstellen der Zahl π oder auch Versuche zur verteilten Faktorisierung großer Zahlen ([Zim99a]), die seit der Entwicklung des *RSA*-Algorithmus im Jahre 1977 ([Adl78]) in verschiedenen kryptographischen Verfahren eine entscheidende Rolle spielen.

Speziell werden in der vorliegenden Arbeit vier Probleme behandelt. Zur Vorbereitung auf die Bewertung der jeweils vorzustellenden Algorithmen wird in Kapitel 2 zunächst ein neues Berechnungsmodell entwickelt, das bestehende Modelle um in der Praxis relevante Eigenschaften erweitert und die formale Grundlage sämtlicher Verfahren darstellen wird. Berücksichtigt werden dabei sowohl Unterschiede in den Komplexitäten verschiedener Operationen als auch Kosten für Speicherzugriffe. Beides hat in der Praxis entscheidenden Einfluß auf die Gesamtkomplexität der Implementierungen von Algorithmen. Der Grundaufbau des Modells wird zunächst formal definiert, den Instruktionen der Modellmaschine dann Kosten zugeordnet. Es folgt in Abschnitt 2.4 die Einführung einer Sprache zur Algorithmenbeschreibung. Diese wird schließlich zum Zwecke der Analyse der folgenden Verfahren durch Angabe eines Übersetzers in Modellmaschinen-Instruktionen auf das Berechnungsmodell zurückgeführt. Sämtliche vorzustellenden Algorithmen werden anhand der Beschreibungssprache erläutert und mit dem vorher geschaffenen Komplexitätsmaß bewertet. Diese Vorgehensweise läßt eine wesentlich realistischere Bewertung der Komplexitäten zu als es in herkömmlichen Modellen möglich gewesen wäre. Eine „Tauglichkeitsuntersuchung“ des entstandenen Komplexitätsmaßes für Programme über dieser Sprache wird im Zusammenhang des nachfolgenden Kapitels durchgeführt.

Kapitel 3 beschäftigt sich mit *Primzahl-Sieben*, also Algorithmen zur Trennung zerlegbarer Zahlen von Primzahlen eines gegebenen Intervalls. Die Basis stellt dabei das über 2000 Jahre alte *Sieb des Eratosthenes* dar. Zunächst werden daran einige einfache Verbesserungen vorgenommen. An der Analyse der daraus resultierenden Verfahren zeigt sich, daß selbst feine Unterschiede durch das Komplexitätsmaß der Beschreibungssprache erfaßt werden können. In einem nächsten Schritt findet dann zur Vorbereitung auf eine spätere Verteilung die *Segmentierung* des Basissiebes statt. Der Begriff der *Segmentierung*, der im Bereich der Primzahlsiebe eine gewisse Tradition hat, wird in dieser Arbeit übernommen. Er ist zum einen als theoretische Vorstufe zu einer praktischen Verteilung zu verstehen und soll zum anderen vom Begriff der *Partitionierung* abgrenzen, unter der man häufig eine disjunkte Zerlegung versteht, die hier nicht immer gemeint sein muß.

Ausgehend von einer ersten segmentierten Version werden schrittweise Verbesserungen entwickelt, die schließlich in einem hochoptimierten Verfahren münden. Es wird gezeigt, daß dieses trotz seiner asymptotisch schlechteren Laufzeit unter realistischen Bedingungen andere Algorithmen übertrifft. Eine Implementierung des resultierenden Primzahlsiebes wird in 3.4 beschrieben. Dabei wurde auf möglichst weitreichende Parametrisierbarkeit geachtet, die sich in späteren Anwendungen zum Teil erheblich auswirkt.

In Kapitel 4 wird eine (Teil-) Verifikation der inzwischen über 250 Jahre alten *Goldbachschen Vermutung* vorgenommen. Dabei handelt es sich um die Frage, ob sich jede gerade Zahl größer oder gleich 4 als Summe zweier Primzahlen darstellen läßt. Trotz der Einfachheit ihrer Formulierung zählt man einen möglichen Beweis der Goldbachschen Vermutung zu den schwierigsten Problemen der gesamten Mathematik überhaupt. Eine etwas schwächere Aussage, die *ternäre* Goldbachsche Vermutung besagt, daß sich jede ungerade Zahl größer oder gleich 7 als Summe dreier Primzahlen darstellen läßt. Die ternäre Vermutung, die aus der Korrektheit der binären folgen würde, konnte inzwischen unter der Annahme der verallgemeinerten Riemannschen Vermutung bewiesen werden. Der letzte Lückenschluß in diesem Beweis wurde unabhängig voneinander von *Y. Saouter* sowie *J.-M. Deshoulliers* et. al. wiederum jeweils durch massiven Rechnereinsatz erreicht([Sao98], [Des97]). Es wäre sogar theoretisch möglich, einen Beweis ohne weitere Voraussetzungen durch Maschinen zu komplettieren, da bereits 1937 von *I.M. Vinogradov* bewiesen wurde, daß die ternäre Vermutung für alle genügend großen n gilt ([Vin37]). Eine explizite Schranke wurde erstmals von *K.G. Borozkin* in [Bor56] angegeben und später von *J.R. Chen* und *Y. Wang* in [Che89] auf 10^{43000} reduziert. Ob eine solche Schranke aber jemals von Rechnern erreicht werden kann, scheint zumindest sehr fraglich. Eine analoge Schranke für die binäre Vermutung ist auch unter weiteren (nichttrivialen) Voraussetzungen nicht bekannt.

Es werden zunächst zwei Methoden zur Verifikation beschrieben sowie deren Vor- und Nachteile aufgezeigt. Aus diesen Überlegungen folgt die Auswahl eines der beiden Verfahren. Laufzeitkritische Punkte konnten durch mehrere Optimierungen entscheidend verbessert werden. Der resultierende Algorithmus wurde implementiert und schließlich verteilt.

Eine Erweiterung des Goldbachschen Problems stellt die Frage dar, wieviele Zerlegungen als Summe zweier Primzahlen es für eine gerade Zahl gibt. Die Anzahl der Zerlegungen scheint zu wachsen, was sich durch heuristische Überlegungen verdeutlichen läßt. Jedoch muß natürlich, solange die Goldbachsche Vermutung nicht bewiesen ist, prinzipiell auch noch in Betracht gezogen werden, daß es möglicherweise nicht einmal eine einzige Zerlegung gibt. Der Frage der *Anzahl der Goldbach-Partitionen* wird in Kapitel 5 nachgegangen. Das Problem der Bestimmung dieser Anzahl läßt sich durch einen sequentiellen Algorithmus eigentlich sehr einfach lösen. Allerdings stößt das Verfahren in der Praxis schnell an seine Grenzen, die vor allem aus dem Mangel an zur Verfügung stehendem Hauptspeicher resultieren. Es wird zunächst gezeigt, daß durch eine Segmentierung des Basisverfahrens ein Minimum an Speicher genügt. Allerdings führt die beschriebene Vorgehensweise zunächst zu einer deutlichen Erhöhung der Zeitkomplexität, die jedoch praktisch durch die sich automatisch ergebende Verteilbarkeit aufgefangen werden kann. Die Ergebnisse der verteilten Berechnung werden schließlich verschiedenen historischen Vermutungen über das Wachstum der Anzahl der Partitionen gegenübergestellt.

Kapitel 6 stellt ein Verfahren zur verteilten Suche nach modulo p verschwindenden *Fermat-Quotienten* zur Basis a für prime a und p vor. Die dazu äquivalente Frage der Lösbarkeit der Kongruenz $a^{p-1} \equiv 1 \mod p^2$ wurde erstmals vor etwa 170 Jahren von *N. Abel* in [Abe28] aufgeworfen. Sie steht in engem Zusammenhang mit dem (inzwischen von *A. Wiles* in [Wil95] bewiesenen) *letzten Satz von Fermat*. Über die Lösungen der Kongruenz oder ihrer Struktur ist wenig bekannt. Es wird zunächst ein einfacher Algorithmus beschrieben, der jedoch in der Praxis entscheidende Nachteile aufweist. Selbst nach der Ersetzung einiger problematischer Stellen verbleiben praktische Schwierigkeiten, die nur durch eine recht komplizierte, aber letztendlich erfolgreiche Vorgehensweise behoben werden können. Das resultierende und schließlich implementierte und verteilte Verfahren zeigt eine praktische Schranke für eine im Berechnungsmodell getroffene Idealisierung auf, die trotz der relativ umständlichen Methode nicht durchbrochen werden mußte. Die getroffene Idealisierung erfährt damit eine weitere Rechtfertigung.

Abschließend wird eine Softwareimplementierung zur Steuerung verteilter Rechnungen vorgestellt. Dabei werden sowohl vernetzte Systeme mit oder ohne gemeinsamem Speicher unterstützt als auch einzelne Rechner eingebunden, die nur teilweise netzverbunden sind oder auch völlig getrennt arbeiten. Durch die relativ schlanke Struktur und einfache Konfigurierbarkeit ist es prinzipiell möglich, weltweite Rechnerressourcen in eine verteilte Rechnung einzubinden. Netzkommunikationen werden dabei weitestgehend vermieden, um eine Abhängigkeit von zentralen Servern zu vermeiden. Sämtliche Rechnungen wurden unter Verwendung dieser Software auf relativ kleinen Rechnern an unterschiedlichen Standorten durchgeführt.

Das Ergebnis der Rechnung zur Goldbachschen Vermutung – die Verifikation bis $4 \cdot 10^{14}$ – stellt eine Ausdehnung des bisher bestätigten Bereichs auf das Vierfache dar. Die vorangegangene Rechnung hatte dabei erst kurz zuvor jeweils zur Hälfte auf einer Großrechenanlage (*J.-M. Deshoulliers* und *H.J.J. te Riele*) und einem Parallelrechner (*Y. Saouter*) stattgefunden ([Des98]).

Die verteilte Implementierung des segmentierten Verfahrens zur Anzahl der Goldbach-Partitionen resultierte in der Berechnung der Zerlegungen aller geraden Zahlen unterhalb von $5 \cdot 10^8$, was wiederum einer Vervierfachung des bisher abgedeckten Bereichs entspricht. Die letzte Berechnung dieser Art wurde 1997 von *D. Lavenier* und *Y. Saouter* durchgeführt ([Lav98]). Dabei war eine speziell für das Problem entwickelte Hardware zum Einsatz gekommen.

Durch eine maschinennahe Implementierung und Verteilung konnten bisherige Berechnungen zum Problem der Fermat-Quotienten um das etwa zwölffache erweitert werden ([Kel97], [Ern97]). Die für alle primen $a < 1000$ und $p < 10^{11}$ durchgeführte Rechnung lieferte acht neue Lösungen, darunter eine, die die erste Lösung zur Basis $a = 929$ darstellt.

In der vorliegenden Arbeit wird gezeigt, daß durch sorgfältige Implementierung und Verteilung von Algorithmen kleine, kostengünstige Rechnersysteme durchaus mit speziellen Architekturen oder Großrechnern konkurrieren und diese in ihrer Leistung sogar übertreffen können. Anhand mehrerer Algorithmen zu Problemen aus der Zahlentheorie wird dargelegt, wie auch sehr aufwendige Berechnungen durch Verteilung realisiert werden können.

1.1 Notationen

Die folgende Auflistung gibt einen Überblick über die in dieser Arbeit verwendeten Notationen (nicht notwendigerweise in der Reihenfolge ihres Auftretens). Dabei bezeichnen kleine griechische Buchstaben gewöhnlich reelle Zahlen, lateinische Buchstaben ganze Zahlen (speziell sind p und q Primzahlen).

$\mathbb{N}$	Die natürlichen Zahlen $\{1, 2, 3, \dots\}$
$\mathbb{N}_0$	$\mathbb{N} \cup \{0\}$
$\mathbb{Z}$	Die ganzen Zahlen $\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$
$\mathbb{R}$	Die reellen Zahlen
$\lvert\mathbf{M}\rvert$	Die Mächtigkeit der Menge $\mathbf{M}$
$\lvert z\rvert$	Der Betrag einer ganzen Zahl
p_n	Die n-te Primzahl
$\pi(x)$	Die Anzahl der Primzahlen kleiner oder gleich x
$m \mid n$	m teilt n, also $n = k \cdot m$, wobei $k \in \mathbb{Z}$
$m \nmid n$	m teilt n nicht
$\lfloor\alpha\rfloor$	Die ganze Zahl a, so daß $a \leq \alpha < a + 1$
$\lceil\alpha\rceil$	Die ganze Zahl a, so daß $a - 1 < \alpha \leq a$
$ggT(m, n)$	Der größte, gemeinsame Teiler von m und n
$\varphi(n)$	Die Eulersche Funktion
$n \mod m$	Der Rest von n nach Division durch m, also $n - m \cdot \left\lfloor\frac{n}{m}\right\rfloor$
$\exists$	Existenzquantor
$\forall$	Allquantor
$f(x) = g(x) + O(h(x))$	$\exists\ c, x_0 \in \mathbb{R}$ mit $c, x_0 > 0 :\ f(x) - g(x) \leq c \cdot h(x)\ \forall\ x \geq x_0$
$f(x) \sim g(x)$	$\lim_{x\to\infty} f(x)/g(x) = 1$
$a \approx b$	a und b sind „ungefähr" gleich
ε	Das leere Wort, die Folge der Länge 0
A^0	ε
A^i	$(a_1, a_2, \dots, a_i), a_i \in A$ Worte der Länge i über A
A^*	$\bigcup_{i=0}^{\infty} A^i$ die Menge aller unendlichen Folgen über A
A^+	$\bigcup_{i=1}^{\infty} A^i$ die Menge aller nichtleeren Folgen über A
$T[i]$	Die Selektion der i-ten Komponente eines Tupels T
M_k	$\prod_{i=1}^{k} p_k$ das Produkt der ersten k Primzahlen

2 Berechnungsmodell

2.1 Einführung

Bei der Bewertung und beim Vergleich zweier Verfahren, die dasselbe Problem lösen, sind im wesentlichen zwei Kriterien von Bedeutung. Einerseits ist man an möglichst kurzer Laufzeit interessiert, andererseits möchte man den während des Ablaufs benötigten Speicher minimieren. Diese beiden Ziele sind nicht unabhängig voneinander. Häufig müssen sie gegeneinander aufgewogen werden, um eine Gesamtoptimierung zu erreichen. Die Dauer, die ein Verfahren zur Lösung eines Problems in Anspruch nimmt, wird als *Zeitkomplexität* bezeichnet. Der dabei benötigte Speicher wird durch die *Raumkomplexität* beschrieben. Beide Maße werden als Funktionen der jeweils festzulegenden Problemgröße definiert.

Die Analyse der Zeit- und Raumkomplexität von Verfahren erfordert zunächst die Beschreibung eines formalen Berechnungsmodells. Ein Berechnungsmodell sollte einerseits eine realistische Abschätzung der Komplexitäten auf einfache Art ermöglichen. Andererseits darf das Modell keine wesentlichen Leistungseinschränkungen aufweisen, es muß *berechnungsuniversell*, also äquivalent zu einer Turingmaschine im Sinne der Berechenbarkeit von Funktionen sein.

Das hier entwickelte Modell verbindet Elemente einer *RAM* (*R*andom *A*ccess *M*achine) und einer *RASP* (*R*andom *A*ccess *S*tored *P*rogram *M*achine) deren Funktionsweisen wiederum auf die von *John von Neumann* im Jahre 1946 beschriebene sequentielle Rechenmaschine zurückgehen. Verschiedene Konstruktionen der Modelle führen zu teilweise erheblichen Differenzen in den daraus resultierenden Komplexitäten. Die vorliegende Arbeit legt besonderen Wert auf Implementierbarkeit und Realisierung der vorgestellten Algorithmen. Dieser Tatsache wird in der folgenden Konstruktion der Modellmaschine *RX-RAM* (*R*egister e*X*tended *R*andom *A*ccess *M*achine) Rechnung getragen. Dabei wird versucht, in der Praxis entscheidende Faktoren in das Modell einzubauen um somit realistische Aussagen über die Komplexitäten der Verfahren treffen zu können.

Abschnitt 2.2 beschäftigt sich zunächst mit der formalen Definition sowohl des Modells als auch der Zeit- und Raumkomplexitäten von *RX-RAM*-Programmen. In Abschnitt 2.3 folgt eine Diskussion der Maschine, wobei neben der Realitätsnähe insbesondere der Vergleich mit anderen Modellen im Vordergrund steht. Anschließend wird in 2.4 eine Sprache vorgestellt, die der späteren Beschreibung von Algorithmen dient. Die operationelle Semantik der Sprache wird durch Reduktion auf *RX-RAM*-Operationen definiert, wodurch eine genaue Festlegung der Komplexitätsbegriffe auch für Programmen der Beschreibungssprache möglich wird. Bei der späteren Analyse der vorzustellenden Verfahren werden die aus diesem Kapitel resultierenden Komplexitäten zur Anwendung kommen.

2.2 Die Modellmaschine *RX-RAM*

Definition 2.1 Es sei die Länge $\mathrm{lg} : \mathbb{Z} \to \mathbb{N}$ einer Zahl $z \in \mathbb{Z}$ für $1 < k \in \mathbb{N}$ definiert durch

$$\mathrm{lg}(z) = \begin{cases} 2 & \text{falls } z = 0 \\ \lfloor \log_k(|z|) \rfloor + 2 & \text{falls } z \neq 0 \end{cases}$$

Definition 2.2 (Zelle) Eine *Zelle* ist ein Platzhalter für eine ganze Zahl. Die Darstellung einer Zahl z innerhalb der Zelle erfolge dabei in $\mathrm{lg}(z)$ Stellen zur Basis $k > 1$.

Bemerkung. Speziell sei im folgenden $k = 2$ gewählt. Dies wird keine Einschränkung bedeuten, jede andere Basis hätte ebenso funktioniert. Es sei angenommen, daß jede Zelle potentiell unendlich viele Stellen hat, wobei der Inhalt einer Zelle zu jedem Zeitpunkt eine endliche Darstellung besitzt. Ein Vorzeichenbit existiert, woraus die in 2.1 festgelegte Länge resultiert.

Definition 2.3 (Modellmaschine *RX-RAM*) Eine 7-Tupel $\mathscr{R} = (\mathbf{R}, \mathbf{S}, \mathbf{E}, \mathbf{A}, \mathbf{I}, \mathbf{O}, \mathbf{P})$ heißt *Register Extended Random Access Machine* (*RX-RAM*). Dabei ist

$\mathbf{R}$ = $\{R_0, R_1, R_2, \ldots, R_\rho\}$ mit $\rho \in \mathbb{N}_0$ die endliche Menge der *Registerzellen*, wobei speziell R_0 der *Programmzähler* ist,

$\mathbf{S}$ = $\{S_1, S_2, \ldots\}$ die Menge der *Speicherzellen*,

$\mathbf{E}$ = $\{E_1, E_2, \ldots\}$ die Menge der *Eingabezellen*,

$\mathbf{A}$ = $\{A_1, A_2, \ldots\}$ die Menge der *Ausgabezellen*,

$\mathbf{I}$ = {`mov`,`ld`,`st`,`add`,`sub`,`mul`,`div`,`mod`,`and`,`or`,`not`,`xor`,`shl`,`shr`,`eq`,`neq`} $\cup$ {`leq`,`geq`,`gt`,`lt`,`jmp`,`jz`,`jgz`,`jlz`,`get`,`put`,`end`} die Menge der *Instruktionen*,

$\mathbf{O}$ $\subseteq$ $\mathbf{I} \times \mathbf{F}^3$ die Menge der *Operationen*,

$\mathbf{P}$ = $\{P_1, P_2, \ldots\}$ der *Programmspeicher*.

$\mathbf{F} = \mathbb{Z} \cup \mathbf{R} \cup \{S_j : j \in \mathbb{N} \cup \mathbf{R}\} \cup \mathbf{E} \cup \mathbf{A}$ bezeichnet die Menge der *Operanden*. Die Elemente P_i des Programmspeichers seien Platzhalter für Operationen $o \in \mathbf{O}$.

Bemerkung. Die Mengen $\mathbf{S}, \mathbf{R}, \mathbf{E}, \mathbf{A}$ sind (geordnete) Mengen von Zellen im Sinne von Definition 2.2. Die in der Definition von $\mathbf{F}$ auftauchenden Operanden S_{R_i} sind als „Pointervariablen“ zu verstehen, R_i ist dabei ein Register, das die Adresse einer anderen Speicherzelle enthält. Die Definition der Modellmaschine ist unabhängig vom Inhalt der Eingabezellen und des Programmspeichers. Formal wird nun eine Modellmaschine *RX-RAM* auch als eine Abbildung $\mathscr{R} : \mathbf{O}^+ \times \mathbb{Z}^+ \longrightarrow \mathbb{Z}^+$ verstanden. Abbildung 1 zeigt die Modellmaschine schematisch.

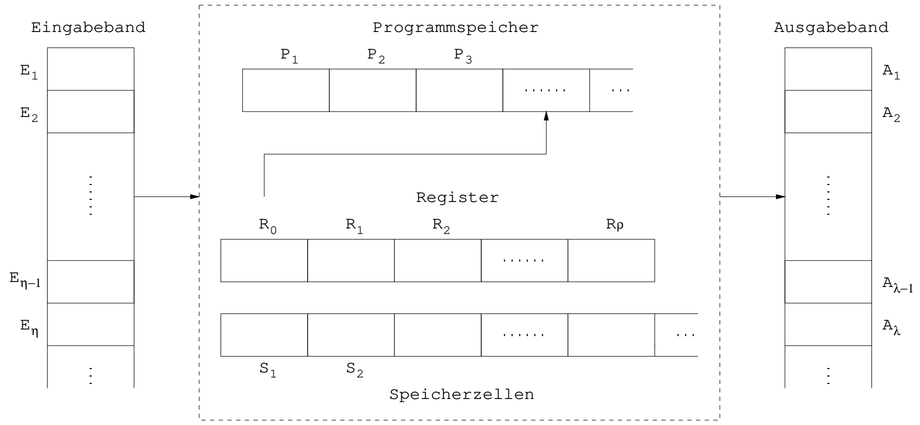


Abbildung 1: Die Modellmaschine *RX-RAM*

Definition 2.4 (Programm) Ein Tupel

$$P = (p_1, p_2, \ldots, p_\kappa) \text{ mit } \kappa \in \mathbb{N}, p_i \in \mathbf{O}, 1 \leq i \leq \kappa$$

heißt *(RX-RAM-)Programm.*

Es soll nun einer Modellmaschine ein Programm zugeordnet werden:

Definition 2.5 (Programmierte *RX-RAM*) Es sei $\mathscr{R}$ eine *RX-RAM* und P ein Programm. Dann heißt eine Abbildung

$$\mathscr{R}[\![P]\!] \ : Z^+ \to Z^+$$

programmierte RX-RAM.

Bemerkung. Die Definition ist so zu verstehen, daß sich eine programmierte *RX-RAM* aus der Basismaschine durch Füllen des Programmspeichers mit einem Programm $P \in \mathbf{O}^+$ ergibt. Programmspeicherzelle P_i enthalte also p_i. Im folgenden wird nun immer davon ausgegangen, daß eine *RX-RAM* programmiert sei.

Ziel wird es nun sein, die Semantik eines Programms P mit einer mit P programmierten *RX-RAM* zu identifizieren. Dazu einige Vorbereitungen.

Definition 2.6 (Inhalt) Es sei $j \in \mathbb{N}$ und $\mathscr{R} = (\mathbf{R}, \mathbf{S}, \mathbf{E}, \mathbf{A}, \mathbf{I}, \mathbf{O}, \mathbf{P})$ eine *RX-RAM*. Dann bezeichne s_j die ganze Zahl in Speicherzelle S_j, r_j diejenige in Register R_j (für $j \leq \rho$) sowie e_j und a_j die in Eingabezelle E_j bzw. Ausgabezelle A_j. Die Funktion $c : \mathbf{F} \to \mathbb{Z}$ weist einem Operanden F einen *Inhalt* zu:

$$c(F) = \begin{cases} F & F \in \mathbb{Z} \\ e_j & \text{falls } F = E_j \\ a_j & \text{falls } F = A_j \\ s_j & \text{falls } F = S_j \\ r_j & \text{falls } F = R_j \\ s_i & falls i = r_j und F = S_{R_j} \end{cases}$$

Definition 2.7 (Konfiguration der Modellmaschine) Eine *Konfiguration* K_i $(i \in \mathbb{N}_0)$ einer Modellmaschine *RX-RAM* ist ein Quadrupel (R, S, E, A). Dabei seien

$$\begin{aligned} R &= (r_0, r_1, r_2, \dots, r_\varrho) \text{ die Inhalte der ersten } \varrho \text{ Register,} \\ S &= (s_1, s_2, s_3, \dots, s_\sigma) \text{ die Inhalte der ersten } \sigma \text{ Zellen aus } S, \\ E &= (e_k, e_{k+1}, \dots, e_\eta) \text{ die } \eta - k + 1 \text{ noch zu lesenden Eingabezellen,} \\ A &= (a_1, a_2, \dots, a_\lambda) \text{ die Inhalte der bisher beschriebenen } \lambda \text{ Ausgabezellen.} \end{aligned}$$

ϱ und σ sind die maximalen Indizes aller bisher verwendeten Register- bzw. Speicherzellen. Die Inhalte der bisher nicht verwendeten Zellen mit einem Index kleiner ϱ bzw. kleiner σ seien dabei 0. Die Startkonfiguration einer Maschine sei $K_0 = ((1), \varepsilon, (e_1, e_2, \dots, e_\eta), \varepsilon)$. Im Falle eines maximalen Index 0 wird die entsprechende Koordinate (außer R) mit dem leeren Wort ε identifiziert. Die Menge aller Konfigurationen wird mit $\mathbf{K}$ bezeichnet.

2.2.1 Operationen der Modellmaschine

Tabelle 1 ordnet den Operationen der Modellmaschine eine Semantik $\mathscr{S} : \mathbf{O} \to (\mathbf{K} \to \mathbf{K})$ zu. Die (partiellen) Funktionen $\mathscr{S}[\![o]\!] : \mathbf{K} \to \mathbf{K}$ werden tabellarisch durch Angabe der Folgekonfiguration K_{i+1} der als gegenwärtig angenommenen Konfiguration K_i definiert. Dabei werde in R bzw. S von K_{i+1} jeweils der Inhalt der j-ten Zelle verändert ($j = 0$ für die Instruktionen `jmp, jz, jgz, jlz`, $j > 0$ sonst). Darüber hinaus seien $rop, rop_i \in \mathbf{R}$, $sop \in \{S_j : j \in \mathbb{N} \cup \mathbf{R}\}$ und $l \in \mathbb{N}$. Formal wird den zweistelligen Operationen als drittem Operanden die Zahl 0 zugeordnet, im Falle von `end` 0,0,0. Falls bei den Sprung-Instruktionen `jmp, jz, jgz, jlz` für ein l und ein Programm $P = (p_1, \dots, p_\kappa)$ gilt $l \notin \{1, \dots, \kappa\}$, so sei die Folgekonfiguration definiert als $((0, \dots), S, E, A)$ (Abfangen illegaler Sprünge durch Halten der Maschine). Dieselbe Folgekonfiguration sei auch für Anweisungen definiert, die auf nicht existierende Register zugreifen. Es gibt keine Operation zur Änderung des Programmspeichers. Daraus folgt, daß eine programmierte *RX-RAM* eher als *RAM* denn als *RASP* zu verstehen ist. Eine Diskussion folgt in 2.3.

Operation o_i	Bedeutung	Folgekonfiguration $K_{i+1} = \mathscr{S}[o_i](K_i)$
$(\mathtt{mov}, R_j, rop, 0)$	Verschiebung	$((r_0+1, \dots, c(rop), \dots), S, E, A)$
$(\mathtt{ld}, R_j, sop, 0)$	Laden	$((r_0+1, \dots, c(sop), \dots), S, E, A)$
$(\mathtt{st}, S_j, rop, 0)$	Speichern	$((r_0+1, \dots), (\dots, c(rop), \dots), E, A)$
$(\mathtt{add}, R_j, rop_1, rop_2)$	Addition	$((r_0+1, \dots, c(rop_1)+c(rop_2), \dots), S, E, A)$
$(\mathtt{sub}, R_j, rop_1, rop_2)$	Subtraktion	$((r_0+1, \dots, c(rop_1)-c(rop_2), \dots), S, E, A)$
$(\mathtt{mul}, R_j, rop_1, rop_2)$	Multiplikation	$((r_0+1, \dots, c(rop_1)\cdot c(rop_2), \dots), S, E, A)$
$(\mathtt{div}, R_j, rop_1, rop_2)$	Division	$((r_0+1, \dots, \lfloor c(rop_1)/c(rop_2)\rfloor, \dots), S, E, A)$
$(\mathtt{mod}, R_j, rop_1, rop_2)$	Rest nach Division	$((r_0+1, \dots, c(rop_1)\%c(rop_2), \dots), S, E, A)$
$(\mathtt{and}, R_j, rop_1, rop_2)$	bitweises Und	$((r_0+1, \dots, c(rop_1)\wedge c(rop_2), \dots), S, E, A)$
$(\mathtt{or}, R_j, rop_1, rop_2)$	bitweises, inkl. Oder	$((r_0+1, \dots, c(rop_1)\vee c(rop_2), \dots), S, E, A)$
$(\mathtt{not}, R_j, rop, 0)$	bitweise Negation	$((r_0+1, \dots, \neg c(rop), \dots), S, E, A)$
$(\mathtt{xor}, R_j, rop_1, rop_2)$	bitweises, exkl. Oder	$((r_0+1, \dots, c(rop_1)\tilde{\vee}c(rop_2), \dots), S, E, A)$
$(\mathtt{shl}, R_j, rop_1, rop_2)$	bitweiser Linksshift	$((r_0+1, \dots, c(rop_1)\ll c(rop_2), \dots), S, E, A)$
$(\mathtt{shr}, R_j, rop_1, rop_2)$	bitweiser Rechtsshift	$((r_0+1, \dots, c(rop_1)\gg c(rop_2), \dots), S, E, A)$
$(\mathtt{eq}, R_j, rop_1, rop_2)$	Gleichheit	$((r_0+1, \dots, 1, \dots), S, E, A)$, falls $rop_1 = rop_2$ $((r_0+1, \dots, 0, \dots), S, E, A)$, sonst
$(\mathtt{neq}, R_j, rop_1, rop_2)$	Ungleichheit	$((r_0+1, \dots, 1, \dots), S, E, A)$, falls $rop_1 \neq rop_2$ $((r_0+1, \dots, 0, \dots), S, E, A)$, sonst
$(\mathtt{gt}, R_j, rop_1, rop_2)$	Vergleich	$((r_0+1, \dots, 1, \dots), S, E, A)$, falls $rop_1 > rop_2$ $((r_0+1, \dots, 0, \dots), S, E, A)$, sonst
$(\mathtt{lt}, R_j, rop_1, rop_2)$	Vergleich	$((r_0+1, \dots, 1, \dots), S, E, A)$, falls $rop_1 < rop_2$ $((r_0+1, \dots, 0, \dots), S, E, A)$, sonst
$(\mathtt{geq}, R_j, rop_1, rop_2)$	Vergleich	$((r_0+1, \dots, 1, \dots), S, E, A)$, falls $rop_1 \geq rop_2$ $((r_0+1, \dots, 0, \dots), S, E, A)$, sonst
$(\mathtt{leq}, R_j, rop_1, rop_2)$	Vergleich	$((r_0+1, \dots, 1, \dots), S, E, A)$, falls $rop_1 \leq rop_2$ $((r_0+1, \dots, 0, \dots), S, E, A)$, sonst
$(\mathtt{jmp}, R_0, l, 0)$	unbed. Sprung	$((l, \dots), S, E, A)$
$(\mathtt{jz}, R_0, l, rop_1)$	bedingter Sprung	$((l, \dots), S, E, A)$, falls $rop_1 = 0$ $((r_0+1, \dots), S, E, A)$, sonst
$(\mathtt{jgz}, R_0, l, rop_1)$	bedingter Sprung	$((l, \dots), S, E, A)$, falls $rop_1 > 0$ $((r_0+1, \dots), S, E, A)$, sonst
$(\mathtt{jlz}, R_0, l, rop_1)$	bedingter Sprung	$((l, \dots), S, E, A)$, falls $rop_1 < 0$ $((r_0+1, \dots), S, E, A)$, sonst
$(\mathtt{get}, R_j, E_k, 0)$	Eingabe	$((r_0+1, \dots, e_k, \dots), S, (e_{k+1}, \dots, e_{\lvert E\rvert}), A)$
$(\mathtt{put}, A_k, rop_1, 0)$	Ausgabe	$((r_0+1, \dots), S, E, (a_1, a_2, \dots, a_{\lvert A\rvert}, c(rop_1)))$
$(\mathtt{end}, 0, 0, 0)$	Halten	$((0, r_1, r_2, \dots), S, E, A)$

Tabelle 1: Operationen der Modellmaschine *RX-RAM*

Definition 2.8 (Binäre Operationen) Es seien z_1 und z_2 zwei ganze Zahlen mit Binärdarstellung $z_1 = (x_m, x_{m-1}, \dots, x_1, x_0)$, $z_2 = (y_n, y_{n-1}, \dots, y_1, y_0)$, wobei $x_i, y_i \in \mathbb{Z}_2, x_k = 0 \ \forall k > m$ und $y_k = 0 \ \forall k > n$. Dann sind die bitweisen Operationen aus Tabelle 1 wie folgt definiert:

$$
\begin{array}{lcl}
z_1 \wedge z_2 & = & (x_{min(m,n)} \cdot y_{min(m,n)}, \dots, x_1 \cdot y_1, x_0 \cdot y_0) \\
z_1 \vee z_2 & = & (x_{\max(m,n)} \cdot y_{\max(m,n)} + x_{\max(m,n)} + y_{\max(m,n)}, \dots, x_0 \cdot y_0 + x_0 + y_0) \\
z_1 \tilde{\vee} z_2 & = & (x_{\max(m,n)} + y_{\max(m,n)}, \dots, x_1 + y_1, x_0 + y_0) \\
\neg z_1 & = & (x_m + 1, x_{m-1} + 1, \dots, x_1 + 1, x_0 + 1), \text{ für } z_1 \neq 0 \\
 & & (1), \text{ falls } z_1 = 0 \\
z_1 \ll z & = & (x_m, x_{m-1}, \dots, x_0, \underbrace{0, 0, \dots, 0}_{z}), \text{ falls } z \geq 0, \text{ sonst } z_1 \gg -z \\
z_1 \gg z & = & (x_m, x_{m-1}, \dots, x_z), \text{ falls } z \geq 0, \text{ sonst } z_1 \ll -z
\end{array}
$$

(Multiplikation und Addition finde dabei in $\mathbb{Z}_2$ statt.)

Ein *Rechenschritt* ist ein Konfigurationsübergang einer Modellmaschine gemäß der Definition der Operationssemantik $\mathscr{S}$, also $K_{j+1} = \mathscr{S}[\![p_{r_0}]\!](K_j)$.

Definition 2.9 (Semantik von *RX-RAM*-Programmen)
Es sei $P = (p_1, p_2, \dots, p_\kappa)$ ein Programm und $K_0 = ((1), \varepsilon, (e_1, e_2, \dots, e_\eta), \varepsilon)$ eine Anfangskonfiguration. Dann sei die zugehörige Berechnung $(K_0, \dots, K_j, \dots)$ festgelegt durch

$$K_{j+1} = \begin{cases} \mathscr{S}[\![p_{K_j[1][1]}]\!](K_j) & \text{falls } K_j \text{ keine Endkonfiguration ist} \\ K_j & \text{sonst} \end{cases}$$

Dann wird die Erweiterung der Operationssemantik folgendermaßen festgelegt:

$$\mathscr{S}^* \, [\![P\,]\!](K_0) = \begin{cases} K_{end} & \text{falls } end = \min\{j : K_j \text{ ist Endkonfiguration}\} \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

Dann ist die Abbildung $\mathscr{R}[\![P]\!]$ für ein gegebenes Programm P definiert durch die Inhalte des Ausgabebandes $\mathbf{A}$ der Endkonfiguration K_{end} bei Eingabe $E = (e_1, e_2, \dots, e_\eta)$, also

$$\mathscr{R}[\![P]\!](E) = \begin{cases} (\mathscr{S}^*[\![P]\!](((1), \varepsilon, E, \varepsilon)))[4] & \text{falls } S^*[\![P]\!](((1), \varepsilon, E, \varepsilon))) \text{ definiert} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Beispiel 2.10
Gegeben sei eine *RX-RAM* $(\{R_0, R_1, R_2\}, \{S_1, S_2, \dots\}, \{E_1, E_2, \dots\}, \{A_1, \dots\}, \mathbf{I}, \mathbf{O}, \mathbf{P})[\![P]\!]$. Das Programm P sei dabei wie folgt definiert:

$$
\begin{aligned}
p_1 &= (\texttt{get}, R_1, E_1, 0)\\
p_2 &= (\texttt{get}, R_2, E_2, 0)\\
p_3 &= (\texttt{add}, R_1, R_1, R_2)\\
p_4 &= (\texttt{sub}, R_2, R_1, 100)\\
p_5 &= (\texttt{jgz}, R_0, 8, R_2)\\
p_6 &= (\texttt{put}, A_1, 100, 0)\\
p_7 &= (\texttt{jmp}, R_0, 9, 0)\\
p_8 &= (\texttt{put}, A_1, R_1, 0)\\
p_9 &= (\texttt{end}, 0, 0, 0)
\end{aligned}
$$

P bildet die Summe der Eingabewerte e_1 und e_2, vergleicht diese mit 100 und gibt den größeren Wert nach A_1 aus. Eine Beispielrechnung mit Eingabewerten 17 und 4 sieht dabei wie folgt aus:

$$
\begin{aligned}
K_0 &= ((1), \varepsilon, (17, 4), \varepsilon)\\
K_1 &= ((2, 17), \varepsilon, (4), \varepsilon)\\
K_2 &= ((3, 17, 4), \varepsilon, \varepsilon, \varepsilon)\\
K_3 &= ((4, 21, 4), \varepsilon, \varepsilon, \varepsilon)\\
K_4 &= ((5, 21, -79), \varepsilon, \varepsilon, \varepsilon)\\
K_5 &= ((6, 21, -79), \varepsilon, \varepsilon, \varepsilon)\\
K_6 &= ((7, 21, -79), \varepsilon, \varepsilon, (100))\\
K_7 &= ((9, 21, -79), \varepsilon, \varepsilon, (100))\\
K_8 &= ((0, 21, -79), \varepsilon, \varepsilon, (100))
\end{aligned}
$$

Die Semantik des obigen Beispielprogramms P ist

$$\mathscr{R}[\![P]\!]((17, 4)) = (\mathscr{S}^*[\![P]\!](((1), \varepsilon, (17, 4), \varepsilon)))[4] = ((0, 21, -79), \varepsilon, \varepsilon, (100))[4] = (100)$$

2.2.2 Kosten von Maschinenoperationen

Es werden nun den Operationen der Modellmaschine *Kosten* zugeordnet. In der Literatur werden im allgemeinen zwei verschiedene Anschauungen vertreten:

Das *gleichförmige* Kostenkriterium ordnet jeder Operation genau eine zeitliche Kosteneinheit und jeder Zelle genau eine räumliche Kosteneinheit zu. Der Vorteil dieses Vorgehens ist die Einfachheit der Bestimmung der Komplexitäten. Was diese Art der Kostendefinition aber unrealistisch macht, ist die Tatsache, daß dabei weder Rücksicht auf die Größe der Operanden noch auf die Art der Operation genommen wird. Tatsächlich spielt aber beides in der Realität eine entscheidende Rolle.

Das *logarithmische* Kostenkriterium definiert die Kosten jeweils in Abhängigkeit der Länge der auftretenden Operanden. Dies führt prinzipiell zu einer besseren Übereinstimmung der Komplexitätsvoraussagen mit den zu beobachtenden Laufzeiten und dem Speicheraufwand von Implementierungen eines Verfahrens. Der Nachteil des logarithmischen Kriteriums ist die Schwierigkeit der genauen Bestimmung der Komplexitäten an sich, da gerade über die Länge eines Operanden häufig keine genaue Aussage getroffen werden kann.

Der hier definierte Kostenbegriff orientiert sich am logarithmischen Kostenkriterium. Es wird nun zunächst der zeitliche Aufwand von Rechenschritten der RX-RAM definiert. Neben den eigentlichen Kosten bei der Ausführung einer Operation wird dabei berücksichtigt, daß Speicherzugriffe sowie Ein- und Ausgabe zusätzliche Zeit benötigen. Darüber hinaus wird die Art der Operation unterschieden. Einfachen Operationen wie Additionen werden geringere Kosten zugeordnet als etwa Multiplikationen.

Eine spätere Vereinfachung, die sich hauptsächlich aus der Beschränkung der Größe realer Rechnerworte ergibt, wird das verwendete Maß zwar wieder in die „Nähe" des gleichförmigen Kostenkriteriums rücken. Allerdings werden nach wie vor die Zugriffskosten sowie die Unterschiede der jeweiligen Operationen beachtet. Das resultierende Modell wird es erlauben, recht präzise Aussagen über die Komplexitäten der Verfahren zu treffen.

Definition 2.11 (Zugriffskosten auf Operanden) Es sei $F \in \mathbf{F}$. Dann bezeichnet $t_{op} : \mathbf{F} \to \mathbb{N}_0$ definiert durch

$$t_{op}(F) = \begin{cases} 0 & \text{falls } F \in \mathbb{Z} \text{ oder } F \in \mathbf{R} \\ \lg(j) & \text{falls } F = E_j \text{ oder } F = A_j \text{ oder } F = S_j \\ \lg(r_j) & \text{falls } F = S_{R_j}. \end{cases}$$

die *Zugriffskosten* auf den Operanden F.

Bemerkung. Es wird zwischen den verschiedenen Arten der Operanden unterschieden. Der Zugriff auf die Registerzellen erfolgt genauso schnell wie auf Konstanten, nämlich sofort. Dahingegen ist „gewöhnlicher" Speicher sowie Ein- und Ausgabe relativ langsam. Notwendige Adreßberechnungen spiegeln sich in den Zugriffskosten auf Nicht-Registerzellen wider. Daß dem indirekten Speicherzugriff dabei keine höheren Kosten als dem direkten Zugriff beigemessen wird, liegt an der (realistischen) Einschränkung, daß sich Adressen dabei bereits in Registern befinden und der Zugriff auf diese sofort erfolgt.

Es folgt nun die Zuordnung von Kosten zu den verschiedenen Operationen der Modellmaschine.

Definition 2.12 (Ausführungskosten, Zugriffskosten von Operationen)
Die *Ausführungskosten* $t_a : \mathbf{O} \to \mathbb{N}$ sowie die *Zugriffskosten* $t_z : \mathbf{O} \to \mathbb{N}_0$ von Operationen der Modellmaschine sind durch Tabelle 2 bzw. 3 angegeben. Dabei sei $uninstr \in \{\texttt{mov}, \texttt{not}, \texttt{jz}, \texttt{jgz}, \texttt{jlz}\}$, $bininstr \in \{\texttt{add}, \texttt{sub}, \texttt{and}, \texttt{or}, \texttt{xor}, \texttt{shl}, \texttt{shr}, \texttt{eq}, \texttt{neq}, \texttt{geq}, \texttt{leq}, \texttt{lt}, \texttt{gt}\}$, rop, rop_1 und $rop_2 \in \mathbf{R}$ und $sop \in \{S_j : j \in \mathbb{N} \cup \mathbf{R}\}$.

Operation $o \in \mathbf{O}$	Zugriffskosten $t_z(o)$
$(\texttt{ld}, R_j, sop, 0)$	$t_{op}(sop)$
$(\texttt{st}, sop, rop, 0)$	$t_{op}(sop)$
$(\texttt{get}, R_j, E_i, 0)$	$t_{op}(E_i)$
$(\texttt{put}, A_i, rop, 0)$	$t_{op}(A_i)$

Tabelle 2: Zugriffskosten von *RX-RAM*-Operationen

Operation $o \in \mathbf{O}$	Ausführungskosten $t_a(o)$
$(\texttt{ld}, R_j, sop, 0)$	$\lg(c(sop))$
$(\texttt{st}, sop, rop, 0)$	$\lg(c(rop))$
$(\texttt{mul}, R_j, rop_1, rop_2)$	$M(\lg(c(rop_1)), \lg(c(rop_2)))$
$(\texttt{div}, R_j, rop_1, rop_2)$	$d \cdot M(\lg(c(rop_1)), \lg(c(rop_2)))$
$(\texttt{mod}, R_j, rop_1, rop_2)$	$d \cdot M(\lg(c(rop_1)), \lg(c(rop_2)))$
$(\texttt{jmp}, R_0, b)$	1
$(\texttt{get}, R_j, E_i, 0)$	$\lg(e_i)$
$(\texttt{put}, A_i, rop, 0)$	$\lg(c(rop))$
$(\texttt{end}, 0, 0, 0)$	1
$(uninstr, R_j, rop, 0)$	$\lg(c(rop))$
$(bininstr, R_j, rop_1, rop_2)$	$\lg(c(rop_1)) + \lg(c(rop_2))$

Tabelle 3: Ausführungskosten von *RX-RAM*-Operationen

Bemerkung. Die Kosten, die die verschiedenen Operationen verursachen, sind folgendermaßen motiviert: Einfachen, binären Operationen wie Addition und bitweiser Verknüpfung/Verschiebung wird die Summe der Größen ihrer Operanden in Bits als Kosten zugeordnet. Dies entspricht der naiven Berechnung „auf dem Papier“. Die Kosten $M(\lg(rop_1), \lg(rop_2))$ der Multiplikation seien an dieser Stelle nur durch $k \cdot n^{\mu}$ mit $n = \max(\lg(c(rop_1)), \lg(c(rop_2)))$, $1 \leq \mu \leq 2$ und konstantem $k > 1$ eingeschränkt ($\mu = 2$ entspräche dabei der „Schulmultiplikation“). Die Division unterscheide sich von der Multiplikation nur durch einen konstanten Faktor d, hier sei nur $d > 1$ festgelegt. Eine genaue Diskussion von M und d findet in Abschnitt 2.3.3 statt. Den Sprung- und Zuweisungsoperationen werden nur geringe Kosten zugeordnet. Die Realitätsnähe der hier definierten zeitlichen Kosten wird im Abschnitt 2.3.4 betrachtet.

Die Summen der jeweiligen Spalten bilden nun die Gesamtkosten, die eine Operation verursacht:

Definition 2.13 (Gesamtkosten einer Operation) Die *Gesamtkosten* t_g einer Operation $o \in \mathbf{O}$ sind definiert durch

$$t_g(o) = t_z(o) + t_a(o)$$

Es soll nun die Dauer einer Rechnung der *RX-RAM* definiert werden.

Definition 2.14 (Laufzeit) Die Laufzeit rt einer Rechnung $(K_j)_{j=0}^{end}$ ist definiert durch

$$rt((K_j)) = \sum_{j=0}^{end-1} t_g(o_j),$$

wobei o_j die Operation beim Konfigurationsübergang von K_j nach K_{j+1} Operation in der Rechnung ist.

Bemerkung. Unter Verwendung des gleichförmigen Kostenkriteriums würde man die Anzahl der Rechenschritte als Laufzeit definieren.

Beispiel 2.15 Die Laufzeit der Rechnung im Beispiel ergibt sich wie folgt:

o	$t_z(o)$	$t_a(o)$	
$(\mathtt{get}, R_1, E_1, 0)$	1	5	
$(\mathtt{get}, R_2, E_2, 0)$	2	3	
$(\mathtt{add}, R_1, R_1, R_2)$	0	8	
$(\mathtt{sub}, R_2, R_1, 100)$	0	12	
$(\mathtt{jgz}, R_0, 8, R_2)$	0	7	
$(\mathtt{put}, A_1, 100, 0)$	1	7	
$(\mathtt{jmp}, R_0, 9, 0)$	0	1	
$(\mathtt{end}, 0, 0, 0)$	0	1	
$rt((K_j))$	4	$+\ 44$	$=48$

Im Falle, daß nur ein Register vorhanden ist, erhöht sich die Laufzeit (durch Speicherzugriff) bereits auf 54.

Die folgende Definition legt das Maß für den während einer Berechnung erforderlichen Speicher fest.

Definition 2.16 (Speicheraufwand) Es sei (K_j) eine Rechnung mit Endkonfiguration $K_{end} = ((0, r_1, r_2, \ldots, r_\varrho), (s_1, s_2, \ldots, s_\sigma), \varepsilon, (a_1, \ldots, a_\lambda))$. Dann ist der *Speicheraufwand* rs von (K_j) definiert durch

$$\begin{aligned} rs((K_j)) = & \sum_{k=1}^{\sigma} \max_{0 \le i \le end} \Big\{ \lg(s_k) : s_k \text{ ist Inhalt von } S_k \text{ in } K_i \} \Big\} \\ & + \sum_{k=1}^{\varrho} \max_{0 \le i \le end} \Big\{ \lg(r_k) : r_k \text{ ist Inhalt von } R_k \text{ in } K_i \} \Big\} \end{aligned}$$

Bemerkung. Der Speicheraufwand einer Rechnung ergibt sich also, indem man für jede einmal verwendete Zelle das Maximum der Stellen aller Zahlen addiert, die in der Zelle während der Rechnung vorkamen. Dabei gehen alle nicht verwendeten Zellen, die jedoch einen Index kleiner als eine verwendete Zelle haben, mit einem Beitrag von 1 ein. An dieser Stelle tritt eine ungewöhnliche Eigenschaft der *RX-RAM* (bzw. jeder Maschine mit obigem Raummaß) zutage: In polynomieller Laufzeit ist es möglich, exponentiell viel Speicher zu verwenden.

Beispiel 2.17 Der Anteil von Register R_1 zur Speicherkomplexität beträgt 6, der von Register R_2 8, insgesamt beträgt der Speicheraufwand des Beispiels also 14.

Sowohl Zeit- als auch Raumkomplexität von *RX-RAM*-Programmen sollen nun formal definiert werden. Sie werden jeweils über die Länge der Eingabe festgelegt, die wiederum als Summe der Längen der einzelnen Eingabezellen zu verstehen ist. Beide Definitionen gehen vom jeweils schlechtesten Fall aus, der für sämtliche Rechnungen bei fester Eingabelänge möglich ist. Es werden dabei nur korrekte Programme betrachtet, d.h. illegale Registerzugriffe seien ausgeschlossen.

Es sei in den folgenden Definitionen $\mathscr{R}[\![P]\!]$ eine *RX-RAM* mit Startkonfiguration $K_0 = ((1), \varepsilon, (e_1, e_2, \ldots, e_\eta), \varepsilon)$.

Definition 2.18 (Zeitkomplexität)
Die *Zeitkomplexität* $t(n)$ eines *RX-RAM*-Programms P sei definiert durch

$$t(n) = \max \Big\{ rt((K_j)) : (K_j) \text{ ist Berechnung}, \sum_{i=1}^{\eta} \lg(e_i) = n \Big\}$$

Im Falle, daß die rechte Seite dabei undefiniert ist, setze $t(n) = \infty$.

Bemerkung. Die Zeitkomplexität ist also das Maximum aller möglichen Laufzeiten für Eingaben der Länge n, wobei hiermit die Summe der Längen aller Eingabezellen gemeint ist.

Definition 2.19 (Raumkomplexität)
Die *Raumkomplexität* $s(n)$ eines *RX-RAM*-Programms P sei definiert durch

$$s(n) = \max\left\{rs((K_j)) : (K_j) \text{ ist Berechnung}, n = \sum_{i=1}^{\eta} \lg(e_i)\right\}$$

Im Falle, daß die rechte Seite undefiniert ist, setze $s(n) = \infty$.

Bemerkung. Alternativ zu obigen Definitionen wäre es möglich, die Komplexitäten als durchschnittlich zu erwartende Werte festzulegen.

2.3 Diskussion der Modellmaschine *RX-RAM*

2.3.1 Vergleich mit anderen Modellen

Eine Definition eines Berechnungsmodells, das sich an realen Maschinen orientiert, findet sich erstmals in [Elg64]. Dabei handelt es sich um ein Modell, bei dem das Programm selbst im Speicher liegt (*Random Access Stored Program Machine*, *RASP*) und veränderbar ist. Ein Maschine mit fest verdrahtetem Programm wird als *Random Access Machine* (*RAM*) bezeichnet. Die erste *RAM* geht auf [Coo73] zurück.

Da sich in einer *RX-RAM* zwar ein Programmspeicher befindet, dieser aber nach einer Programmierung nicht veränderlich ist, ist eine programmierte *RX-RAM* eher als *RAM* denn als *RASP* zu verstehen. Die Unveränderlichkeit des Programms stellt allerdings keine Einschränkung dar. In [Aho74] wird gezeigt, daß *RASP*-Modelle linear zeitverknüpft mit *RAM*s sind.

Umfangreiche Diskussionen und Vergleiche auch mit anderen Berechnungsmodellen finden sich z.B. in [Wag86], [Rei90]. Die verschiedenen Beschreibungen der Modellmaschine *RAM* entsprechen meist einer programmierten *RX-RAM* mit einem oder gar keinem ausgezeichneten Rechenregister. Die fehlende Existenz mehrerer Register erfordert die Möglichkeit, Speicherzellen direkt als Operanden zuzulassen, was zumindest dann kritikwürdig erscheint, wenn dies nicht in der Definition der Zugriffskosten berücksichtigt wird. Oft wird auch auf die Unterscheidung von Speicher und Registern ganz verzichtet. Mehrere Rechenregister, die das vielfache Verschieben von Operanden überflüssig machen, sparen Zugriffskosten und beschleunigen eine Rechnung der *RX-RAM* gegenüber Einregistermaschinen um einen konstanten Faktor (siehe 2.20).

Häufig dienen Berechnungsmodelle eher theoretischen Betrachtungen denn als Hilfsmittel zur Einschätzung praktischer Laufzeiten. Allerdings gibt es auch Beschreibungen, die sehr praktisch orientiert sind. So wird beispielsweise in [Knu81a] ein Modell definiert, das eine „Mischung“ aus 16 (!) realen Maschinen darstellt.

Neben den Unterschieden in der Definition von Zugriffskosten – falls überhaupt vorhanden – spielt die Zuordnung der Ausführungskosten an die Operationen eine entscheidende Rolle. Die Annahme, daß sich die multiplikativen Operationen in linearer Zeit bewältigen lassen, entspricht nicht dem heutigen Wissensstand (siehe auch 2.3.3 und 2.3.4). Die hier getroffene Einschränkung der Komplexität von Multiplikation und Division erzeugt eine maximal polynomielle Zeitverknüpfung zwischen *RX-RAM* und *RAM*, was hier festgehalten werden soll:

Behauptung 2.20 *RAM* und *RX-RAM* sind maximal polynomiell zeitverknüpft.

Beweis. Es genügt, jede Operation der Form $(instr, res, op_1, op_2)$ durch die folgenden drei Operationen zu ersetzen: $(ld,\ reg, op_1)$ $(instr,\ reg, op_2)$ $(st,\ res, reg)$. Im Falle, daß Multiplikation und Division in der *RAM*-Definition als in logarithmisch linearer Zeit durchführbar festgelegt sind, ergibt sich also eine maximal polynomielle Zeitverknüpfung. □

Eine exakte Diskussion um die Auswirkung der Anzahl der Register auf die Zeitkomplexität von Programmen scheint wenig aussichtsreich, da sich immer ein Programm angeben läßt, das auf regulären Speicher zugreifen muß. Beobachtungen und Analysen des Assemblercodes realer Programme zeigen jedoch, daß die Anzahl der Speicherzugriffe auf Operanden arithmetischer Operationen im Durchschnitt sehr gering ist.

Im folgenden sei eine *RAM* gleichbedeutend mit einer *RX-RAM*, die nur ein einziges Register besitzt. Dabei werde das logarithmische Kostenkriterium zugrunde gelegt und keine Festlegung der Kosten von multiplikativen Operationen der *RAM* getroffen.

2.3.2 Berechnungsuniversalität

Satz 2.21 (Berechnungsuniversalität)
Die Modellmaschine *RX-RAM* ist berechnungsuniversell.

Beweis. Eine *RX-RAM* kann eine Turingmaschine mit k Bändern auf folgende Weise simulieren: Das i-te Band sei in Speicherzelle S_i dargestellt, die Stellen von S_i repräsentieren dabei die Zellen des i-ten Bandes. Die Positionen der Köpfe des i-ten Bandes seien in S_{k+i} abgelegt. Der Zugriff auf die einzelnen Stellen einer Speicherzelle und damit auf die Zellen der Turingmaschinenbänder kann durch die *RX-RAM*-Instruktionen `shl` und `and` erfolgen. Der aktuelle Zustand der Turingmaschine sei z.B. in S_{2k+1} gespeichert. Das *RX-RAM*-Programm könnte nun folgendermaßen vorgehen, um einen Turingmaschinen-Übergang zu simulieren: Zunächst werden die aktuellen Einträge der k-Bänder in Zelle S_{2k+2} abgelegt. Dann wird der aktuelle Zustand aus S_{2k+1} angehängt. Die Überführungsfunktion der Turingmaschine befinde sich in S_{2k+3} bis $S_{2\cdot(2k+3+2^k\cdot|Z|)}$, wobei

$|Z|$ die Mächtigkeit der Zustandsmenge sei. Diese muß nun durchlaufen und nach dem Argument s_{2k+2} durchsucht werden, um die Konsequenzen (z.B. ähnlich codiert in einer Zahl) zu ermitteln und in den Registern zu realisieren. Auf die formale Übersetzung des Turingmaschinenprogramms soll nicht verzichtet werden, sie erfolgt jedoch als Nachtrag, sobald eine höhere Programmiersprache zur Beschreibung des Verfahrens zur Verfügung steht. □

Bemerkung. Auch die Umkehrung gilt: Eine Turingmaschine kann eine *RX-RAM* simulieren. Beweise für *RAM*s, die sich nach 2.20 vom hier vorgestellten Modell ja maximal polynomiell in ihrer Zeitkomplexität unterscheiden, finden sich z.B. in [Aho74] und [Pau78]. Dort wird auch gezeigt, daß Turingmaschine und RAM polynomiell zeitverknüpft sind, d.h. für jede *RAM* gibt es eine Turingmaschine, die sie in polynomieller Laufzeit simuliert und umgekehrt.

RX-RAM und Turingmaschine sind also von ihrer Leistung her – womit die Fähigkeit zur Berechnung von Funktionen gemeint ist – gleichwertig. Alle turingberechenbaren Funktionen sind auch *RX-RAM*-berechenbar.

2.3.3 Multiplikation und Division

Die Kostenfunktion M der Multiplikations- und Divisionsoperationen der *RX-RAM* soll zunächst nicht genauer festgelegt werden. Eine mögliche Festlegung, die der Realität nahe kommt, ergäbe sich aus der Methode von *Karatsuba* [Knu81b]:

$$M(\lg(c(rop_1)), \lg(c(rop_2))) = 2^{k \log_2 3}$$

mit $k - 1 < \max(\lg(rop_1), \lg(rop_2)) \leq k$. Da $\log_2 3 \approx 1,59$, ist dies wesentlich schneller als die Schulmultiplikation. Es gibt zwar Algorithmen, die die Multiplikation noch schneller ausführen (z.B. *Toom-Cook* ($O(\lg(n)^{1+\frac{3,5}{\sqrt{\log(\log(n))}}} \log(\log(n)))$), oder *Schönhage-Strassen* ($O(\log(n)\log(\log(n))\log(\log(\log(n))))$)), diese lohnen sich aber erst bei sehr großen Operanden. Eine Berücksichtigung solcher Verfahren bei hauptsächlich kleinen Operanden führt zu allgemein unrealistischen Aussagen. Sie sind aber durch die Einschränkung von M in 2.2.2 nicht ausgeschlossen. Daß die Division bis auf einen konstanten Faktor – hier mit d bezeichnet – dieselbe Zeit wie die Multiplikation benötigt, wird z.B. in [Knu81b] gezeigt und d nach oben auf 8 beschränkt. In [Knu81b] und [Aho74] finden sich detaillierte Diskussionen verschiedener Multiplikations- und Divisionsalgorithmen. Auch auf eine genauere Festlegung von d soll an dieser Stelle verzichtet werden (zum Vorteil zunächst unterlassener Festlegungen siehe auch 2.3.4).

2.3.4 Realitätsnähe

Heutige Prozessoren realer Rechner verfügen über eine Vielzahl an schnellen Rechenregistern. Die meisten Berechnungen können innerhalb des Prozessors ohne zusätzliche, teure Speicherzugriffe stattfinden. Diese Tatsache wird im vorgestellten Modell (insbesondere auch im nächsten Abschnitt) berücksichtigt. Andererseits sind keine Algorithmen bekannt, die in linearer Zeit multiplizieren oder dividieren können. Die in der Beispieldefinition von M auftauchende „Teile und herrsche“ -Methode von *Karatsuba* wird heute häufig softwaremäßig implementiert, falls die Operanden die Maschinenwortlänge überschreiten (z.B. bei *long long*-Datentypen). Tabelle 4 gibt einen Vergleich der durchschnittlichen Laufzeiten verschiedener Operationen auf realen Maschinen. Dabei wurde die Zeit einer Addition als Einheit festgelegt. Die Durchführung der Tests fand auf sieben verschiedenen Rechnern statt, die mit fünf Prozessortypen unterschiedlicher Bauarten ausgerüstet waren.

Operation	Dauer
Addition	1
Multiplikation	4,5
Division	21
Sequentieller Speicherzugriff	3,5

Tabelle 4: Relative, reale Laufzeiten von Operationen

Modulorechnung und bitweise Operationen entsprachen dabei den Werten für Division bzw. Addition.

Die Analyse der vorzustellenden Algorithmen wird zunächst unabhängig von einer genauen Festlegung der Kosten für Multiplikation und Division vorgenommen und erst später anhand von Messungen auf realen Maschinen bewertet werden. Diese Vorgehensweise läßt eine spätere Aussage zu Laufzeiten auch dann noch zu, wenn sich die Kosten der multiplikativen Operationen und der Speicherzugriffe in der Realität geändert haben sollten. Additionen kosten auf realen Maschinen etwa genausoviel wie die sonstigen, elementaren Operationen, was sich im hier festgelegten Kostenmaß widerspiegelt. Sprungoperationen sind keine zusätzlichen Kosten beigemessen. Techniken wie *Pipelinening* sorgen in der Realität dafür, daß der Zugriff auf die als nächstes auszuführende Operation tatsächlich sehr schnell erfolgt. Die Festlegung eines Beitrages zur Raumkomplexität nicht wirklich verwendeter Zellen, die allerdings einen kleineren Index als eine andere, verwendete Zelle besitzen, ist nicht unrealistisch, da in einem Programm häufig Lücken im Speicher in Kauf genommen werden müssen. Das logarithmische Maß an sich ist allerdings bei der Bewertung des Speicheraufwandes insofern nicht unproblematisch, als daß in realen Maschinen Speicherplätze immer eine feste Größe in Form von Maschinenworten einnehmen. Auf diese Tatsache wird im folgenden Abschnitt noch Rücksicht genommen.

2.4 Die Algorithmen-Beschreibungssprache *ADL*

Maschinenprogramme sind aufgrund ihrer Unübersichtlichkeit zur Beschreibung von Algorithmen ungeeignet. Auch wenn es in der Praxis notwendig sein kann, auf Assemblersprachen zurückzugreifen (siehe Kapitel 6), ist es im allgemeinen günstiger, zur Implementierung eines Verfahrens eine höhere Sprache zur Verfügung zu haben. Zu diesem Zweck wird nun die Sprache *ADL* (*Algorithm Description Language*) entwickelt, die an die realen Programmiersprachen *C* und *Pascal* angelehnt ist und dabei versucht, vorteilhafte Konzepte beider zu vereinigen. Dabei wird auf die Einführung verschiedener Datentypen und daher auch auf Deklarationen gänzlich verzichtet. Die einzige Datenstruktur sind Felder, Standarddatentyp sind ganze Zahlen. In Abschnitt 2.4.1 wird zunächst die Syntax der Sprache *ADL* in Backus-Naur-Form angegeben. Abschnitt 2.4.2 definiert die operationelle Semantik von *ADL* durch Übersetzung in *RX-RAM*-Maschinenoperationen. Schließlich wird in 2.4.4 mit dem Ziel der späteren Komplexitätsanalyse der vorzustellenden Algorithmen eine genaue Aussage über die Zeit- und Raumkomplexitäten von *ADL*-Konstrukten getroffen.

2.4.1 Syntax der Sprache ADL

Tabelle 5 gibt die Syntax der Sprache *ADL* in Backus-Naur-Form an. Dabei bedeutet $\{x\}$ beliebig oftmaliges (auch kein) Auftreten von x und $[x]$ ein oder kein x. Im Falle von Mehrdeutigkeiten werden Terminalsymbole unterstrichen dargestellt.

Beispiel 2.22 Das Beispiel könnte in *ADL* nun folgendermaßen aussehen:

Algorithmus 2.1 *Summe und Maximum*

$s_1 \leftarrow input()$ { Eingabe des ersten Summanden }
$s_2 \leftarrow input()$ { Eingabe des zweiten Summanden }
$sum \leftarrow s_1 + s_2$ { Bildung der Summe }
if $sum < 100$ **then** { Summe ist kleiner als 100 }
    $output(100)$
else { Summe ist größer als 100 }
    $output(sum)$
end if

Bemerkung. Strenggenommen sind tiefergestellte Indizes in der Definition von *ADL* nicht enthalten. s_1 ist als $s1$ zu interpretieren.

<table>
<tr><td><algorithm></td><td>::=</td><td>Algorithmus <number>.<number> <constant text>
<program></td></tr>
<tr><td><program></td><td>::=</td><td><statement></td></tr>
<tr><td><statement></td><td>::=</td><td><compound statement> | <labeled statement> |
<conditional statement> | <iteration statement> |
<jump statement> | <procedure call> |
<assignment statement> | <empty statement></td></tr>
<tr><td><empty statement></td><td>::=</td><td>ε</td></tr>
<tr><td><compound statement></td><td>::=</td><td><statement> <comment> {<statement>}</td></tr>
<tr><td><labeled statement></td><td>::=</td><td><identifier> : <statement></td></tr>
<tr><td><conditional statement></td><td>::=</td><td>if <expression> then <statement> { elsif <statement> }
[else <statement>] end if</td></tr>
<tr><td><iteration statement></td><td>::=</td><td><while statement> | <repeat statement> | <for statement></td></tr>
<tr><td><assignment statement></td><td>::=</td><td><variable> ← <expression></td></tr>
<tr><td><comment></td><td>::=</td><td>{ <constant text> }</td></tr>
<tr><td><while statement></td><td>::=</td><td>while <expression> do <statement> end while</td></tr>
<tr><td><repeat statement></td><td>::=</td><td>repeat <statement> until <expression></td></tr>
<tr><td><for statement></td><td>::=</td><td>for <assignment statement> to <expression>
do <statement> end for</td></tr>
<tr><td><jump statement></td><td>::=</td><td>goto <identifier> | continue | break | return <expression></td></tr>
<tr><td><expression></td><td>::=</td><td><unary expression> |
<expression> <relational operator> <unary expression></td></tr>
<tr><td><relational operator></td><td>::=</td><td><logical operator> | <comparison operator> |
<binary operator> | <arithmetical operator></td></tr>
<tr><td><logical operator></td><td>::=</td><td>and | or</td></tr>
<tr><td><comparison operator></td><td>::=</td><td>$=$ | $\neq$ | $<$ | $>$ | $\leq$ | $\geq$</td></tr>
<tr><td><binary operator></td><td>::=</td><td>$\wedge$ | $\tilde{\vee}$ | $\vee$ | $\ll$ | $\gg$</td></tr>
<tr><td><arithmetical operator></td><td>::=</td><td>$+$ | $-$ | $\times$ | $\div$ | %</td></tr>
<tr><td><unary expression></td><td>::=</td><td><unary operator> <primary expression> | <primary expression></td></tr>
<tr><td><unary operator></td><td>::=</td><td>$+$ | $-$ | not | $\neg$</td></tr>
<tr><td><primary expression></td><td>::=</td><td><variable> | <constant> | <function call> | (<expression>)</td></tr>
<tr><td><variable></td><td>::=</td><td><identifier> { [<expression>] }</td></tr>
<tr><td><identifier></td><td>::=</td><td><letter> {<letter> | <digit> | _ }</td></tr>
<tr><td><constant text></td><td>::=</td><td>{ <letter> | <digit> | ⊔ }</td></tr>
<tr><td><constant></td><td>::=</td><td><boolean constant> | <number></td></tr>
<tr><td><boolean constant></td><td>::=</td><td>true | false</td></tr>
<tr><td><number></td><td>::=</td><td>0 | <nonzero digit> {<digit>}</td></tr>
<tr><td><letter></td><td>::=</td><td>a | b | c | ... | z | A | B | C | ... | Z</td></tr>
<tr><td><digit></td><td>::=</td><td>0 | <nonzero digit></td></tr>
<tr><td><nonzero digit></td><td>::=</td><td>1 | 2 | 3 | ... | 9</td></tr>
<tr><td><procedure call></td><td>::=</td><td><procedure identifier> (<argument list>)</td></tr>
<tr><td><function call></td><td>::=</td><td><function identifier> (<argument list>)</td></tr>
<tr><td><procedure identifier></td><td>::=</td><td><builtin procedure> | <identifier></td></tr>
<tr><td><function identifier></td><td>::=</td><td><builtin function> | <identifier></td></tr>
<tr><td><builtin procedure></td><td>::=</td><td>output</td></tr>
<tr><td><builtin function></td><td>::=</td><td>input</td></tr>
<tr><td><argument list></td><td>::=</td><td><expression> {, <expression> }</td></tr>
</table>

Tabelle 5: Syntax der Sprache *ADL*

2.4.2 Operationelle Semantik der Sprache *ADL*

Die operationelle Semantik der Sprache *ADL* soll nun durch Übersetzung in *RX-RAM*-Operationen angegeben werden. Da das Hauptinteresse dabei der Bestimmung der Komplexitäten von Anweisungen gilt, wird sowohl die vollständige Klammerung von *ADL*-Ausdrücken als auch die Ersetzung mehrfacher Feldklammern analog Definition 2.23 als gegeben vorausgesetzt. Im folgenden bezeichne *Cmd* die Menge aller *Anweisungen* und *Expr* die Menge aller *Ausdrücke* über *ADL*.

Definition 2.23 Für eine Variable v eines *ADL*-Programms sei

$$st(v) = S_j$$

für ein $j \in \mathbb{N}$ diejenige Funktion, die den Variablen ihren Speicherplatz zuordnet. Dabei sei immer $S_j \in \mathbf{R}$. Ausnahme: Für Feldvariablen f gelte $st(f) = R_j$, aber $st(f[k]) = S_{c(R_j)+k}$, wobei $S_{c(R_j)+l} \notin \mathbf{R}\ \forall\ l \in \mathbb{N}$.

Bemerkung. Die Definition beinhaltet die Annahme, daß vor Ausführung einer Anweisung die einfachen Variablen in Register geladen wurden, während sich große Speicherbereiche wie Felder im normalen Speicher befinden. Diese Annahme entspricht natürlich zunächst nicht der Realität, ist aber so zu interpretieren, daß sich reale Berechnungen tatsächlich weitestgehend in Registern abspielen. In Kapitel 6 wird ein Verfahren vorgestellt, dessen Implementierung gerade mit der Anzahl tatsächlich verfügbarer Register auskommt, also eine praktische Obergrenze der hier getroffenen Annahme aufzeigt. Feldindizes können mit 0 beginnen.

Es werden nun Übersetzungsfunktionen definiert, die Ausdrücke und Anweisungen der Sprache *ADL* in *RX-RAM*-Operationen übersetzen. Die Auswertung erfolgt unter Verwendung eines Hilfsstapels. Zwischenergebnisse werden auf dem Stapel auf- und nach Verwendung wieder abgebaut. Dabei sei *sp* (der *Stackpointer*) eine Hilfsvariable mit Anfangswert 0 und $++sp$ bzw. $--sp$ sei zu verstehen als „erhöhe bzw. verringere *sp* um 1 und benutze den Wert von *sp* dann“ sowie $sp++$ bzw. $sp--$ als „benutze den Wert von *sp* und erhöhe bzw. verringere *sp* dann um 1“. Darüber hinaus seien *oc* und *ic* Zähler der bereits benutzten Eingabe- und Ausgabezellen mit Anfangswert 0.

Es folgt zunächst die Übersetzung der Ausdrücke.

Definition 2.24 Es seien E, E_1 und $E_2 \in Expr$ und $(binop, instr) \in \{(+, \texttt{add}), (-, \texttt{sub}), (\times, \texttt{mul}), (\div, \texttt{div}), (\%, \texttt{mod}), (\wedge, \texttt{and}), (\vee, \texttt{or}), (\tilde{\vee}, \texttt{xor}), (\ll, \texttt{shl}), (\gg, \texttt{shr}), (=, \texttt{eq}), (\neq, \texttt{neq}), (\geq, \texttt{geq}), (>, \texttt{gt}), (\leq, \texttt{leq}), (<, \texttt{lt})\}$.

Dann sei $expx : Expr \to \mathbf{O}^+$ definiert durch:

$expx[\![\ z\ ]\!]$	$:=$	$(\texttt{mov}, R_{++sp}, z, 0)$ falls z numerische Konstante
$expx[\![\ true\ ]\!]$	$:=$	$(\texttt{mov}, R_{++sp}, 1, 0)$
$expx[\![\ false\ ]\!]$	$:=$	$(\texttt{mov}, R_{++sp}, 0, 0)$
$expx[\![\ v\ ]\!]$	$:=$	$(\texttt{mov}, R_{++sp}, st(v), 0)$ falls v Variable
$expx[\![\ f[E]\ ]\!]$	$:=$	$expx[\![\ E\ ]\!]$ $(\texttt{add}, R_{sp}, st(f), R_{sp})$ $(\texttt{ld}, R_{sp}, S_{R_{sp}}, 0)$
$expx[\![$ **not** $E\ ]\!]$	$:=$	$expx[\![\ E\ ]\!]$ $(\texttt{eq}, R_{sp}, R_{sp}, 0)$
$expx[\![\ \neg E\ ]\!]$	$:=$	$expx[\![\ E\ ]\!]$ $(\texttt{not}, R_{sp}, R_{sp}, 0)$
$expx[\![\ -E\ ]\!]$	$:=$	$expx[\![\ E\ ]\!]$ $(\texttt{sub}, R_{sp}, 0, R_{sp})$
$expx[\![\ E_1$ **and** $E_2\ ]\!]$	$:=$	$expx[\![\ E_1\ ]\!]$ $(\texttt{jz}, R_0, l_1, R_{sp})$ $expx[\![\ E_2\ ]\!]$ $(\texttt{mov}, R_{--sp}, R_{sp}, 0)$ $l_1 :$
$expx[\![\ E_1$ **or** $E_2\ ]\!]$	$:=$	$expx[\![\ E_1\ ]\!]$ $(\texttt{jz}, R_0, l_1, R_{sp})$ $(\texttt{jmp}, R_0, l_2, 0)$ $l_1 : expx[\![\ E_2\ ]\!]$ $(\texttt{mov}, R_{--sp}, R_{sp}, 0)$ $l_2 :$
$expx[\![\ E_1\ binop\ E_2\ ]\!]$	$:=$	$expx[\![\ E_1\ ]\!]$ $expx[\![\ E_2\ ]\!]$ $(instr, R_{--sp}, R_{sp-1}, R_{sp})$
$expx[\![\ input()\ ]\!]$	$:=$	$(\texttt{get}, R_{++sp}, E_{++ic}, 0)$

Die Semantik von Funktions-/Prozeduraufrufen wird dabei mit der Semantik eines *ADL*-Programms gleichgesetzt, das die Funktion bzw. Prozedur implementiert.

Die folgende Funktion realisiert die Übersetzung von Anweisungen.

Definition 2.25 Es seien $C,\ C_1,\ C_2, C_3 \in Cmd,\ E, E_1, E_2 \in Expr,\ v$ eine Variable und l, l_1, l_2, l_3 (Hilfs-) Labels von RX-RAM-Operationen. Dann sei $cmdx : Cmd \to \mathbf{O}^+$ definiert durch die Tabellen 6 und 7.

$cmdx\llbracket\ v \leftarrow E\ \rrbracket$	:=		$expx\llbracket\ E\ \rrbracket$
			(mov,$st(v)$,R_{sp--},0)
$cmdx\llbracket\ f[E_1] \leftarrow E_2\ \rrbracket$	:=		$expx\llbracket\ E_1\ \rrbracket$
			(add,R_{sp},$st(f)$,R_{sp})
			$expx\llbracket\ E_2\ \rrbracket$
			(st,$S_{R_{--sp--}}$,R_{sp},0)
$cmdx\llbracket\ C_1\ C_2\ \rrbracket$	:=		$cmdx\llbracket\ C_1\ \rrbracket$
			$cmdx\llbracket\ C_2\ \rrbracket$
$cmdx\llbracket\ l : C\ \rrbracket$	:=	l :	$cmdx\llbracket\ C\ \rrbracket$
$cmdx\llbracket$ **goto** $l\ \rrbracket$	:=		(jmp,R_0,l,0)
$cmdx\llbracket$ **return** $E\ \rrbracket$	:=		$expx\llbracket\ E\ \rrbracket$
			(put,A_{++oc},R_{sp--},0)
			(end,0,0,0)
$cmdx\llbracket$ **if** E **then** C **end if** $\rrbracket$	:=		$expx\llbracket\ E\ \rrbracket$
			(jz,R_0,l,R_{sp--})
			$cmdx\llbracket\ C\ \rrbracket$
		l :	
$cmdx\llbracket$ **if** E **then** C_1 **else** C_2 **end if** $\rrbracket$	:=		$expx\llbracket\ E\ \rrbracket$
			(jz,R_0,l_1,R_{sp--})
			$cmdx\llbracket\ C_1\ \rrbracket$
			(jmp,R_0,l_2,0)
		l_1 :	$cmdx\llbracket\ C_2\ \rrbracket$
		l_2 :	
$cmdx\llbracket$ **if** E_1 **then** C_1	:=		$expx\llbracket\ E_1\ \rrbracket$
elsif E_2 **then** C_2 **else** C_3 **end if** $\rrbracket$			(jz,R_0,l_1,R_{sp--})
			$cmdx\llbracket\ C_1\ \rrbracket$
			(jmp,R_0,l_3,0)
		l_1 :	$expx\llbracket\ E_2\ \rrbracket$
			(jz,R_0,l_2,R_{sp--})
			$cmdx\llbracket\ C_2\ \rrbracket$
			(jmp,R_0,l_3,0)
		l_2 :	$cmdx\llbracket\ C_3\ \rrbracket$
		l_3 :	
$cmdx\llbracket\ output(E)\ \rrbracket$	:=		$expx\llbracket\ E\ \rrbracket$
			(put,A_{++oc},R_{sp--},0)

Tabelle 6: Einfache Anweisungen und Konditionale

$cmdx[\![$ **while** E **do** C **end while** $]\!]$	$:=$	$l_1:$	$expx[\![\ E\ ]\!]$
			$(\texttt{jz},R_0,l_2,R_{sp--})$
			$cmdx[\![\ C\ ]\!]$
			$(\texttt{jmp},R_0,l_1,0)$
		$l_2:$	
$cmdx[\![$ **repeat** C **until** E $]\!]$	$:=$	$l_1:$	$cmdx[\![\ C\ ]\!]$
			$expx[\![\ E\ ]\!]$
			$(\texttt{jz},R_0,l_2,R_{sp--})$
			$(\texttt{jmp},R_0,l_1,0)$
		$l_2:$	
$cmdx[\![$ **for** $v \leftarrow E_1$ **to** E_2 **do** C **end for** $]\!]$	$:=$		$expx[\![\ E_1\ ]\!]$
		$l_1:$	$(\texttt{mov},st(v),R_{sp--},0)$
			$expx[\![\ E_2\ ]\!]$
			$(\texttt{sub}, R_{sp}, R_{sp}, st(v))$
			$(\texttt{jlz}, R_0, l_2, R_{sp--})$
			$cmdx[\![\ C\ ]\!]$
			$(\texttt{add}, R_{++sp}, st(v), 1)$
			$(\texttt{jmp}, R_0, l_1, 0)$
		$l_2:$	
continue (in Schleife)			
$cmdx[\![$ **continue** $]\!]$	$:=$		$(\texttt{jmp},R_0,l_1,0)$
break (in Schleife)			
$cmdx[\![$ **break** $]\!]$	$:=$		$(\texttt{jmp},R_0,l_2,0)$

Tabelle 7: Schleifenkonstrukte

Für jeden zusätzlichen **elsif**-Teil sei dabei ein entsprechender Block einzufügen.

Bemerkung. Auf die Auflösung der Hilfslabels in RX-RAM-Programm-Indizes soll hier verzichtet werden. Sowohl diese Auflösung als auch Berechnungen der sp-Werte werden zur Übersetzungszeit ausgewertet und erzeugen daher keinen zusätzlichen Kostenanteil.

Definition 2.26 (Semantik von ADL-Programmen)
Die Semantik $\mathscr{S}_{ADL}$ eines Programms P über ADL mit Eingabedaten $E = (e_1, e_2, \ldots, e_\eta)$ sei definiert durch

$$\mathscr{S}_{ADL}\ [\![P]\!](E) = \mathscr{R}\ [\![cmdx^*\ [\![P\ ]\!]\ ]\!]((((1), \varepsilon, E, \varepsilon))$$

wobei $cmdx^*$ die um die Ersetzung der Hilfslabels in RX-RAM-Programmindizes ergänzte Übersetzungsfunktion sei.

Eine Beispielübersetzung des *ADL*-Programms zur Summen- und Maximumbildung soll nun die Funktionsweise der Übersetzungsfunktionen veranschaulichen und die Semantik des *ADL*-Programms bestimmt werden.

Beispiel 2.27
Es sei $st = \{(sum, R_{\rho-2}), (s1, R_{\rho-1}), (s2, R_\rho)\}$, wobei ρ gleich der Anzahl der Register sei. Die Werte der Hilfsregister oc, ic und sp sind der Übersichtlichkeit halber jeweils eingesetzt.

Es ist $cmdx\llbracket P \rrbracket$=

Anweisung	Zwischenschritte	*RX-RAM*-Operation p_i	i
	$expx\llbracket input() \rrbracket$	(`get`,R_{++0},E_{++0},0)	1
$cmdx\llbracket s1 \leftarrow input() \rrbracket$		(`mov`,$R_{\rho-1}$,R_{1--},0)	2
	$expx\llbracket input() \rrbracket$	(`get`,R_{++0},E_{++1},0)	3
$cmdx\llbracket s2 \leftarrow input() \rrbracket$		(`mov`,R_ρ,R_{1--},0)	4
	$expx\llbracket s1 \rrbracket$	(`mov`,R_{++0},$R_{\rho-1}$,0)	5
	$expx\llbracket s2 \rrbracket$	(`mov`,R_{++1},R_ρ,0)	6
	$expx\llbracket s1 + s2 \rrbracket$	(`add`,R_{--2},R_{2-1},R_2)	7
$cmdx\llbracket sum \leftarrow s1 + s2 \rrbracket$		(`mov`,$R_{\rho-2}$,R_{1--},0)	8
	$expx\llbracket sum \rrbracket$	(`mov`,R_{++0},$R_{\rho-2}$,0)	9
	$expx\llbracket 100 \rrbracket$	(`mov`,R_{++1},100,0)	10
	$expx\llbracket sum < 100 \rrbracket$	(`lt`,R_{--2},R_{2-1},R_2)	11
$cmdx\llbracket$**if** $sum < 100$		(`jz`,R_0,16,R_{1--})	12
then	$expx\llbracket 100 \rrbracket$	(`mov`,R_{++0},100,0)	13
$output(100)$		(`put`,A_{++0},R_{1--},0)	14
		(`jmp`,R_0,18,0)	15
else	$expr\llbracket sum \rrbracket$	(`mov`,R_{++0},$R_{\rho-2}$,0)	16
$output(sum)$		(`put`,A_{++0},R_{1--},0)	17
end if$\rrbracket$		(`end`,0,0,0)	18

Nach Einsetzen der Beispiel-Eingabewerte ergibt sich in A_1 der Wert 100. Damit ist $\mathscr{S}_{ADL}(P)((17,4)) = (100)$, was natürlich die früher ermittelte Semantik des *RX-RAM*-Maschinenprogramms bestätigen muß.

Bemerkung. Daß sich das Ergebnis der formalen Übersetzung des *ADL*-Programms von der direkten Implementierung als *RX-RAM*-Programm unterscheidet, ist nicht verwunderlich. Gewöhnlich muß ein Verfahren zur Optimierung nach- bzw. zugeschaltet werden, um effizienteren Code zu erzeugen. Darauf kann hier vor allem deshalb verzichtet werden, weil sich sämtliche an Operationen beteiligten (nicht-Feld-) Operanden bereits in Registern befinden und zumindest Zugriffskosten auf einfache Variablen damit praktisch entfallen. Eine umfangreiche Diskussion über Codegenerierung und -optimierung findet sich in z.B. in [Aho86].

Der folgende Nachtrag gibt ein weiteres Beispiel eines Programms der Sprache *ADL*:

Nachtrag zum Beweis von Satz 2.21.
(Übersetzung von Turingmaschinenprogrammen in *RX-RAM*-Programme)

Algorithmus 2.2 *Simulation einer Turingmaschine*

```
S[1] ← input()   { Eingabewort in 1. Band einlesen }
while S[2 × k + 1] ≠ 0 and S[2 × k + 1] ≠ 1 do  { Haltezustände }
  S[2 × k + 2] ← 0   { Initialisiere Sammelzelle }
  for i = 1 to k do  { Sammle Information der k Bänder }
    symbolI ← (1 ≪ S[k + i]) ∧ S[i]   { Hole aktuelles Bit aus i-tem Band }
    S[2 × k + 2] ← S[2 × k + 2] ∨ symbolI   { Setze es in Sammelzelle }
  end for
  S[2 × k + 2] ← S[2 × k + 2] ∨ (S[2 × k + 1] ≪ k)   { Füge Zustand an }
  i ← 1
  while S[2 × k + 2] ≠ D[i][1] do  { Finde Folgekonfiguration }
    i ← i + 1
  end while
  newConf ← D[i][2]
  for i = 1 to k do  { Realisiere neue Konfiguration auf den Bändern }
    action ← (newConf ∧ (7 ≪ 3 × (i − 1))) ≫ (3 × (i − 1))   { Aktion }
    if action = 0 then  { Schreibe 0 }
      if ((S[i] ∧ (1 ≪ S[k + i])) ≫ S[k + i]) = 1 then  { Falls Bit gesetzt ist, }
        S[i] ← S[i] ∨̃ (1 ≪ S[k + i])   { lösche es }
      end if
    elsif action = 1 then  { Schreibe 1 }
      S[i] ← S[i] ∨ (1 ≪ S[k + i])
    elsif action = 2 then  { Gehe nach links }
      S[k + i] ← S[k + i] − 1
    elsif action = 3 then  { Gehe nach rechts }
      S[k + i] ← S[k + i] + 1
    end if
  end for
  S[2 × k + 1] ← newConf ≫ 3 × k   { Setze neuen Zustand }
end while
output(S[2 × k + 1])  { Ausgabe des Ergebnisses }
```

Die k Bänder der Turingmaschine seien nun als erste k Elemente eines Feldes S der Länge $2k + 2$ jeweils in $S[1]$ bis $S[k]$ dargestellt. In $S[k + i]$ befinde sich für $1 \leq i \leq k$ die

Position des Schreib-/Lesekopfes des i-ten Bandes. Der Zustand der Turingmaschine sei in $S[2k+1]$. Das Feldelement $S[2k+2]$ enthalte den jeweils zu sammelnden Gesamtzustand der Turingmaschine. Die Überführungsfunktion befinde sich in einem weiteren Feld D der Länge $2^{k+1} \cdot |Z|$, wobei $|Z|$ die Mächtigkeit der Zustandsmenge der Turingmaschine sei. Immer genau ein Eintrag in $D[i][1]$ stimmt mit der gesammelten Information in $S[2k+2]$ überein, die Folgekonfiguration sei dann für jedes Band in $D[i][2]$ in folgender Form codiert zu finden:

Codierung	Aktion	Codierung	Aktion
111	Keine Aktion	000	Schreibe 0
001	Schreibe 1	010	Gehe nach links
011	Gehe nach rechts		

2.4.3 Vereinfachung des Modells

Es wird nun eine Einschränkung vorgenommen, die die Modellmaschine weiter der Realität annähert. Die Motivation dazu liegt vor allem in der Schwierigkeit begründet, genauere Aussagen über die Komplexitäten treffen zu können, wenn wie im allgemeinen der Fall, über die Größe der Operanden keine Aussage getroffen werden kann oder diese nur ungenau abzuschätzen ist. Die Inhalte einer Zelle der Modellmaschine seien von nun an nach oben beschränkt. Die Anzahl der Binärstellen sei für ein $b \in \mathbb{N}$ durch $B = 2^b$ eingeschränkt. Alle weiteren Aussagen bezüglich der Komplexität von Programmen werden in Abhängigkeit von B getroffen.

Bemerkung. Die Tatsache, daß B eine Zweierpotenz ist, wird dabei keine grundsätzliche Rolle spielen. Sie wird aber aus Gründen der leichten Übertragbarkeit auf reale Maschinen gefordert.

Es ist nun wesentlich einfacher möglich, verwendbare Komplexitätsaussagen zu treffen. Dazu sei im folgenden angenommen, daß für einen Operanden F immer $\lg(c(F)) = B$ gelte. Die Operation $\neg z$ sei im Falle $z = 0$ nun so definiert, daß $\neg 0 = (\underbrace{1, 1, 1, \ldots, 1}_{B})$.

2.4.4 Zeitkomplexität von *ADL*-Programmen

Es werden nun tabellarisch die zeitlichen Kosten von *ADL*-Ausdrücken und -Anweisungen angegeben. Sie ergeben sich durch Summation der Kosten derjenigen *RX-RAM*- Operationen, die die *ADL*-Anweisung aufbauen. Die Kostenfunktion sei dabei mit t_{ADL} bezeichnet, *binop* $\in \{+, -, \wedge, \vee, \tilde{\vee}, \ll, \gg, =, \neq, \geq, >, \leq, <, \mathbf{and}, \mathbf{or}\}$ und *unop* $\in \{-, \neg, \mathbf{not}\}$. Die mittlere Spalte gibt die Basiskosten des Konstruktes an, nicht hinzuberechnet sind hier Kosten, die durch Schleifenkörper entstehen. Die letzte Spalte zeigt die entstehenden Gesamtkosten. Die Kosten von Multiplikation, Division sind entsprechend mit $M(B)$ bzw. $D(B)$ angegeben. $expx' [\![E]\!]$bezeichne dabei die letzte *RX-RAM*-Operation in der Auswertung eines Ausdrucks E durch $expx$.

Ausdruck/Anweisung	Basiskosten	Summe
z (Konstante)	B	B
$true/false$	B	B
v (Variable)	B	B
$f[E]$	$3B$	$3B + t_{ADL}(E) + S(c(expx'[\![E]\!][2]))$
$unop\ E$	B	$B + t_{ADL}(E)$
$E_1\ binop\ E_2$	$2B$	$2B + t_{ADL}(E_1) + t_{ADL}(E_2)$
$E_1 \times E_2$	$M(B)$	$M(B) + t_{ADL}(E_1) + t_{ADL}(E_2)$
$E_1 \div E_2$	$D(B)$	$D(B) + t_{ADL}(E_1) + t_{ADL}(E_2)$
$E_1\ \%\ E_2$	$D(B)$	$D(B) + t_{ADL}(E_1) + t_{ADL}(E_2)$
$input()$	$2B$	$2B$
$v \leftarrow E$	B	$B + t_{ADL}(E)$
$f[E_1] \leftarrow E_2$	$3B$	$3B + t_{ADL}(E_1) + t_{ADL}(E_2)+$ $S(c(expx'[\![E_1]\!][2]))$
$f[E_1] \leftarrow f[E_1]\ op\ E_2$	$3B$	$3B + t_{ADL}(E_1) + t_{ADL}(f[E_1]\ op\ E_2)$
goto l	0	0
return E	$2B$	$2B + t_{ADL}(E)$
if E **then** C **end if**	B	$B + t_{ADL}(E)[+t_{ADL}(C)]$
if E **then** C_1 **else** C_2 **end if**	B	$B + t_{ADL}(E)$ $[+(t_{ADL}(C_1))$ oder $t_{ADL}(C_2)]$
if E_1 **then** C_1 **elsif** E_2 **then** C_2 **else** C_3 **end if**	B	$B + t_{ADL}(E_1)$ $[+(t_{ADL}(C_1))$ oder $(t_{ADL}(E_2) + B)$ $[+(t_{ADL}(C_2))$ oder $t_{ADL}(C_3)]]$
while E **do** C **end while**	B	$B + t_{ADL}(E)+$ $\sum_{E=true}(B + t_{ADL}(E) + t_{ADL}(C))$
repeat C **until** E	B	$B + t_{ADL}(E) + t_{ADL}(C)$ $\sum_{E=false}(B + t_{ADL}(E) + t_{ADL}(C))$
for $v \leftarrow E_1$ **to** E_2 **do** C **end for**	B	$B + t_{ADL}(E_1)+$ $\sum_{E_2-E_1>0}(t_{ADL}(C) + t_{ADL}(E_2) + 6B)$
$output(E)$	$2B$	$2B + t_{ADL}(E)$
continue	0	0
break	0	0

Tabelle 8: Zeitkomplexität von *ADL*-Konstrukten

Bemerkung. In der Definition der *RX-RAM* war den Zugriffskosten auf Speicher ein logarithmisches Maß zugeordnet, das hier durch die Beschränkung der Größe der Maschinenworte wie auch bei anderen Operationen mit einem Anteil von B bei den Operationen `ld` und `st` eingehen müßte. Tatsächlich verhalten sich die Speicherzugriffskosten bei realen Maschinen in etwa logarithmisch, allerdings nicht genau und auch nicht in jedem Fall. Um später exaktere Aussagen treffen zu können, wurde eine zusätzliche Funktion $S(n)$ eingeführt. Diese wird analog der Kosten für Multiplikation und Division später experimentell bestimmt werden. Speziell sei bemerkt, daß diese im Falle einer Operation der Form $f[E_1] \leftarrow f[E_1]$ *op* E_2 nur einmal berechnet wird. Zum Zeitpunkt der Erstellung dieser Arbeit liegt der Wert von b für reale Maschinen bei 5 bis 8.

Die Beschränkung der Maschinenworte zerstört natürlich die Berechnungsuniversalität der RX-RAM. Der durch die Maschine adressierbare Speicher ist auf 2^B-1 Zellen beschränkt. Die Speicherung der Position des Turingmaschinen-Schreib-/Lesekopfes in der Simulation ist nicht mehr möglich. Dies wäre leicht zu beheben, indem man nur eine einzige Zelle unbeschränkt beließe.

Beispiel 2.28 Die ADL-Zeitkomplexität des Beispiels 2.22 ergibt sich wie folgt:

(1) $2B$ für die Eingabe, $1B$ für die Zuweisung.
(2) $3B$
(3) Jeweils $1B$ für die Auswertung der s_i, $2B$ für Addition, $1B$ für Zuweisung.
(4) $1B$ für das **if**, $4B$ für die Bedingung.
(5) $2B$ für die Ausgabe, $1B$ für das Argument (falls $sum < 100$).
(6) —
(7) $2B$ für die Ausgabe, $1B$ für das Argument (falls $sum \geq 100$).
(8) —

Die Zeitkomplexität des Beispielprogramms beträgt damit $18B$ Bitoperationen.

Formal sei die Zeitkomplexität eines Programms über der Sprache ADL nun folgendermaßen festgelegt:

Definition 2.29 (Zeitkomplexität) Die *Zeitkomplexität* t_{ADL} eines Programms $P = C_1C_2\dots C_k$ mit Anweisungen C_i der Sprache ADL ist

$$t_{ADL}(P, n_1, n_2, \dots) = \sum_{i=1}^{k} t_{ADL}(C_i)$$

Dabei sind $n_1, n_2, \dots$ Eingaben des Programms P.

2.4.5 Raumkomplexität von ADL-Programmen

Es soll nun definiert werden, was unter der Raumkomplexität eines ADL-Programms zu verstehen ist:

Definition 2.30 (Raumkomplexität) Die *Raumkomplexität* s_{ADL} eines Programms $P = C_1 C_2 \dots C_k$ mit Anweisungen C_i der Sprache ADL ist

$$s_{ADL}(P, n_1, n_2, \dots) = \varrho + \sum_k \max\{j : \exists \text{ Feld } f_k \text{ und } f_k[j] \text{ kommt in } P \text{ vor}\}$$

Dabei ist ϱ die maximale Anzahl benötigter Register und $n_1, n_2, \dots$ sind Parameter des Programms P.

Bemerkung. Die Definition der Raumkomplexität basiert hier wieder wesentlich auf der Idealisierung, daß alle Rechnungen komplett innerhalb der endlichen Menge von Registern stattfinden. Bei der späteren Algorithmenanalyse wird meist nur kurz $t_{ADL}(n)$ (bzw. $s_{ADL}(n)$) statt $t_{ADL}(P, n)$ (bzw. $s_{ADL}(P, n)$) geschrieben, wenn es eindeutig ist, welches Programm gemeint ist.

Beispiel 2.31
Die ADL-Raumkomplexität des Beispiels 2.22 beträgt nach der Beispielübersetzung in 2.27 $5B$ Bits (3 für $R_\varrho, R_{\varrho-1}, R_{\varrho-2}$ sowie 2 für R_1 und R_2).

2.4.6 Diskussion der Sprache ADL

Die Beschreibungssprache ADL stellt syntaktisch eine gewisse Verknüpfung von Teilmengen der beiden realen Programmiersprachen C und $Pascal$ dar.

$Pascal$ ist eine sehr gute Lehrsprache und wird vielfach zu Ausbildungszwecken genutzt. C und ihr „objektorientiertes Pendant“ $C++$ hingegen spielen in heutigen Anwendungen in fast allen Bereichen eine entscheidende Rolle. Sämtliche hier vorzustellenden Algorithmen sind (mangels ADL-Compiler) in C implementiert, wobei keine Konstrukte verwendet werden, die nicht in ADL ausdrückbar sind. Sie werden anhand von ADL erläutert und analysiert.

Die in diesem Kapitel geschaffenen Komplexitätsmaße werden dabei zur Anwendung kommen und sowohl Aussagen zu Laufzeiten und Speicheraufwand ermöglichen als auch den Weg zu Optimierungen aufzeigen. Die noch offenen Funktionen $M(B)$, $D(B)$ und $S(n)$ werden im nächsten Abschnitt an zwei Beispielmaschinen gemessen.

Der Begriff des *Algorithmus* wird in dieser Arbeit von nun an mit ADL-Programmen identifiziert.

2.5 Messungen der Funktionen M, D und S auf realen Maschinen

Um die Komplexitäten von ADL-Programmen auf reale Maschinen übertragen zu können, müssen die in der Praxis entscheidenden Funktionen M, D und S bestimmt werden. Exemplarisch soll dies hier auf zwei Maschinen geschehen. Die Messungen fanden auf zwei Sun-Workstations unterschiedlicher Bauart statt: eine SPARCstation-4 mit einem MB86904-Prozessor und 64 MB Hauptspeicher sowie eine Ultra-1 Enterprise mit Ultra-SPARC-Prozessor und 384 MB Hauptspeicher, jeweils unter dem Betriebssystem Solaris 2.6. Die Tatsache, daß beide Maschinen vom selben Hersteller stammen, ist dabei – wie in den nächsten Abschnitten sichtbar wird – unwesentlich, da sich die gemessenen Funktionen bereits bedeutsam unterscheiden.

2.5.1 Messung der Kosten einer Addition

Zunächst werden die Kosten einer Addition gemessen. Dazu diente der folgende Assemblercode, der jeweils 10^8-mal ausgeführt wurde:

```
.LL1:

  add %o1, %i0, %i1 )
  ...                } × 98
  add %o1, %i0, %i1 )

  cmp %o1, %o0
  bleu,a .LL1
  add %o1, 1, %o1
```

Das Register `%o0` enthält dabei die Obergrenze 10^8, `%o1` die Laufvariable. Die Zahl 98 ergibt sich aus der Tatsache, daß zusätzlich noch ein `cmp` – was einer Subtraktion entspricht – und die Inkrementierung des Schleifenzählers (`add %o1, 1, %o1`) vorgenommen wird. Insgesamt werden also 100 Operationen berechnet. Die Ergebnisse sind in den Tabellen 9 und 10 aufgeführt.

2.5.2 Messung der Kosten einer Multiplikation

Für die Messung der Multiplikation wurde der folgende Assemblercode ebenfalls 10^8-mal ausgeführt:

```
.LL1:

  umul %o1, %i0, %i1 )
  ...                 } × 100
  umul %o1, %i0, %i1 )

  cmp %o1, %o0
  bleu,a .LL1
  add %o1, 1, %o1
```

Davon abgezogen wurden die Basiskosten des Schleifenkonstruktes, also

```
.LL1:

  cmp %o1, %o0
  bleu,a .LL1
  add %o1, 1, %o1
```

Die Ergebnisse der Messung sind wiederum in den Tabellen 9 und 10 zu finden.

2.5.3 Messungen der Kosten einer Division

Die Messung der Division erfolgte analog der Messung der Multiplikation, allerdings wurden nur 10 Zeilen der Form `udiv %o1, %i0, %i1` ausgeführt.

2.5.4 Meßergebnisse

Die Ergebnisse der Messungen in Nanosekunden pro 1 Operation:

Operation	SPARCstation-4	Ultra-1
Addition	92	60
Multiplikation	1222	300
Division	4200	2500

Tabelle 9: Kosten von Multiplikation und Division in Nanosekunden

Einer Addition wird nun gemäß der Festlegung der Kosten der Sprache ADL die Einheit $2B$ zugeordnet. Daraus ergibt sich für die anderen Operationen im Verhältnis zur Addition:

Operation	SPARCstation-4	Ultra-1
Addition	$2B$	$2B$
Multiplikation	$43B$	$10B$
Division	$91B$	$83B$

Tabelle 10: Kosten von Multiplikation und Division in ADL-Einheiten

2.5.5 Messungen der Speicherzugriffe

Speicherzugriffe können einen erheblichen Anteil an der Gesamtkomplexität haben. Entscheidend ist dabei die Zufälligkeit der Zugriffe auf verschiedene Zellen im Verhältnis zur Größe des beteiligten Speicherbereichs. Der Grund dafür ist die Existenz möglicherweise mehrstufiger, schneller Zwischenspeicher („*Cache*"s), die allerdings normalerweise im Verhältnis zum verfügbaren Gesamthauptspeicher sehr klein sind (z.B. 1%). Beim Zugriff auf Speicher werden ganze Datenblöcke in den Cache gelesen. Bei sequentiellem Zugriff auf die Speicherzellen ist dieses Einlesen nur selten notwendig, bei zufälligem Zugriff fast immer. Daher ergeben sich in den Messungen beider Szenarien erhebliche Unterschiede.

Sequentielle Zugriffe Die Messung sequentieller Zugriffe auf Speicher erfolgte mit folgender Assemblersequenz, wobei die Laufvariable von 0 bis $k \cdot 10^7 - 1$ ($k \in \{0.5, 1, 2\}$ (2 nur auf Ultra-1)) lief:

```
.LL1:

  sll %o1,2,%o0
  add %o1,1,%o1
  cmp %o1,%o2
  bleu .LL1
  ld [%i0+%o0],%i1
```

Davon wurde analog der Messungen zur Multiplikation bzw. Division die Zeit zur Durchführung der folgenden Zeilen abgezogen. Die Operation `nop` ersetzt die Addition `%i0+%o0`:

```
.LL1:

  sll %o1,2,%o0
  add %o1,1,%o1
  cmp %o1,%o2
  bleu .LL1
  nop
```

Das Ergebnis war bei beiden Maschinen (unabhängig von der Größe des Intervalls) die doppelte Zeit einer Addition, also $S_{seq}(n) = 4B$.

Zufällige Zugriffe Zufällige Zugriffe wurden mit folgender Sequenz gemessen:

```
.LL1:
        call random,0
        add %i0,1,%i0
        call .urem,0
        mov %i1,%o1
        sll %o0,2,%o0
        cmp %i0,%l2
        bleu .LL1
        ld [%l0+%o0],%l1
```

Die Pseudozufallsfunktion `random` besitzt dabei eine hinreichend lange Periode. Wieder wurden die Basiskosten der Sequenz abgezogen.

Die Instruktion `nop` ersetzt dabei wiederum die Addition `%l0+%o0`. Die Messungen erfolgten jeweils 10mal für Felder der Längen 1000, 5000, 10000, 25000, 50000, 75000, 100000, 150000, 200000, 300000, 400000, 500000, 750000, 1000000, 2000000, 3000000, 4000000 und 5000000. Die Meßergebnisse sind zunächst tabellarisch auf zwei Nachkommastellen gerundet, dann jeweils graphisch – normalisiert auf B ADL-Einheiten – angegeben:

Intervallgröße	SPARCstation-4	Ultra-1
1000	3.47	1.16
5000	19.91	2.05
10000	25.70	0.77
25000	29.45	5.61
50000	31.33	4.51
75000	41.74	5.79
100000	50.03	2.72
150000	58.79	34.96
200000	63.18	61.81
300000	69.54	89.12
400000	72.44	102.41
500000	74.81	111.13
750000	78.57	122.60
1000000	81.41	127.28
2000000	89.12	135.60
3000000	93.66	137.97
4000000	96.77	140.28
5000000	99.71	141.65

Tabelle 11: Kosten zufälliger Speicherzugriffe

Die folgenden Plots zeigen die jeweils gemessenen Speicherzugriffskosten sowie die approximierenden Funktionen

$$S_{rand}(n) = \begin{cases} 4 & n < 100000 \\ \frac{43}{110000} \cdot n - 35 & 100000 \leq n \leq 320000 \\ 60 + 10 \cdot \log(\frac{n}{1000} - 300) & n > 320000 \end{cases}$$

für Ultra-1 sowie

$$S_{rand}(n) = \begin{cases} 4 & n \leq 900 \\ 11 \cdot \log(\frac{n}{100}) - 20 & n > 900 \end{cases}$$

für SPARCstation-4.

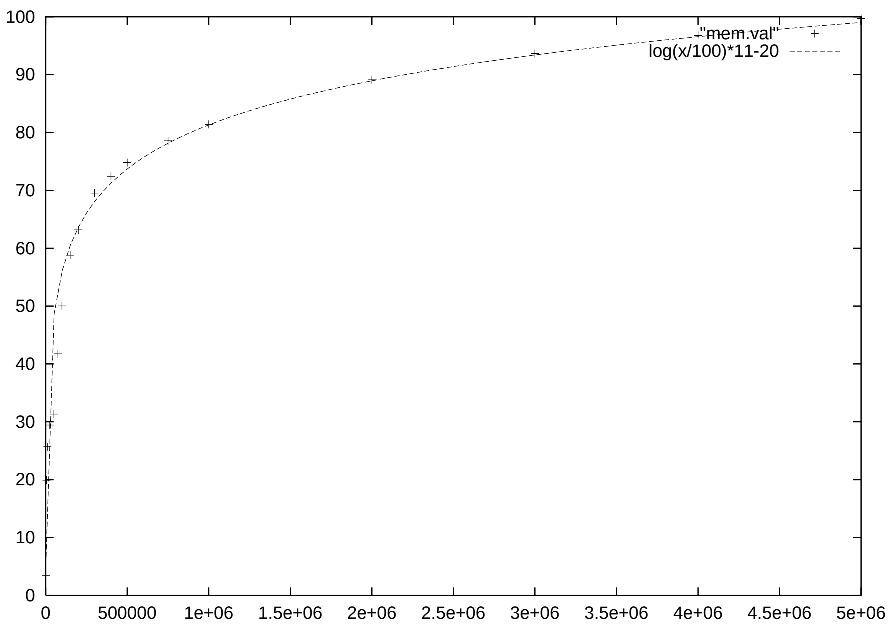


Abbildung 2: Kosten zufälliger Speicherzugriffe, SPARCstation-4

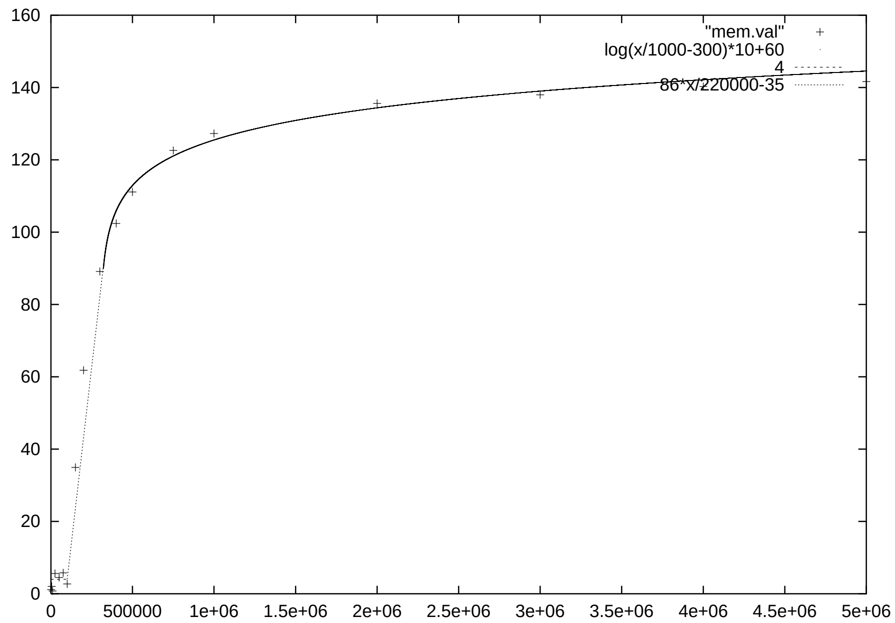


Abbildung 3: Kosten zufälliger Speicherzugriffe, Ultra-1

2.5.6 Diskussion der Meßergebnisse

Die Meßergebnisse an den zwei realen Beispielmaschinen zeigen zunächst, daß Multiplikationen und vor allem Divisionen vermieden oder ihre Anzahl möglichst minimiert werden sollten.

Eine teilweise noch entscheidendere Frage ist jedoch, wie zufällig die Zugriffe auf Speicher erfolgen. Die Zuordnung der Kosten von Speicherzugriffen kann nicht universell angegeben werden, sondern muß für jeden Algorithmus neu abgeschätzt werden. Eine Umordnung der Speicherzugriffe kann die Laufzeit um ein Vielfaches beeinflussen. Meist ist eine solche Umordnung nicht möglich und man ist darauf angewiesen, jeweils abzuschätzen, wieviele Zugriffe sequentiell bzw. zufällig erfolgen, um eine realitätsgetreue Aussage der Zeitkomplexität zu erhalten. Dies kann teilweise schwierig sein, wie schon an den ersten Algorithmen sichtbar wird, die im folgenden Kapitel vorgestellt werden. Die wahre Größenordnung der Funktion S liegt dabei immer zwischen S_{rand} und S_{seq}. Es sei aufgrund der unterschiedlichen Kosten von Speicherzugriffen im folgenden nur $S(n) = O(\log n)$ festgelegt. Im einzelnen wird S dann jeweils abgeschätzt.

3 Primzahlsiebe

3.1 Einführung

Primzahlsiebe sind Verfahren, die in einem gegebenen Intervall Primzahlen von zerlegbaren Zahlen trennen. Bereits um 200 v. Chr. erfand *Eratosthenes von Kyrene* ein erstes Sieb, das in seiner Beständigkeit als unübertroffen angesehen werden muß. Es gibt wohl kaum ein anderes Verfahren, das über zwei Jahrtausende als Maßstab galt und immer noch gilt.

Das Sieb des Eratosthenes und seine segmentierten Varianten waren lange Zeit mit einer asymptotischen Laufzeit von $O(n \log \log n)$ die schnellsten Verfahren zur Generierung von Primzahlen (dabei werden Speicherzugriffe nicht wie im hier verwendeten Modell mit einem Kostenbeitrag von $O(\log n)$ berechnet). Praktisch hat sich daran nichts geändert. Allerdings sind einige Verfahren bekannt, die zumindest asymptotisch schneller sind.

In [Mai77] findet sich erstmals ein Sieb, das nur lineare Zeit benötigt. Auch [Gri78] läuft in linearer Zeit. Ein theoretischer Durchbruch gelang *Pritchard* im Jahre 1981 [Pri81] mit einem Siebverfahren, das nur $O(\frac{n}{\log \log n})$ Zeit benötigt. Das Problem dieser Verfahren ist der Platzbedarf, der die Siebe nur für recht kleine Bereiche anwendbar werden läßt. Der sublineare Algorithmus von Pritchard benötigt $O(\frac{n}{\log \log n})$ Bits Speicher, was schnell reale Grenzen überschreitet. Die heute besten segmentierten Versionen benötigen $O(n)$ Bitoperationen und $O(\frac{\sqrt{n}}{(\log \log n)^2})$ Bits Speicherplatz (siehe z.B. [Sor98]). Allerdings besitzen auch diese Siebverfahren Nachteile, auf die in Abschnitt 3.5 eingegangen wird.

In diesem Kapitel werden ausgehend vom Sieb des Eratosthenes Schritt für Schritt Verbesserungen vorgenommen. Sämtliche Verfahren werden anhand von Programmen der Sprache *ADL* beschrieben. Reale Laufzeiten von Implementierungen werden denen sich aus den Zeitkomplexitäten von *ADL*-Konstrukten ergebenden Werten gegenübergestellt.

Die Analyse der Zwischenresultate führt schließlich zu einem hochoptimierten Verfahren, das in späteren Kapiteln zur Anwendung kommt. Eine portable Softwareimplementierung des resultierenden Algorithmus wird im Abschnitt 3.4 vorgestellt, die Laufzeiten untersucht und optimiert.

Abschließend wird ein Vergleich mit einem weiteren Verfahren stattfinden, das zwar asymptotisch bessere Laufzeiten aufweist, praktisch aber den Ergebnissen aus 3.4 unterliegt.

3.1.1 Probedivision

Das erste und zunächst naheliegendste Verfahren ist die Probedivision. Dabei sei *sieve* ein Feld der Länge n, wobei zu Beginn alle Elemente auf 0 gesetzt seien.

Algorithmus 3.1 *Probedivision*

1: $n \leftarrow input()$
2: $sieve[1] \leftarrow 1$ { Streiche 1 }
3: **for** $i \leftarrow 2$ **to** n **do** { Durchlaufe das Intervall }
4:    $sqrti \leftarrow sqrt(i)$ { Berechne Wurzel }
5:    **for** $j \leftarrow 2$ **to** $sqrti$ **do** { Teste Kandidaten durch Probedivision }
6:       **if** $(i \ \% \ j) = 0$ **then** { Teiler gefunden }
7:          $sieve[i] \leftarrow 1$ { Dies im Feld merken ... }
8:       **end if**
9:    **end for**
10: **end for**
11: **return** *sieve*

Satz 3.1 Algorithmus 3.1 siebt das Intervall $[1, n]$ in $t_{ADL}(n) > \frac{2}{3} n^{\frac{3}{2}} B$ Bitoperationen und $s_{ADL}(n) = B \cdot n + O(1)$ Raum.

Beweis. Da jede zerlegbare Zahl mindestens einen Teiler kleiner oder gleich ihrer Wurzel besitzt, wird dieser in Zeile 4 gefunden werden und somit im Feld *sieve* vermerkt. Es ist nun offensichtlich, daß nach dem Lauf von Algorithmus 3.1 $sieve[i]$ genau dann eine 0 enthält, wenn i eine Primzahl ist. Die Raumkomplexität ergibt sich im wesentlichen aus dem Feld *sieve* der Länge n, die Wurzel kann in konstantem Raum berechnet werden. Zur Zeitkomplexität: Es werden jeweils $\sqrt{i}$ Schritte durchlaufen. Dies führt mit

$$\sum_{i=1}^{n} \sqrt{i} > \int_{0}^{n} \sqrt{t} \, dt = \frac{2}{3} \cdot n^{\frac{3}{2}}$$

zur Behauptung. □

Die Abschätzung ist natürlich relativ grob. Es gibt einige Möglichkeiten, Algorithmus 3.1 zu verbessern (siehe z.B. [Adl83]), darauf soll aber verzichtet werden, da nun ein prinzipiell besseres Verfahren zur Anwendung kommt.

3.2 Das Sieb des Eratosthenes

Das Sieb des Eratosthenes von Kyrene (276–194 v. Chr.) wurde etwa im Jahre 100 von *Nicomachus* beschrieben (siehe [Doo26]). Eine erste Computerimplementierung ist in [Woo61] zu finden. Es folgt nun eine Beschreibung des Grundprinzips.

3.2.1 Basisverfahren

Algorithmus 3.2 *erat1*

```
n ← input()   { Eingabe der Obergrenze }
sqrtn ← sqrt(n)   { Berechne Wurzel }
sieve[1] ← 1   { Streiche 1 }
for p ← 2 to sqrtn do  { Durchlaufe das Intervall }
  if sieve[p] = 0 then  { Falls p prim }
    ndivp ← n ÷ p   { Bestimme Anzahl der Vielfachen }
    for j ← p to ndivp do  { Durchlaufe alle Vielfachen von p }
      sieve[p × j] ← 1   { Zerlegbare Zahl markieren }
    end for
  end if
end for
return sieve
```

Satz 3.2 Algorithmus 3.2 siebt die Primzahlen aus dem Intervall $[1, n]$ in

$$\begin{aligned} t_{ADL}(n) &= (15B + S(\sqrt{n}/2)) \cdot \sqrt{n} + (5B + D(B)) \cdot \pi(\sqrt{n}) + \\ &\quad +(13B + M(B) + S(n/2)) \cdot N(2, \sqrt{n}, n) + O(1) \\ &= (13B + M(B) + S(n/2)) \cdot N(2, \sqrt{n}, n) + O(\sqrt{n} \log n) \\ &= O(n \log n \log \log n) \end{aligned}$$

Bitoperationen und $s_{ADL}(n) = B \cdot n + O(1)$ Bits Speicherbedarf aus. Dabei ist

$$N(j, k, m) = \sum_{\substack{j \le p \le k \\ p \text{ prim}}} \left(\left\lfloor \frac{m}{p} \right\rfloor - p + 1 \right)$$

Beweis. p durchläuft alle Primzahlen bis zur Wurzel von n: Beginnend mit 2 werden alle Vielfachen aus dem Feld gestrichen. Sobald p auf ein neues Feldelement gesetzt wird, wird getestet, ob dies schon als zerlegbar markiert wurde. Falls ja, sind auch schon alle Vielfachen dieser Zahl markiert und p wird auf das nächste Element gesetzt. Erst wenn $sieve[p] = 0$ gilt, werden in der inneren Schleife alle Vielfachen dieser (Prim-) Zahl beginnend mit p^2 markiert. Alle möglichen kleinsten Teiler aller Zahlen kleiner gleich n werden von p durchlaufen, da $2 \le p \le \sqrt{n}$ gilt.

Der Platzbedarf von 3.2 wird wiederum durch das Feld *sieve* bestimmt. Zur Zeitkomplexität: Der **if**-Block 5–10 wird $\pi(\sqrt{n})$ mal durchlaufen. Für jedes prime $p \in \{2, \dots, \sqrt{n}\}$ sind $\lfloor \frac{n}{p} \rfloor - (p-1)$ Multiplikationen und Markierungen nötig, insgesamt also $N(2, \sqrt{n}, n)$.

Exemplarisch soll nun einmal im einzelnen gezeigt werden, wie sich die Gesamtkomplexität ergibt, an späteren Stellen wird nicht jede einzelne Zeile erklärt:

Zeile 1 erfordert $3B$ Bitoperationen. Die Berechnung der Wurzel benötigt quadratische Zeit in der Länge des Maschinenwortes und konstanten Platz (siehe z.B. [Aho74] für eine *RAM*), also sind für Zeile 2 $O(B + B^2)$ Operationen nötig. Zeile 3 trägt mit $5B + S(1)$ Schritten, Zeile 4 (inklusive dem **if** ohne Rumpf) mit $(15B + S(\sqrt{n}/2)) \cdot (\sqrt{n} - 1)$ Einheiten bei. Zeile 6 wird $\pi(\sqrt{n})$ mal ausgeführt und bringt daher $(3B + D(B)) \cdot \pi(\sqrt{n})$ Schritte, Zeile 7 zunächst $2B \cdot \pi(\sqrt{n})$, hinzu kommen $(13B + M(B) + S(n/2)) \cdot N(2, \sqrt{n}, n)$ aus Multiplikation, Zuweisung und Test des Schleifenkopfes. Insgesamt benötigt Algorithmus 3.2 also $(15B + S(\sqrt{n}/2)) \cdot \sqrt{n} + (5B + D(B)) \cdot \pi(\sqrt{n}) + (13B + M(B) + S(n/2)) \cdot N(2, \sqrt{n}, n) + O(1)$ Bitoperationen.

Wegen A.1, A.4 und A.6 ist

$$\begin{aligned}
N(2, \sqrt{n}, n) &= \sum_{p \le \sqrt{n}} \left\lfloor \frac{n}{p} \right\rfloor - \sum_{p \le \sqrt{n}} p + \sum_{p \le \sqrt{n}} 1 \\
&\le \sum_{p \le \sqrt{n}} \frac{n}{p} - \frac{n}{2 \log n} + \pi(\sqrt{n}) \\
&\le n \log \log n + O(1) - \frac{n}{2 \log n} + \frac{2\sqrt{n}}{\log n} \\
&= O(n \log \log n).
\end{aligned} \tag{1}$$

Mit $M(B) = O(B)$ (und $D(B) = O(B)$) gehen auch die restlichen Terme in $O(n \log \log n)$ auf. Wegen $S(n) = O(\log n)$ ist ein zusätzlicher Faktor $\log n$ anzufügen. Das Argument $n/2$ ergibt sich aus der Tatsache, daß der durchschnittliche Index eines Zugriffs auf das Feld etwa $n/2$ ist. □

Die anderen Terme $k_1 \cdot \sqrt{n}$ und $k_2 \cdot \pi(\sqrt{n})$ spielen selbst für kleine n keine wesentliche Rolle. Es muß also versucht werden, am Faktor von $N(2, \sqrt{n}, n)$ bzw. an $N(2, \sqrt{n}, n)$ selbst zu optimieren. Dazu wird zunächst $M(B)$ eliminiert.

3.2.2 Vermeidung von Multiplikationen

Um die Multiplikationen in Zeile 8 von Algorithmus 3.2 zu vermeiden, muß nur eine Variable beginnend mit p^2 sukzessive um p erhöht werden:

Algorithmus 3.3 *erat2*

```
1:  n ← input()   { Eingabe der Obergrenze }
2:  sqrt_n ← sqrt(n)   { Berechne Wurzel }
3:  sieve[1] ← 1   { Streiche 1 }
4:  for p ← 2 to sqrt_n do  { Durchlaufe Intervall }
5:    if sieve[p] = 0 then  { Falls p prim }
6:      ndivp ← n ÷ p   { Bestimme Anzahl der Vielfachen }
7:      next ← p × p   { Initialisiere Laufvariable }
8:      for j ← p to ndivp do  { Durchlaufe Vielfache von p }
9:        sieve[next] ← 1   { Zerlegbare Zahl markieren }
10:       next ← next + p   { Nächstes Vielfaches }
11:     end for
12:   end if
13: end for
```

Satz 3.3 Algorithmus 3.3 siebt die Primzahlen aus dem Intervall $[1, n]$ in

$$\begin{aligned} t_{ADL}(n) &= (15B + S(\sqrt{n}/2)) \cdot \sqrt{n} + (8B + D(B) + M(B)) \cdot \pi(\sqrt{n}) + \\ &\quad +(17B + S(n/2)) \cdot N(2, \sqrt{n}, n) + O(1) \\ &= (17B + S(n/2)) \cdot N(2, \sqrt{n}, n) + O(\sqrt{n} \log n) \\ &= O(n \log n \log\log n) \end{aligned}$$

Bitoperationen und $s_{ADL}(n) = B \cdot n + O(1)$ Bits Raum aus.

Beweis. Die Korrektheit folgt direkt, da lediglich die Multiplikation durch sukzessive Addition ersetzt wurde. An der Raumkomplexität hat sich nichts geändert. Zur Zeitkomplexität: In Zeile 7 kommen $M(B) + 3B$ Schritte hinzu, der **for**-Rumpf erzeugt einen Beitrag von $17B + S(n/2)$. An der asymptotischen Laufzeit hat sich nichts geändert. □

Der Unterschied beträgt bei z.B. $M(B) = 10B$ immerhin $6B \cdot N(2, \sqrt{n}, n)$ Schritte. Bevor die Auswirkungen in einer graphischen Übersicht dargestellt werden, folgt zunächst noch eine weitere Verbesserung am Faktor von $N(2, \sqrt{n}, n)$. Dabei wird die Division in Zeile 6 durch Ersetzung der **for**-Schleife in einen **while**-Konstrukt vermieden. Der Unterschied ist gering, dient aber auch der Vorbereitung zu weiteren Optimierungen.

3.2.3 Vermeidung der Division/Ersetzung der for-Schleife

Algorithmus 3.4 *erat3*

```
1: n ← input()   { Eingabe der Obergrenze }
2: sqrtn ← sqrt(n)   { Berechne Wurzel }
3: sieve[1] ← 1   { Streiche 1 }
4: for p ← 2 to sqrtn do   { Durchlaufe Intervall }
5:   if sieve[p] = 0 then   { Falls p prim }
6:     next ← p × p   { Initialisiere Laufvariable }
7:     while next ≤ n do   { Durchlaufe Vielfache von p }
8:       sieve[next] ← 1   { Zerlegbare Zahl markieren }
9:       next ← next + p   { Nächstes Vielfaches }
10:    end while
11:  end if
12: end for
13: return sieve
```

Satz 3.4 Algorithmus 3.4 siebt die Primzahlen aus dem Intervall $[1, n]$ in der Zeit

$$\begin{aligned} t_{ADL}(n) &= (14B + S(\sqrt{n}/2)) \cdot \sqrt{n} + (M(B) + 5B) \cdot \pi(\sqrt{n}) + \\ &\quad + (15B + S(n/2)) \cdot N(2, \sqrt{n}, n) + O(1) \\ &= (15B + S(n/2)) \cdot N(2, \sqrt{n}, n) + O(\sqrt{n} \log n) \\ &= O(n \log n \log\log n) \end{aligned}$$

unter Verwendung von $s_{ADL}(n) = B \cdot n + O(1)$ Bits Speicher aus.

Beweis. An der Korrektheit hat sich wiederum nichts geändert, die Ermittlung der festen Anzahl der Schritte pro p wird nun vermieden, die Variable *next* durchläuft immer noch sämtliche Vielfache der p bis n. Die Zeitkomplexität nimmt weiter ab: Die Beiträge der Zeilen 1, 2 und 3 bleiben, Zeilen 6 und 7 kosten zunächst $(M(B) + 5B) \cdot \pi(\sqrt{n})$. Der **while**-Rumpf wird wiederum $N(2, \sqrt{n}, n)$ mal durchlaufen, die Kosten reduzieren sich hier auf $(15B + S(n/2)) \cdot N(2, \sqrt{n}, n)$. Insgesamt ergibt sich also die angegebene Zeit. □

Eine Bemerkung zur Ausgabe: Eigentlich müßte man natürlich jeweils das gesamte Feld *sieve* durchlaufen und ausgeben, was mit $(12B + S(n/2)) \cdot n$ einen erheblichen Anteil ausmacht. Tatsächlich kostet eine (Bildschirm- oder Datei-) Ausgabe in der Praxis mehr Zeit als die gesamte Rechnung (wobei der Faktor vor n bei solchen Ausgaben noch wesentlich höher anzusetzen ist). Da jedoch bei Anwendungen der Verfahren die Felder selbst nicht ausgegeben werden, sondern im Speicher verbleiben, wurde hier auf die komplette Ausgabe verzichtet und nur ein Zeiger zurückgeliefert. Die Bemerkung, daß Ausgaben sehr teuer sind, läßt sich dahingehend erweitern, daß es sich auch nicht lohnt, gesiebte Intervalle zur späteren Verwendung auf Festplatten zu speichern. Auch Eingaben aus Dateien sind in der Praxis ähnlich langsam und kosten wiederum mehr Zeit als der Siebvorgang an sich.

3.2.4 Speicherreduzierung

Es soll nun eine erste Verbesserung der Raumkomplexität vorgenommen werden. Bisher wurde zur Speicherung des zu siebenden Feldes *sieve* pro Zahl ein Maschinenwort benutzt. Notwendig ist nur die Information, ob die entsprechende Zahl prim ist oder nicht, woraus folgt, daß eigentlich ein Bit reicht. Algorithmus 3.5 wird dieser Tatsache gerecht. Dabei bezeichnet B wie bisher die Maschinenwortlänge und $b = \log_2 B$. Um unnötige Umrechnungen zu sparen, wird nun auch das Element $sieve[0]$ verwendet.

Algorithmus 3.5 *erat4a*

```
1: n ← input()   { Eingabe der Obergrenze }
2: sqrtn ← sqrt(n)   { Berechne Wurzel }
3: sieve[0] ← 1   { Streiche 1 }
4: for p ← 2 to sqrtn do   { Durchlaufe Intervall }
5:   if (sieve[p ÷ B] ∧ (1 ≪ (p % B))) = 0 then   { Falls p prim }
6:     next ← p × p
7:     while next ≤ n do   { Durchlaufe Vielfache von p }
8:       sieve[next ÷ B] ← sieve[next ÷ B] ∨ (1 ≪ (next % B))   { Markieren }
9:       next ← next + p   { Nächstes Vielfaches }
10:    end while
11:  end if
12: end for
13: return sieve
```

Auf den ersten Blick scheint die Verminderung des Speichers zeitlich unbezahlbar, da drei Divisionsoperationen auftauchen. Nun läßt sich aber ausnutzen, daß es sich bei B um eine Zweierpotenz handelt und Divisionen von Zweierpotenzen durch einfache Shiftoperationen realisierbar sind. Dazu sei die folgende Zeile durch $setBit1(sieve, next)$ abgekürzt:

$$sieve[next \gg b] \leftarrow sieve[next \gg b] \vee (1 \ll (next \wedge (B-1)))$$

Darüber hinaus stehe $getBit1(sieve, p)$ nun für:

$$(sieve[p \gg b] \wedge (1 \ll (p \wedge (B-1)))$$

Die Zeile 8 wird nun durch $setBit1(sieve, next)$ und die Bedingung in Zeile 4 durch $getBit1(sieve, p) = 0$ ersetzt. Dabei kann und sollte $B-1$ vorberechnet sein, um eine weitere Operation zu sparen.

Algorithmus 3.6 *erat4b*

```
1: n ← input()   { Eingabe der Obergrenze }
2: sqrtn ← sqrt(n)   { Berechne Wurzel }
3: sieve[0] ← 3   { Streiche 0 und 1 }
4: for p ← 2 to sqrtn do   { Durchlaufe Intervall }
5:   if getBit1(sieve, p) = 0 then   { Falls p prim }
6:     next ← p × p
7:     while next ≤ n do   { Durchlaufe Vielfache von p }
8:       setBit1(sieve, next)
9:       next ← next + p   { Nächstes Vielfaches }
10:    end while
11:  end if
12: end for
13: return sieve
```

Satz 3.5 Algorithmus 3.6 siebt die Primzahlen aus dem Intervall $[1, n]$ in der Zeit

$$\begin{aligned} t_{ADL}(n) &= (34B + S(\sqrt{n}/2B)) \cdot \sqrt{n} + (8B + M(B)) \cdot \pi(\sqrt{n}) + \\ &\quad +(33B + S(n/2B)) \cdot N(2, \sqrt{n}, n) \\ &= (33B + S(n/2B)) \cdot N(2, \sqrt{n}, n) + O(\sqrt{n} \log n) \\ &= O(n \log n \log \log n) \end{aligned}$$

unter Verwendung von $s_{ADL}(n) = n + O(1)$ Bits Speicher aus.

Beweis. Unterschiede zu Algorithmus 3.4 bestehen in den Zeilen 3, 5 und 8. In Zeile 3 wird $sieve[0]$ auf (binär) 11 gesetzt, was dem Streichen der Zahlen 0 und 1 entspricht. In Zeile 5 muß gelten:

$$getBit1(sieve, p) = 0 \Longleftrightarrow \text{Bit } p \text{ des Feldes } sieve = 0$$

Bit p befindet sich im $\lfloor \frac{p}{B} \rfloor$-ten Wort des Feldes *sieve* an Position $p \mod B$. Die ganzzahlige Division ist durch den Rechtsshift $p \gg b$ mit $b = \log_2 B$ realisiert, die Berechnung des Rests nach Division durch B mithilfe des bitweisen Und, was bei $B = 2^b$ genau der mod-Operation entspricht. Schließlich wird der Wert des $p \mod B$-ten Bits des $\lfloor \frac{p}{B} \rfloor$-ten Wortes durch die bitweise Und-Verknüpfung mit $2^{p \mod B}$ (entspricht $1 \ll (p \mod B)$) ermittelt. In Zeile 8 muß $setBit1(sieve, next)$ das *next*-te Bit des Feldes *sieve* auf 1 setzen. Dies geschieht analog *getBit1* – diesmal unter Verwendung des bitweisen Oder. □

Bemerkung. Obwohl *setBit1* und *getBit1* syntaktisch als Prozedur bzw. Funktion gelten, sind beide als Textersatz zu verstehen, der z.B. in der Programmiersprache *C* als Präprozessoranweisung beim Kompiliervorgang realisiert werden kann.

Es erfolgt nun eine weitere Halbierung des benötigten Speichers bei gleichzeitiger Verbesserung des Wertes von N.

3.2.5 Speicherung und Sieben ausschließlich ungerader Zahlen

Die Tatsache, daß außer der 2 keine weitere gerade Primzahl existiert, wird nun in das Sieb eingebaut. Sowohl das Sieben der 2 sowie aller Zweifachen der Primzahlen kleiner gleich $\sqrt{n}$ als auch die Speicherung sämtlicher gerader Zahlen wird dabei vermieden. Zum ähnlich schnellen Zugriff auf die einzelnen Bits sei nun $setBit2(sieve, next) =$

$sieve[next \gg (b+1)] \leftarrow sieve[next \gg (b+1)] \vee (1 \ll ((next \gg 1) \wedge (B-1)))$

sowie $getBit2(sieve, p) =$

$(sieve[p \gg (b+1)] \wedge (1 \ll ((p \gg 1) \wedge (B-1)))$

Dabei sei wiederum $b+1$ und $B-1$ vorberechnet.

Bemerkung. Zur Verdeutlichung der Bedeutung der Bitoperationen ein paar Beispiele:

Es sei dazu $B = 32$, also $b = 5$. Dann setzt $setBit2(sieve, 1)$ das nullte Bit im nullten Wort von *sieve* auf 1, $setBit2(sieve, 2)$ das erste, $setBit2(sieve, 3)$ ebenso das erste. $setBit2(sieve, 65)$ setzt das nullte Bit des ersten Wortes.

Allgemein setzt $setBit2(sieve, n)$ das $\lfloor \frac{n}{2} \rfloor \mod B$-te Bit des $\lfloor \frac{n}{2B} \rfloor$-ten Wortes. Analoges gilt für *getBit*.

Das folgende Verfahren speichert und siebt nur noch ungerade Zahlen:

Algorithmus 3.7 *erat5*

```
n ← input()   { Eingabe der Obergrenze }
sqrtn ← sqrt(n)   { Berechne Wurzel }
sieve[0] ← 1   { Streiche 1 }
p ← 3
while p ≤ sqrtn do   { Durchlaufe Intervall }
  if getBit2(sieve, p) = 0 then   { Falls p prim }
    twop ← p + p
    next ← p × p
    while next ≤ n do   { Durchlaufe Vielfache von p }
      setBit2(sieve, next)
      next ← next + twop   { Nächstes Vielfaches }
    end while
  end if
  p ← p + 2
end while
return sieve
```

Satz 3.6 Algorithmus 3.7 siebt die Primzahlen aus dem Intervall $[3, n]$ in der Zeit

$$\begin{aligned} t_{ADL}(n) &= (33B + S(\sqrt{n}/4B)) \cdot \sqrt{n}/2 + (13B + M(B)) \cdot \pi(\sqrt{n}) + \\ &\quad +(36B + S(n/4B)) \cdot N(3, \sqrt{n}, n)/2 \\ &= (36B + S(n/4B)) \cdot N(3, \sqrt{n}, n)/2 + O(\sqrt{n} \log n) \\ &= O(n \log n \log \log n) \end{aligned}$$

unter Verwendung von $s_{ADL}(n) = n + O(1)$ Bits Speicher aus.

Beweis. Analog Algorithmus 3.6. □

Bemerkung. Eine weitere Reduzierung der Raumkomplexität auf diese Weise (z.B. Beschränkung auf Zahlen $\pm 1 \mod 6$) lohnt sich nicht mehr, da dann eine echte Division sowie eine Multiplikation und eine Subtraktion zum Markieren notwendig werden.

Das Sieb des Eratosthenes hat in seiner bisher gezeigten Form noch eine Schwäche, die bisher noch nicht angesprochen wurde: Es finden Mehrfachmarkierungen statt, d.h. beim Durchlaufen der möglichen Primfaktoren aller Zahlen kleiner n wird bisher keine Rücksicht darauf genommen, ob gewisse Schritte überhaupt noch notwendig sind oder nicht. Darauf wird an späterer Stelle noch eingegangen.

Es soll nun eine experimentelle Laufzeitanalyse der vorgestellten Verfahren stattfinden.

3.2.6 Laufzeitanalyse

In diesem Abschnitt werden die ADL-Komplexitäten der Algorithmen 3.2 – 3.7 und 3.9 experimentell gemessenen Zeiten gegenübergestellt. Es wird dabei kein absoluter Vergleich etwa in CPU-Sekunden durchgeführt, sondern vielmehr das Verhältnis der Laufzeiten der Algorithmen untereinander verglichen.

Die Zufälligkeit der Speicherzugriffe eines Programms, die einen entscheidenden Einfluß auf die Kosten hat (siehe 2.5.6), kann durch den Abgleich zweier einzelner Meßwerte mit den entsprechenden ADL-Komplexitäten ermittelt werden. Im Falle der Siebalgorithmen ergab sich so ein Wert von $S(n) \approx \frac{1}{4} \cdot S_{rand}(n)$. Die sich so ergebende Funktion $S(n)$ und die gemessenen Funktionen $M(B)$ und $D(B)$ werden nun in die Ermittlung der ADL-Komplexitäten einfließen. Die Zeitmessungen erfolgten dabei aus den Programmen selbst heraus, da eine Messung von außen – etwa über das Kommando `time` – zu Unschärfen führte. Die ADL-Zeitkomplexitäten wurden durch Programme bestimmt, die die jeweils genauesten Terme der Aussagen der Sätze 3.2 – 3.8 berechnen. Alle realen Zeiten sind in Einheiten von 10 Millisekunden angegeben, alle ADL-Komplexitäten in B Bitoperationen. Betrachtet wurde jeweils das Intervall $[1,\ 5 \cdot 10^6]$ in Schritten der Länge 1000.

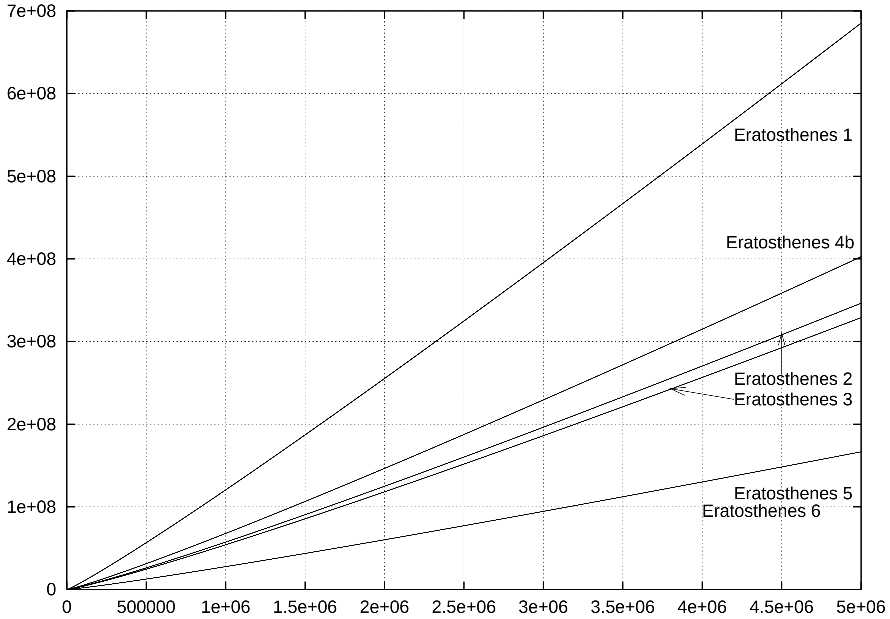


Abbildung 4: *ADL*-Zeitkomplexitäten (SPARCstation-4)

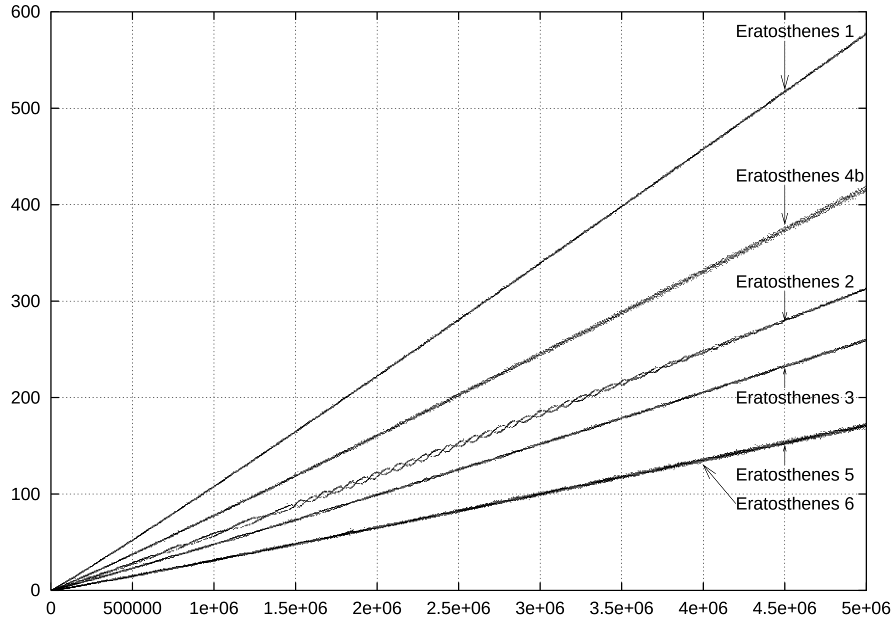


Abbildung 5: Reale Laufzeiten (SPARCstation-4)

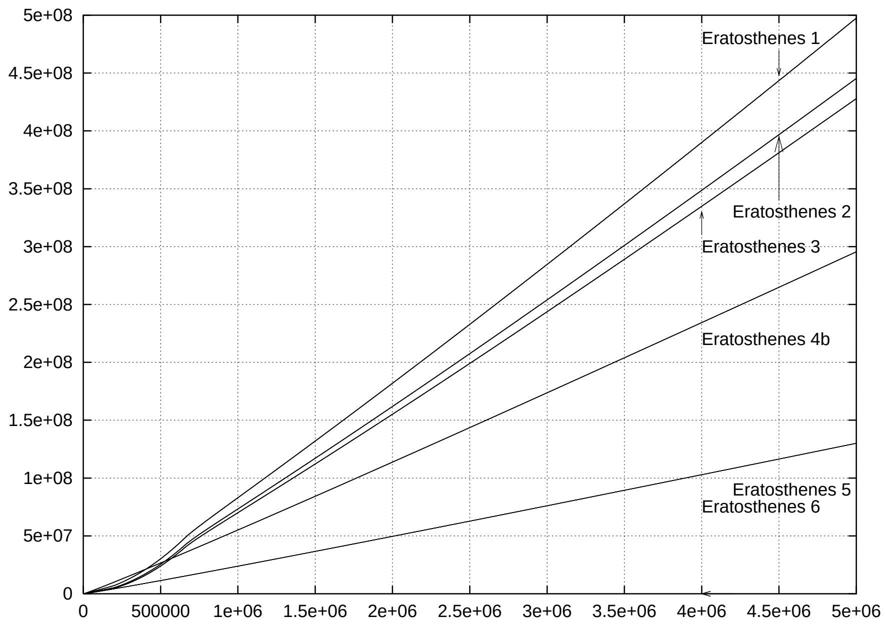


Abbildung 6: ADL-Zeitkomplexitäten (Ultra-1)

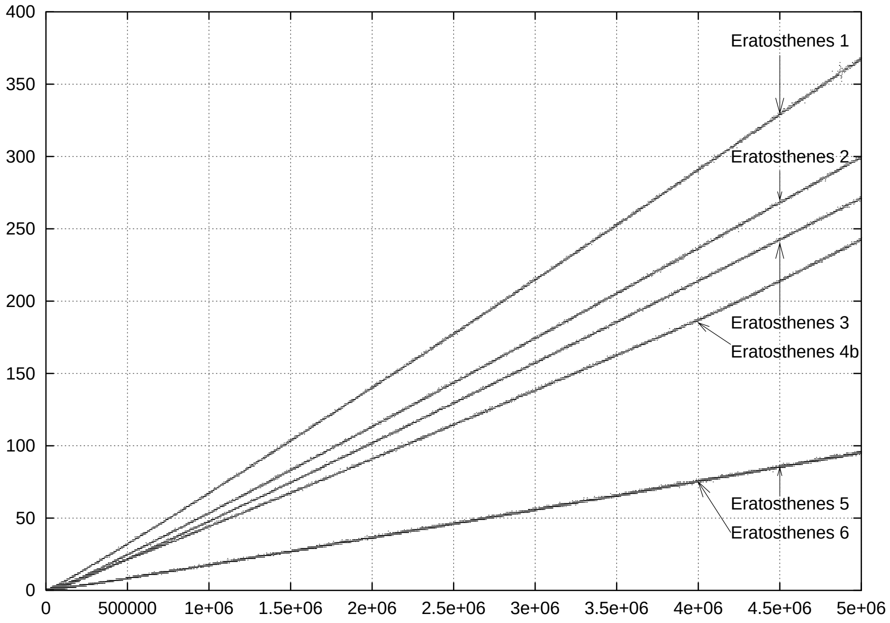


Abbildung 7: Reale Laufzeiten (Ultra-1)

Es wird erwartungsgemäß sichtbar, daß die Vermeidung der Multiplikationen in 3.2 bei den weiteren Verfahren eine entscheidende Verbesserung auf beiden Beispielmaschinen bedeutet. Die Ersetzung der **for**-Schleife durch einen **while**-Konstrukt ergibt nur einen kleinen Vorteil. Die erwartete Verbesserung von Algorithmus 3.6 durch Reduzierung des Speicheraufwandes wird auf der SPARCstation-4 durch den zusätzlichen Aufwand der Operationen zur Berechnung der Bitpositionen und Feldindizes zunichte gemacht. Erst die weitere Halbierung sowie die Unterlassung des Siebens mit der geraden Primzahl führt zu einer deutlichen Verbesserung. Anders verhält es sich bei der Ultra-1, wo bereits 3.6 einen – wenn auch relativ kleinen – Vorteil bedeutet.

Das Sieb in den bis hierher gezeigten Versionen weist einen großen praktischen Nachteil auf. Selbst nach der Reduzierung des Speicherbedarfs auf $n/2$ werden sehr schnell praktische Grenzen erreicht. Zum Zeitpunkt der Erstellung dieser Arbeit lag der durchschnittlich zur Verfügung stehende Hauptspeicher pro Maschine bei etwa 10^8 Bytes[1], d.h. es ergibt sich eine obere Grenze der siebbaren Intervalle in der Größenordnung von 10^9 – 10^{10}. Der folgende Abschnitt beschäftigt sich mit einer Lösung dieses Problems.

[1] diese Größe wurde in [Bay77] noch als „absurd“ bezeichnet

3.3 Segmentierung

Ein *segmentiertes Sieb* taucht in der Literatur erstmals in [Bre73] im Zusammenhang mit der Suche nach großen Primzahllücken auf. In [Bay77] wurde die Segmentierung erstmals speziell behandelt.

3.3.1 Prinzip

Das Prinzip der Segmentierung ist einfach. Zunächst erfolgt eine Aufteilung des zu siebenden Intervalls in normalerweise gleichgroße Teilstücke (*Segmente*). Jedes Segment wird nun für sich gesiebt. Bei k Teilen ergibt sich eine Reduzierung der Raumkomplexität um den Faktor $1/k$. Problematisch ist dabei allerdings zunächst, daß die Information über die ersten $\pi(\sqrt{n})$ Primzahlen über den gesamten Siebvorgang hinweg verfügbar sein muß.

3.3.2 Verwendung einer Primzahlliste bis $\sqrt{n}$

Es wird nun ein Verfahren vorgestellt, das zunächst eine Liste der ersten $\pi(\sqrt{n})$ Primzahlen generiert, mit denen gesiebt wird. Es ist dann insbesondere nicht mehr notwendig, das gesamte Feld *sieve* zu durchlaufen und nach 0-Werten bzw. korrespondierenden Primzahlen zu suchen, denn alle möglichen Primfaktoren sind vorberechnet worden. Die Vorberechnung des Feldes *gaps* wird in 3.8 mit Algorithmus 3.7 erfolgen. Die Primzahlen kleiner gleich $\sqrt{n}$ werden in *gaps* als Differenzen aufeinander folgender Primzahlen dargestellt. Dies ist eine sehr sparsame Methode, Primzahlen zu speichern, da bis 436273009 ($\approx \sqrt{2 \cdot 10^{17}}$) nur ein Byte zur Speicherung notwendig ist (siehe z.B. [Rie94]).

Bemerkung. Eigentlich müßte man nur ungerade Zahlen, also halbe Lücken speichern, dies erfordert dann jedoch wieder eine zusätzliche Rechenoperation bei der Umrechnung auf die Primzahlen selbst. Allerdings erhöht sich damit auch der Bereich, in dem nur 1 Byte notwendig ist, auf 304599508537 ($\approx \sqrt{9 \cdot 10^{22}}$).

Es folgt zunächst der Algorithmus zur Erzeugung des Feldes mit Differenzen aufeinander folgender Primzahlen kleiner gleich $\sqrt{n}$:

Algorithmus 3.8 *pdiff*

```
1: n ← input()
2: tmpsieve ← erat5(n)
3: index ← 0  { Index für gaps-Feld }
4: thisgap ← 1  { Nächste Lücke }
5: p ← 3  { Laufvariable für Feld tmpsieve }
6: while p ≤ n do  { Sammle Lücken aus tmpsieve in gaps }
7:   if getBit2(tmpsieve, p) = 0 then  { Falls p prim }
8:     gaps[index] ← thisgap  { Lücke merken }
9:     index ← index + 1  { Index erhöhen }
10:    thisgap ← 2  { Lücke zurücksetzen }
11:  else
12:    thisgap ← thisgap + 2  { Ansonsten Lücke erhöhen }
13:  end if
14:  p ← p + 2  { Laufvariable erhöhen }
15: end while
16: return gaps
```

Lemma 3.7 Es sei $\max\{d : d = p_i - p_{i-1}, i \leq \pi(n)\} < 2^B$. Dann erzeugt Algorithmus 3.8 die Liste *gaps* der Differenzen aufeinanderfolgender Primzahlen kleiner gleich n (beginnend mit $1 = 3 - 2$) in der Zeit

$$\begin{aligned} t_{ADL}(n) &= (36B + S(n/4B)) \cdot N(3, \sqrt{n}, n)/2 + (33B + S(n/4B)) \cdot n + \\ &\quad + (8B + S(\pi(n)/2)) \cdot \pi(n) \\ &= (36B + S(n/4B)) \cdot N(3, \sqrt{n}, n)/2 + O(n \log n) \\ &= O(n \log n \log \log n) \end{aligned}$$

unter Verwendung von $s_{ADL}(n) = n + \pi(n) \cdot B + O(1)$ Bits Speicher.

Beweis. Der wesentliche Beitrag zur Zeitkomplexität liegt im Sieben des Intervalls. Die Zeilen 8–10 werden $\pi(n)$-mal durchlaufen, Zeile 12 $n - \pi(n)$-mal. Sobald eine Primzahl gefunden wurde, wird der jeweilige Wert der Lücke im Feld *gaps* vermerkt. In Zeile 10 wird dieser dann auf 2 zurückgesetzt, ansonsten in Zeile 12 jeweils um 2 erhöht. □

Bemerkung. Die Voraussetzung von Satz 3.7, daß ein Maschinenwort zur Speicherung einer Lücke ausreicht, ist in der Realität keine Einschränkung. In Implementierungen reicht für heute erreichbare Obergrenzen ein Byte aus.

Algorithmus 3.9 *erat6*

1: $n \leftarrow input()$
2: $sqrtn \leftarrow sqrt(n)$ { Berechne Wurzel }
3: $gaps \leftarrow pdiff(sqrtn)$ { Erzeugen der Primzahllücken }
4: $setBit2(sieve, 1)$ { Markiere 1 als zerlegbar }
5: $index \leftarrow 1$ { Index für *gaps*-Feld }
6: $p \leftarrow 3$ { Initialisiere Laufvariable }
7: **while** $p \leq sqrtn$ **do** { Durchlaufe Primzahlen $\leq \sqrt{n}$ }
8: $\quad twop \leftarrow p + p$ { Verdopple p }
9: $\quad next \leftarrow p \times p$ { Setze $next$ auf p^2 }
10: $\quad$ **while** $next \leq n$ **do** { Markiere Vielfache von p }
11: $\quad\quad setBit2(sieve, next)$ { Markiere zerlegbare Zahl }
12: $\quad\quad next \leftarrow next + twop$ { Nächstes Vielfaches }
13: $\quad$ **end while**
14: $\quad index \leftarrow index + 1$ { Erhöhe *gaps*-Index }
15: $\quad p \leftarrow p + gaps[index]$ { Setze p auf nächste Primzahl }
16: **end while**
17: **return** *sieve*

Es wird beginnend mit Index 1 die Liste durchlaufen und sukzessive das nächste Element von *gaps* auf p aufaddiert und gesiebt.

Satz 3.8 Algorithmus 3.9 siebt die Primzahlen aus dem Intervall $[3, n]$ in der Zeit

$$\begin{aligned} t_{ADL}(n) &= (31B + S(\pi(\sqrt{n})/2)) \cdot \pi(\sqrt{n}) + (36B + S(n/4B)) \cdot N(3, \sqrt{n}, n)/2 + \\ &\quad + (36B + S(\sqrt{n}/4B)) \cdot N(3, \sqrt[4]{n}, \sqrt{n})/2 + O(\sqrt{n} \log n) \\ &= (36B + S(n/4B)) \cdot N(3, \sqrt{n}, n)/2 + O(\sqrt{n} \log n \log\log n) \\ &= O(n \log n \log\log n) \end{aligned}$$

unter Verwendung von $s_{ADL}(n) = \pi(\sqrt{n}) \cdot B + \sqrt{n}/2 + n/2 + O(1)$ Bits Speicher aus.

Beweis. Beginnend mit der Primzahl 3 und Index 2 wird sukzessive das Feld *gaps* durchlaufen, um die Differenz zur nächsten Primzahl und somit schließlich alle möglichen Faktoren kleiner gleich $\sqrt{n}$ zu erhalten. Der wesentliche Anteil an der Zeitkomplexität bleibt gleich. Wegen $\pi(\sqrt{n}) = O(\sqrt{n}/\log n)$ und $N(3, \sqrt[4]{n}, \sqrt{n}) = O(\sqrt{n} \log\log n)$ ist dieser wiederum (durch die Zeilen 10–13) von der Größenordnung $n \log n \log\log n$. Temporär wird ein zusätzliches Feld der Länge $\sqrt{n}/2$ in *pdiff* zum Sieben des Intervalls $[2, \sqrt{n}]$ benötigt. $\square$

Das resultierende Verfahren ist nicht etwa schneller, sondern unwesentlich langsamer als Algorithmus 3.7. Jedoch ist die Kenntnis der ersten $\pi(\sqrt{n})$ Primzahlen unerläßlich bei der nun folgenden Segmentierung. Zudem muß zugute gehalten werden, daß die Generierung der Primzahlliste bis $\sqrt{n}$ dann nur ein einziges Mal erfolgen muß.

3.3.3 Erste segmentierte Version

Algorithmus 3.11 teilt das zu siebende Gesamtintervall in Teilstücke gerader Länge auf und siebt diese. Die Länge der Segmente wird dabei zunächst als Parameter übergeben. Zunächst aber Algorithmus 3.10 zum Sieben eines Intervalls:

Algorithmus 3.10 *sieve1seg*

```
gaps ← input()   { Eingabe des Lückenfeldes }
iMin1 ← input()   { Eingabe der Segmentnummer }
l ← input()   { Eingabe des Segmentlänge }
first ← 1 + l × iMin1   { Erste Zahl des Segments }
last ← first + l − 1   { Letzte Zahl des Segments }
sqrtLast ← sqrt(last)   { Bestimmung des größtmöglichen Faktors }
index ← 1   { Index für gaps-Feld }
p ← 3
while p ≤ sqrtLast do   { Durchlaufe Primzahlen ≤ √last }
  twop ← p + p   { p · 2 }
  next ← first % p   { Bestimme ersten „Treffer“ im Segment }
  if next = 0 then   { Falls Division aufgegangen }
    next ← 1   { Erster Treffer ist first selbst }
  else
    next ← p − next + 1   { ansonsten addiere p + 1 auf p × (first ÷ p) }
  end if
  if next ∧ 1 = 0 then   { Treffer war gerade, ein p aufaddieren }
    next ← next + p
  end if
  while next ≤ l do   { Markiere Vielfache von p }
    setBit2(segment, next)   { Markiere zerlegbare Zahl }
    next ← next + twop   { Nächstes Vielfaches }
  end while
  index ← index + 1   { Erhöhe gaps-Index }
  p ← p + gaps[index]   { Setze p auf nächste Primzahl }
end while
return segment
```

Lemma 3.9 Es bezeichne l die Länge eines Segments. Algorithmus 3.10 siebt die Primzahlen aus einem Intervall $[(i-1)\cdot l+1, i\cdot l]\cap[3,n], i\in[1,n/l]$ in der Zeit

$$\begin{aligned} t_{ADL}(i,l) &= (45B + D(B) + S(\pi(\sqrt{i\cdot l})/2))\cdot\pi(\sqrt{i\cdot l}) \\ &\quad +(36B + S(l/4B))\cdot N^*(3,\sqrt{i\cdot l},l)/2 + O(1) \\ &= (36B + S(l/4B))\cdot N^*(3,\sqrt{i\cdot l},l)/2 + O(\sqrt{i\cdot l}) \\ &= O(l\log l\log\log(i\cdot l)) \end{aligned}$$

unter Verwendung von $s_{ADL}(l) = l/2 + O(1)$ Bits Speicher aus. Dabei ist

$$N^*(j,k,m) = \sum_{\substack{j \le p \le k \\ p \text{ prim}}} \left\lfloor \frac{m}{p} \right\rfloor$$

Beweis. Die Zeilen 11–19 dienen der Ermittlung der Bitposition des ersten Vielfachen des gerade aktuellen p im Intervall. Das Intervall $[(i-1)\cdot l + 1, i \cdot l]$ wird dabei auf $[1, l/2]$ abgebildet. Alle Vielfachen des jeweiligen p im aktuellen Intervall werden in der Zeile 21 vermerkt. Für jedes $p \le \sqrt{i \cdot l}$ werden die Zeilen 21 und 22 $\lfloor l/p \rfloor$ mal ausgeführt, insgesamt also $N^*(3, \sqrt{i \cdot l}, l)$ mal. □

Algorithmus 3.10 wird nun verwendet:

Algorithmus 3.11 *segerat1*

$n \leftarrow input()$
$l \leftarrow input()$
$sqrtn \leftarrow sqrt(n)$ { Berechne Wurzel }
$gaps \leftarrow pdiff(sqrtn)$ { Erzeugung des Lücken-Feldes }
if $l \wedge 1$ **then** { Nur gerade Segmentlängen }
  $l \leftarrow l + 1$
end if
$n_{segs} \leftarrow n \div l$ { Bestimme Anzahl der Segmente }
if $l \times n_{segs} < n$ **then** { Falls n nicht durch l teilbar ist }
  $n_{segs} \leftarrow n_{segs} + 1$ { Ein Segment mehr }
end if
$wordsMin1 \leftarrow l \div (2 \times B)$ { Anzahl der Wörter pro Segment - 1 }
$segment \leftarrow erat5(l)$ { Siebe erstes Segment mit Eratosthenes 5 }
for $i \leftarrow 2$ **to** n_{segs} **do** { Siebe Segmente }
  for $k \leftarrow 0$ **to** $wordsMin1$ **do** { Durchlaufe *segment* }
    $segment[k] \leftarrow 0$ { Setze Feldelemente zurück }
  end for
  $segment \leftarrow sieve1seg(gaps, i-1, l)$ { Ein Segment sieben }
end for

Satz 3.10 Algorithmus 3.11 siebt die Primzahlen aus dem Intervall $[3, n]$ in der Zeit

$$\begin{aligned} t_{ADL}(n,l) &= O(\sqrt{n} \log n \log\log n) + O(l \log l \log\log l) + \sum_{i=1}^{\lceil n/l \rceil} (12B + (l/2B + 1) \cdot \\ &\quad \cdot (7B + S(l/4B)) + (36B + S(l/4B)) \cdot N^*(3, \sqrt{i \cdot l}, l)/2 + O(\sqrt{i \cdot l})) \\ &= O((n^{\frac{3}{2}}/l) \log l \log\log n) + O(l \log l \log\log l) \end{aligned}$$

unter Verwendung von $s_{ADL}(n,l) = \pi(\sqrt{n})B + \max(l/2, \sqrt{n}/2) + O(1)$ Bits Speicher aus.

Beweis. Die Erzeugung der Liste der Primzahldifferenzen benötigt $O(\sqrt{n}\log n\log\log n)$ Bitoperationen. Das erste Segment (Segment 0) wird mit Algorithmus 3.7 gesiebt und trägt mit $O(l\log l\log\log l)$ Bitoperationen bei. Die Segmente 1 bis $\lceil n/l\rceil$ benötigen jeweils $(36B + S(l/4B))\cdot N^*(3,\sqrt{i\cdot l},l)/2 + O(\sqrt{i\cdot l})$. Wegen $\sqrt{i\cdot l} = O(\sqrt{n})$, $S(l/4B) = O(\log l)$ und $N^*(3,\sqrt{i\cdot l},l) = l\log\log l + O(1)$ wird insgesamt $t_{ADL}(n,l) = O((n/l)\cdot\sqrt{n}\cdot\log l\log\log n) + O(l\log l\log\log l)$. $\square$

3.3.4 Diskussion

Für $l = \sqrt{n}$ wird für 3.11 $t_{ADL}(n,\sqrt{n}) = O(n\log n\log\log n)$ sowie $s_{ADL} = O(\sqrt{n})$, was einen Anhaltspunkt für die optimale Segmentlänge liefert. Allerdings wird noch eine wesentliche Verbesserung vorgenommen, die wiederum Einfluß auf die Segmentlänge hat, so daß $l = \sqrt{n}$ wirklich nur als Anhaltspunkt zu sehen ist. Auch kann $\sqrt{n}\cdot B$ bereits den zur Verfügung stehenden Hauptspeicherbereich überschreiten, so daß zur Speicherreduzierung eine etwas höhere Laufzeit in Kauf genommen werden muß.

Es sollte nicht unterschlagen werden, daß durch die Segmentierung eine Eigenschaft des Eratosthenes-Siebes verloren gegangen ist. Es sind zu keinem Zeitpunkt alle Primzahlen des Intervalls $[1,n]$ vorhanden. Natürlich könnte man dies durch eine Ausgabe der einzelnen Segmente „reparieren“, aufgrund der dadurch entstehenden Kosten ist dies allerdings keine eigentliche Lösung. Es gibt Probleme, die die Kenntnis zumindest einer Teilmenge verschiedener Segmente benötigen. Ein solches Problem wird in Kapitel 5 zur Sprache kommen.

Es folgt nun eine genauere Betrachtung der Funktion $N^*(j,k,m)$. Abbildung 8 zeigt einen Plot der Punkte $(p,\lfloor n/p\rfloor)$ am Beispiel $n = 10^{10}, p \leq \sqrt{n}$:

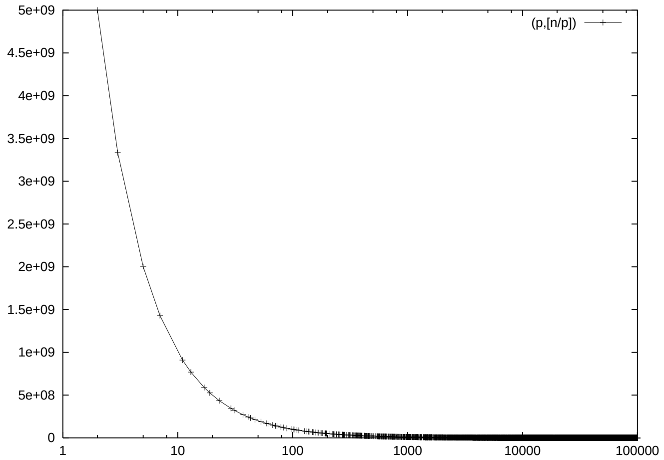


Abbildung 8: $(p,[10^{10}/p]), p \leq 10^5$

Es wird deutlich, daß die wesentlichen Kosten beim Sieben mit kleinen Primzahlen p entstehen, wo $\lfloor \frac{n}{p} \rfloor$ groß wird. Abbildung 9 zeigt die Entwicklung der Gesamtkosten in Abhängigkeit der zuletzt gesiebten Primzahl.

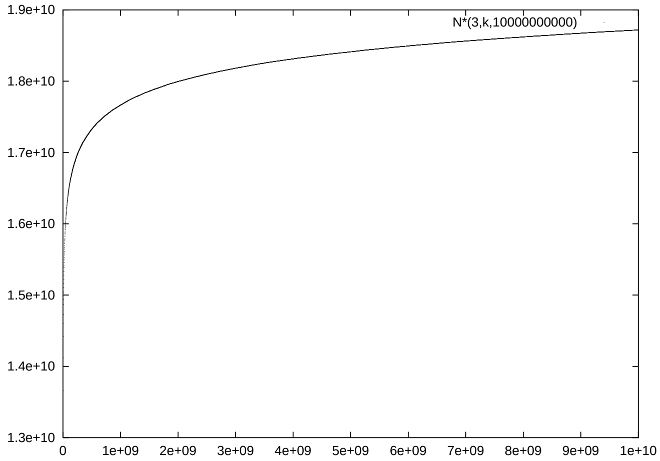


Abbildung 9: $(k, N^*(3, k, 10^{10}))$, $k \leq 10^5$

Die meisten Operationen werden also für die Faktoren kleiner p vorgenommen. Es soll nun für verschiedene m diejenige Primzahl $p = h(n)$ bestimmt werden, bis zu der die Hälfte aller Operationen stattgefunden haben, also die „halbe Arbeit" im Sinne der Anzahl der Operationen erledigt ist. Gesucht wird jeweils $h(n)$ mit $2 \cdot N^*(3, h(n), n) \geq N^*(3, \sqrt{n}, n)$ und $2 \cdot N^*(3, h(n) - 1, n) < N^*(3, \sqrt{n}, n)$. Der schon gesparte Faktor 2 wurde dabei nicht mit aufgenommen. Zu erwarten ist wegen $N^*(3, k, m) = O(m \log\log k)$, daß $h(n) = O(e^{\sqrt{\frac{1}{2}\log n}})$. Statt die O-Konstante nun abzuschätzen, wurde $h(n)$ für einige n experimentell ermittelt.

n	$h(n)$	n	$h(n)$
10^3	5	10^9	29
10^4	7	10^{10}	37
10^5	11	10^{11}	43
10^6	13	10^{12}	53
10^7	17	10^{13}	67
10^8	23	10^{14}	73

Tabelle 12: Die Funktion $h(n)$

3.3.5 Vorsieben kleiner Primfaktoren

Tabelle 12 legt nahe, einen Weg zu finden, das Sieben kleiner Primfaktoren zu vermeiden. Dies kann folgendermaßen geschehen: Zunächst wird ein Feld vorbereitet, in dem alle Vielfachen kleiner Primzahlen $3, 5, 7, \ldots, p_k$ markiert werden. Für jedes zu siebende Segment wird das vorbereitete Feld dann kopiert. Damit sind die vorgesiebten Faktoren nicht mehr zu behandeln, nur mit Primzahlen zwischen p_{k+1} und $p_{\pi(\sqrt{n})}$ muß gesiebt werden. Es ergibt sich daraus allerdings die Einschränkung, daß die Segmentlänge ein Vielfaches von $M_k := \prod_{i=1}^{k} p_i$ sein muß. Algorithmus 3.12 realisiert das Vorsieben bis zu einem p_k, wobei ein Feld *primes* aller kleinen Primzahlen $2, 3, \ldots, p_j$ mit $j \geq k$ vorhanden sei. Dabei wird nicht p_k an 3.12 übergeben, sondern die vorher festzulegende Segmentlänge.

Algorithmus 3.12 *presieve*

1: $l \leftarrow input()$
2: $i \leftarrow 0$
3: $next \leftarrow primes[1]$ { primes[0]=2 }
4: **while** $l \% next = 0$ **do** { Durchlaufe Primteiler der Segmentlänge }
5: $\quad twop \leftarrow p + p$
6: $\quad$ **while** $next \leq l$ **do** { Markiere Vielfache von p }
7: $\quad\quad setBit2(psieve, next)$ { Markiere zerlegbare Zahl }
8: $\quad\quad next \leftarrow next + twop$ { Nächstes Vielfaches }
9: $\quad$ **end while**
10: $\quad i \leftarrow i + 1$
11: $\quad next \leftarrow primes[i]$
12: **end while**
13: **return** *psieve*

Lemma 3.11 Es bezeichne $l = M_k \leq \sqrt{n}$ mit $M_{k+1} > \sqrt{n}$ die Länge eines Segments. Algorithmus 3.12 markiert die Vielfachen der Primzahlen p mit $p \mid l$ in

$$\begin{aligned} t_{ADL}(l) &= 28B \cdot k + S(k/2) + (36B + S(l/4B)) \cdot N^*(3, p_k, l) + O(1) \\ &= (36B + S(l/4B)) \cdot N^*(3, p_k, l) + O(\log l) \\ &= O(l \log l \log\log((\log\log l) \log l)) \end{aligned}$$

Bitoperationen unter Verwendung von $s_{ADL}(n) = l/2 + O(1)$ Bits Speicher.

Beweis. Der wesentliche Anteil ist wiederum die innere **while**-Schleife Die Zeilen 7 und 8 werden $N^*(3, p_k, l)$ mal durchlaufen. Wegen A.5, $k = O(\log l)$ und $S(l/4B) = O(\log l)$ wird insgesamt $N^*(3, p_k, l) = O(l \cdot \log\log((\log\log l) \log l))$. Wegen $l = M_k$ gilt später immer $i + j \cdot l \equiv i \mod M_k$ und somit auch $i + j \cdot l \equiv i \mod p_m$ für alle $m \leq k$. □

Die Algorithmen 3.10 und 3.11 müssen nun nur unwesentlich verändert werden.

Algorithmus 3.13 *segerat2*

```
1: n ← input()
2: l ← input()
3: sqrtn ← sqrt(n)   { Berechne Wurzel }
4: gaps ← pdiff(sqrtn)   { Erzeugung des Lücken-Feldes }
5: psieve ← presieve(l)
6: n_segs ← n ÷ l   { Bestimme Anzahl der Segmente }
7: if l × n_segs < n then   { Falls n nicht durch l teilbar ist }
8:    n_segs ← n_segs + 1   { Ein Segment mehr }
9: end if
10: wordsMin1 ← l ÷ (2 × B)   { Anzahl der Wörter pro Segment - 1 }
11: segment ← erat5(l)   { Siebe erstes Segment mit Eratosthenes 5 }
12: for i ← 2 to n_segs do   { Siebe Segmente }
13:    for k ← 0 to wordsMin1 do   { Durchlaufe segment }
14:       segment[k] ← psieve[k]   { Kopiere Vorsieb }
15:    end for
16:    segment ← sieveSeg(gaps, i − 1, l)   { Ein Segment sieben }
17: end for
```

In Algorithmus 3.10 muß nun nur noch Zeile 8 durch folgende Zeilen ersetzt werden, um das Sieben der ersten k Primzahlen zu vermeiden:

```
while l%primes[index] = 0 do
   index ← index + 1
end while
p ← primes[index]
```

Satz 3.12 Es sei wieder $l = M_k \leq \sqrt{n}$ mit $M_{k+1} > \sqrt{n}$ die Länge eines Segments. Algorithmus 3.13 siebt die Primzahlen aus dem Intervall $[3, n]$ in der Zeit

$$\begin{aligned} t_{ADL}(n,l) &= O(l \log l \log\log((\log\log l)\log l)) + O(\sqrt{n}\log n \log\log n) + O(l \log l \log\log l) + \\ &\quad + \sum_{i=1}^{\lceil n/l \rceil} (12B + (l/2B + 1) \cdot (11B + 2 \cdot S(l/4B)) + \\ &\quad +(36B + S(l/4B)) \cdot N^*(p_{k+1}, \sqrt{i \cdot l}, l)/2 + O(\sqrt{n})) \\ &= O((n^{\frac{3}{2}}/l) \log l \log\log n) + O(l \log l \log\log l) \end{aligned}$$

unter Verwendung von $s_{ADL}(n) = \pi(\sqrt{n})B + \max(l/2, \sqrt{n}/2) + O(1)$ Bits Speicher aus.

Beweis. Die einzigen Unterschiede zu 3.11 liegen im Aufruf von *presieve* in Zeile 5 und der Ersetzung der 0 auf der rechten Seite von Zeile 16 in 3.11 durch *psieve*$[k]$. *presieve* wird nur einmal aufgerufen und verursacht daher keine wesentlichen Kosten. Die Kopie des Vorsiebes erfordert nur zusätzliche $3B + S(l/4B)$ Schritte. Zur Korrektheit: Das Sieben der kleinen Primfaktoren $\leq p_k$ wird nun durch das Kopieren des vorgesiebten Feldes erzielt. Da die Länge eines Segments ein Vielfaches des Produkts der ersten k Primzahlen ist, sind für jedes Segment die Positionen m mit $m = j \cdot p_i$, $i \leq k$ vor dem Aufruf von *sieveSeg* bereits markiert. □

Tabelle 13 zeigt die zu erwartende, prozentuale Zeitersparnis bei Vorsieben aller Primzahlen bis einschließlich p_k für die Intervalle $[1, 10^8]$ und $[1, 10^{14}]$. Dabei ergeben sich die Spalten 2 und 3 durch $100 \cdot (1 - N^*(p_k, 10^{i/2}, 10^i)/N^*(3, 10^{i/2}, 10^i))$.

p_k	$[1, 10^{14}]$	$[1, 10^8]$
3	13,1	16,8
5	21,0	26,9
7	26,6	34,1
11	30,2	38,7
13	33,2	42,6
17	35,5	45,5
19	37,6	48,2
23	39,3	50,4

Tabelle 13: Zu erwartende, prozentuale Zeitersparnis durch Vorsieben bis p_k

Nach Satz 3.10 liegt die optimale Segmentlänge l bei $\sqrt{n}$, da bei dieser Wahl einerseits die Zeitkomplexität $O(n \log n \log\log n)$ erreicht, andererseits die Raumkomplexität die Größenordnung $\sqrt{n}$ nicht überschreitet. Durch die Einschränkung, daß bei Vorsieben von Primzahlen kleiner gleich p_k die Segmentlänge ein Vielfaches von M_k sein muß, ist $l = \sqrt{n}$ im allgemeinen nicht genau zu erreichen. In Abschnitt 3.4 werden die optimalen Längen für verschiedene n in Abhängigkeit eines weiteren Parameters bestimmt werden.

3.3.6 Weitere Verbesserungen

In diesem Abschnitt sollen weitere Optimierungsmöglichkeiten aufgezeigt werden. Zunächst wird eine kleine Veränderung an den Pseudofunktionen *setBit* und *getBit* vorgenommen, die für jede Markierung eine Operation spart. Danach wird ein Problem bearbeitet, das an früherer Stelle schon erwähnt worden ist und dessen (Teil-)Lösung eine weitere, entscheidende Verbesserung liefert.

Verbesserung der Bitoperationen Die in der Definition von $setBit2(sieve, next)$ und $getBit2(sieve, p)$ auftauchenden Operationen $next \gg 1$ bzw. $p \gg 1$ können folgendermaßen vermieden werden: Da das Intervall $[1, n]$ im Sieb auf ein Feld $[1, n/2B]$ abgebildet wird, in dem keine geraden Zahlen mehr existieren, kann das Durchlaufen von $[1, n]$ durch das Durchlaufen von $[1, n/2]$ ersetzt werden. Zunächst wird also der erste Treffer im Intervall durch 2 geteilt. Nun wird jeweils nicht mehr *twop*, also das Doppelte der Primfaktoren addiert, sondern p selbst. Es ergeben sich die modifizierten und nun „endgültigen“ Funktionen *setBit*:

$$sieve[next \gg b] \leftarrow sieve[next \gg b] \vee (1 \ll (next \wedge (B-1)))$$

sowie *getBit*:

$$(sieve[p \gg b] \wedge (1 \ll (p \wedge (B-1)))$$

Algorithmus 3.10 wird nun folgendermaßen verändert:

Algorithmus 3.14 *sieve1seg3*

```
gaps ← input()   { Eingabe des Lückenfeldes }
iMin1 ← input()   { Eingabe der Segmentnummer }
l ← input()   { Eingabe des Segmentlänge }
first ← 1 + l × iMin1   { Erste Zahl des Segments }
last ← first + l − 1   { Letzte Zahl des Segments }
l_2 ← l ≫ 1   { Bestimme halbe Segmentlänge }
sqrtLast ← sqrt(last)   { Bestimmung des größtmöglichen Faktors }
while l%primes[index] = 0 do  { Durchlaufe Primteiler der Segmentlänge }
  index ← index + 1   { Justiere Index }
end while
p ← primes[index]   { Erste zu siebende Primzahl }
while p ≤ sqrtLast do  { Durchlaufe Primzahlen ≤ √last }
  next ← first % p   { Bestimme ersten „Treffer“ im Segment }
  if next = 0 then  { Falls Division aufgegangen }
    next ← 1   { Erster Treffer ist first selbst }
  else
    next ← p − next + 1   { ansonsten addiere p + 1 auf p × (first ÷ p) }
  end if
  next ← next ≫ 1   { Justiere ersten Treffer }
  while next ≤ l_2 do  { Markiere Vielfache von p }
    setBit(segment, next)  { Markiere zerlegbare Zahl }
    next ← next + p   { Nächstes Vielfaches }
  end while
  index ← index + 1   { Erhöhe gaps-Index }
  p ← p + gaps[index]   { Setze p auf nächste Primzahl }
end while
return segment
```

Lemma 3.13 Es sei l definiert wie in Lemma 3.11 und $i \in [1, n/l]$. Algorithmus 3.14 siebt die Primzahlen aus einem Intervall $[(i-1) \cdot l + 1, i \cdot l] \cap [3, n]$ in der Zeit

$$\begin{aligned} t_{ADL}(i,l) &= (40B + D(B) + S(\pi(\sqrt{i \cdot l})/2)) \cdot \pi(\sqrt{i \cdot l}) \\ &\quad +(33B + S(l/4B)) \cdot N^*(p_{k+1}, \sqrt{i \cdot l}, l)/2 + O(1) \\ &= (33B + S(l/4B)) \cdot N^*(p_{k+1}, \sqrt{i \cdot l}, l)/2 + O(\sqrt{i \cdot l}) \\ &= O(l \log l \log\log(i \cdot l)) \end{aligned}$$

unter Verwendung von $s_{ADL}(l) = l/2 + O(1)$ Bits Speicher aus.

Beweis. Durch die Vermeidung der Operation $next \ll 1$ sind $3B$ Schritte eingespart worden. Zur Korrektheit: Anstatt die ganzen Faktoren zu durchlaufen, um die Abbildung auf das Intervall $[1, l/2]$ dann in *setBit* vorzunehmen, werden sofort nur halbe Faktoren durchlaufen. Die Halbierung der Vielfachen der zu siebenden p wurde also lediglich von *setBit* verschoben. □

Es sei nun Algorithmus 3.15 (Segmentiertes Sieb 3) wie 3.13, allerdings mit den modifizierten Bitoperationen. Darüber hinaus sei die entsprechende Veränderung in $erat5$ durch Ersetzung von $next$ in Zeile 10 durch $next \gg 1$ gegeben. Dann folgt:

Satz 3.14 Algorithmus 3.15 siebt die Primzahlen aus dem Intervall $[3, n]$ in der Zeit

$$\begin{aligned} t_{ADL}(n) &= O(l \log l \log\log((\log\log l) \log l)) + O(\sqrt{n} \log n \log\log n) + O(l \log l \log\log l) + \\ &\quad + \sum_{i=1}^{\lceil n/l \rceil} (12B + (l/2B + 1) \cdot (11B + 2 \cdot S(l/4B)) + \\ &\quad +(33B + S(l/4B)) \cdot N^*(p_{k+1}, \sqrt{i \cdot l}, l)/2 + O(\sqrt{n})) \\ &= O(n \log n \log\log n) \end{aligned}$$

unter Verwendung von $s_{ADL}(n) = \pi(\sqrt{n})B + \max(l/2, \sqrt{n}/2) + O(1)$ Bits Speicher aus.

Beweis. Der Unterschied der Zeitkomplexität ergibt sich aus der Ersetzung der Operation $next \gg 1$ durch $next$ in *setBit*, womit $3B$ eingespart werden. □

3.3.7 Laufzeitanalyse der segmentierten Siebe

Es folgt nun eine Laufzeitanalyse der segmentierten Siebe 1 – 3. Betrachtet wird das Intervall $[1, 10^8]$ in Schritten der Länge 10^5. Die Segmentlänge von Algorithmus 3.11 ist dabei jeweils $\sqrt{n}$, die von 3.13 und 3.15 gleich $M_6 = 30030$. Die erwartete Verbesserung von etwa 43% durch das Vorsieben wird dabei in den Abbildungen 10 – 13 sehr gut sichtbar.

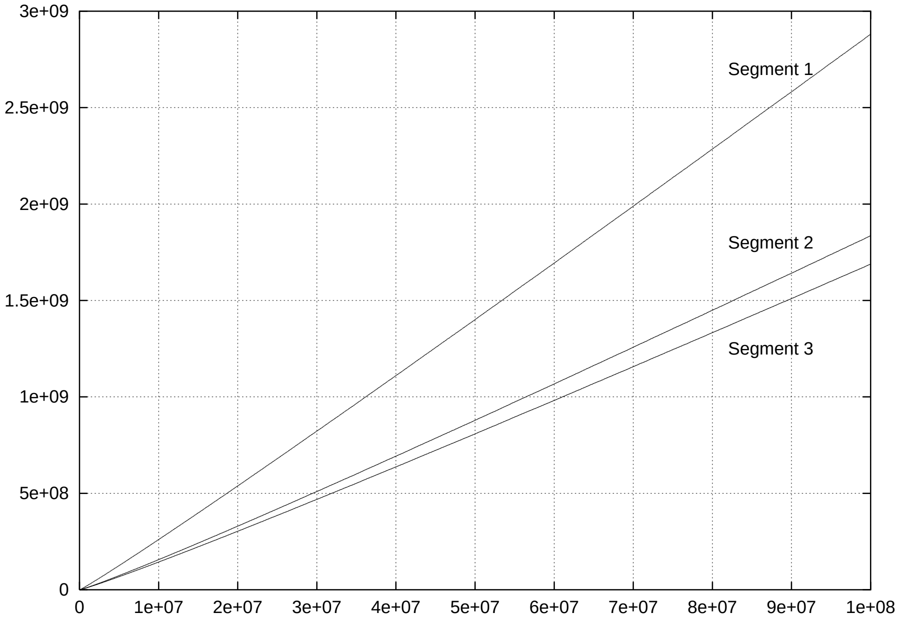


Abbildung 10: ADL-Zeitkomplexitäten (SPARCstation-4)

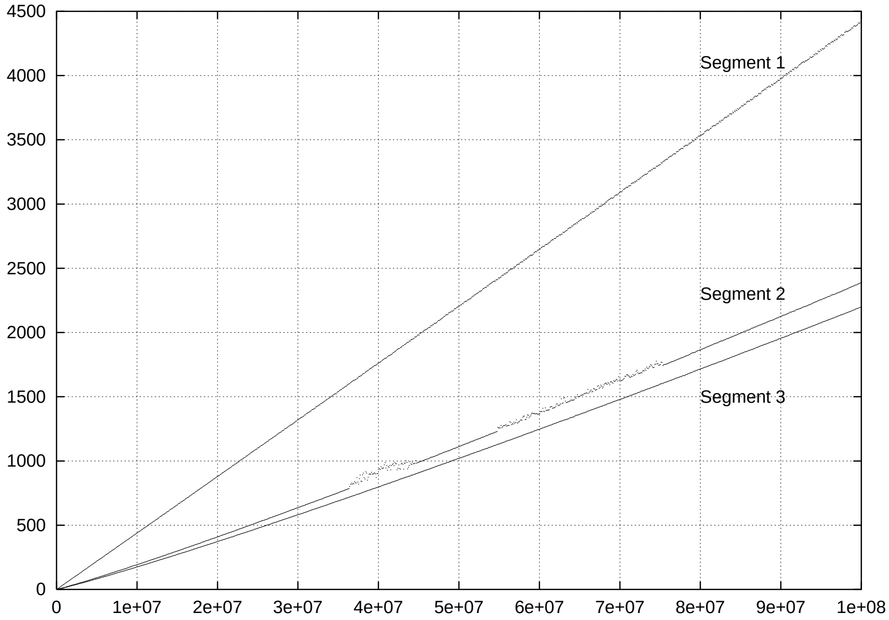


Abbildung 11: Reale Laufzeiten (SPARCstation-4)

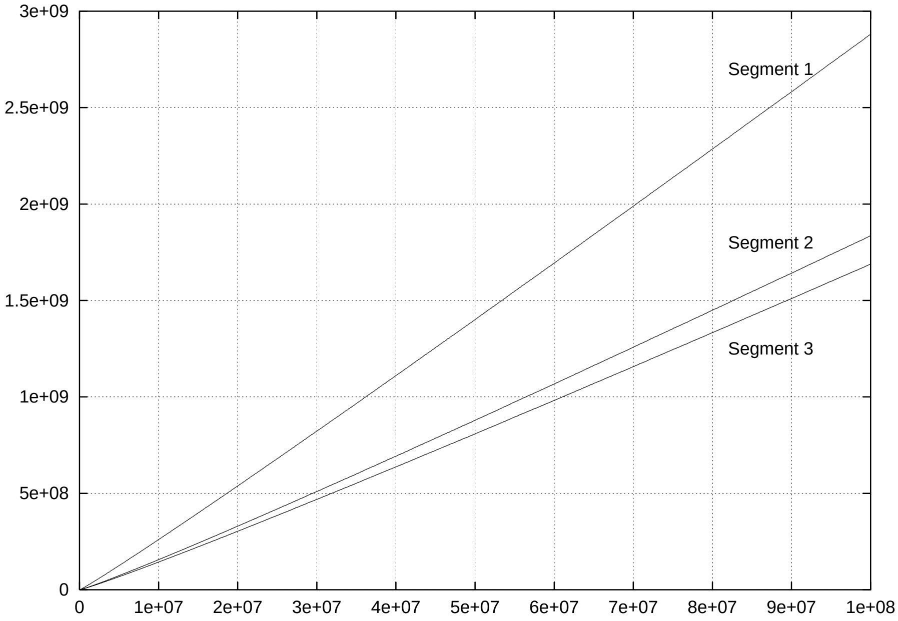


Abbildung 12: ADL-Zeitkomplexitäten (Ultra-1)

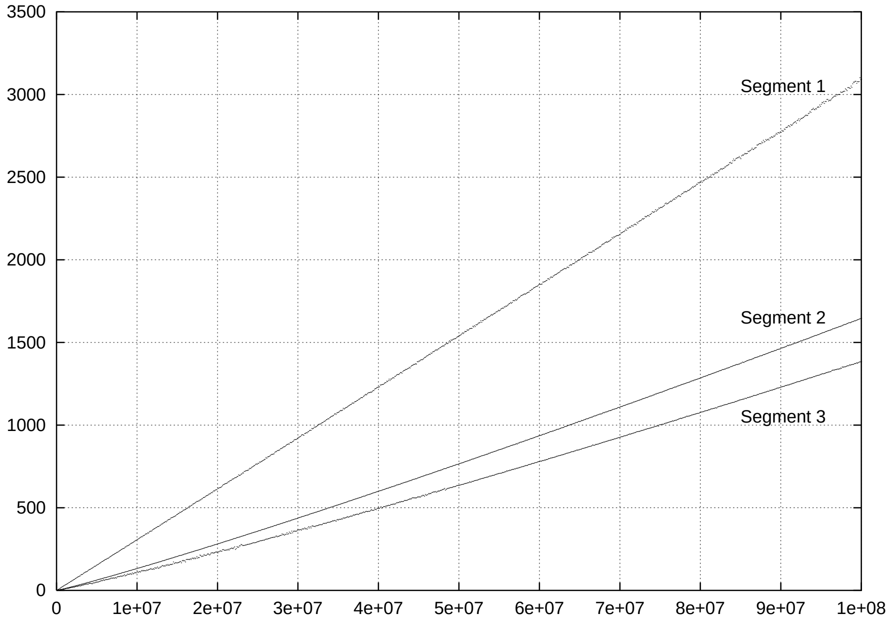


Abbildung 13: Reale Laufzeiten (Ultra-1)

Vermeidung von Mehrfachmarkierungen Ein Problem aller bisherigen Siebversionen ist, daß Mehrfachmarkierungen stattfinden. So ist für jedes zu siebende p_j jede dritte, jede fünfte, jede siebte … jede p_{j-1}-te Operation überflüssig, denn diese Positionen sind bereits als zerlegbar markiert.

Es wird nun beschrieben, wie zumindest jede dritte und jede fünfte überflüssige Operation vermieden werden kann. Die Formulierung findet aus Platzgründen nicht als gesondert aufgeführter Algorithmus statt, sondern wird als Teil des in 3.4 vorgestellten Primzahlgenerators implementiert.

Das Prinzip ist das folgende: Für jede zu siebende Primzahl kleiner einer vorgegebenen Schranke wird solange markiert, bis das entsprechende Produkt ein Vielfaches von $3 \cdot 5 = 15$ ist. Dann wird nicht mehr jeweils ein p addiert, sondern folgende Sequenz (deren Faktoren sich im Grunde in 3.5.1 als $W_3[x]$ wiederfinden): $p, 2p, 3p, p, 3p, 2p, p, 2p$. Alle übrigen Markierungen sind Vielfache von 3 und/oder 5 und werden daher übersprungen. Natürlich könnte man in diese Sequenz auch einspringen, dies erforderte aber eine zusätzliche Fallunterscheidung, die nicht billiger als die Justierung ist. Während der Entwicklung der im folgenden Abschnitt vorgestellten Implementierung eines Primzahlsiebes wurde auch die Vermeidung von Vielfachen der 7 getestet, dies führte allerdings zu keiner Verbesserung mehr, da die Justierungsschritte (bzw. die Fallunterscheidung) den Vorteil aufhoben. Sogar die Vermeidung aller zusätzlicher Markierungen ist möglich, dies wird in Abschnitt 3.5.1 beschrieben.

3.4 Der Primzahlgenerator pg

Die in den letzten Abschnitten gewonnenen Erkenntnisse werden nun zusammengefaßt und als Primzahlgenerator-Software `pg` implementiert. Auf eine komplette Beschreibung in *ADL* wird hier aus Platzgründen verzichtet. Allerdings wird die wesentliche Siebprozedur in *ADL* formuliert und analysiert.

Die Basis von `pg` ist im Grunde das segmentierte Sieb 3 mit der zusätzlichen Implementierung einer Routine zur Vermeidung der Mehrfachmarkierungen, die in Algorithmus 3.17 realisiert wird.

Das Ergebnis dieses Abschnitts stellt einen wesentlichen Teil der in Kapitel 4 vorgestellten Berechnung dar. In 5 und 6 wird `pg` als Hilfsmittel benutzt.

3.4.1 Anforderungen

Neben den grundlegenden Anforderungen wie etwa der vernünftigen Einteilung in Programmodule sowie der Übersichtlichkeit und ausreichenden Dokumentation der Quellen standen bei der Entwicklung von `pg` die folgenden, speziellen Punkte im Vordergrund:

- Einfache Erweiterbarkeit zur späteren Anwendung
- Sieben beliebiger Teilintervalle
- Freie Wahl der Segmentlänge (bei Beachtung der Restriktion, die aus dem Vorsieben resultiert)
- „Haltbarkeit“ durch wählbare Wortlängen bis 128 Bits
- Wahl, ob ein 0- oder 1-Bit eine Primzahl repräsentiert (wichtig z.B. für Kapitel 4)
- Wahl, bis zu welchem Faktor Vielfache von 3 und 5 nur einmal gesiebt werden
- Intensive Verwendung der Möglichkeiten des C-Präprozessors zur Optimierung
- Verwendung von Standard-(C-)Sprachkonstrukten zur leichten Portierbarkeit
- Vermeidung der direkten Verwendung von Grunddatentypen

Der Grund für den letzten Punkt ist dabei wiederum die Möglichkeit zur leichten Portierung, da die Länge der Grunddatentypen in C nicht genau festgelegt ist. Die direkte Vermeidung wurde wiederum durch den Einsatz des C-Präprozessors realisiert.

3.4.2 Softwareaufbau

Der Primzahlgenerator pg setzt sich aus 14 verschiedenen Quelldateien zusammen.

Zweck und Bedeutung der einzelnen Module sind in Tabelle 14 aufgelistet:

Dateiname	Bedeutung/Zweck
pg.c	Hauptprogramm
pg.h	Nicht zu editierende Definitionen
pg_defs.h	Benutzeränderbare Definitionen
pg_get_primes.c	Durchlaufen fertig gesiebter Intervalle
pg_init.c	Initialisierung
pg_load_gaps.c	Laden der Primzahllücken
pg_mem.c	Speicherallokation
pg_msg.c	Ausgabe-/Fehlerfunktionen
pg_next_prime.c	(Optionaler) Code, falls Primzahl gefunden
pg_next_gap.c	(Optionaler) Code bei Ermittlung der nächsten Differenz
pg_scan_args.c	Scannen der Kommandozeilenparameter
pg_sieve.c	Siebprozedur
pg_sieve_ps.c	Siebprozedur für Vorsieb
pg_utils.c	Werkzeuge

Tabelle 14: pg-Module

Die in 3.4.1 erwähnten, zu wählenden Parameter werden durch Präprozessoranweisungen (#define's) in der Datei pg_defs.h festgelegt. Eine wesentliche Rolle spielen dabei die Konstanten SIEVELENLONGS (Länge der Segmente in Maschinenworten) sowie MUL_3_5_THRESHOLD (Maximaler Faktor, bis zu dem Mehrfachankreuzungen von Vielfachen von 3 und 5 vermieden werden).

Die in pg_defs.h benutzerdefinierten Parameter werden von pg.h eingebunden. pg.h selbst wird in alle Quellen eingefügt und stellt somit sämtliche Definitionen überall zur Verfügung.

Ein geeignetes Makefile steuert dabei die bedingte Kompilation. pg kompilierte problemlos auf vier verschiedenen Sun-Workstations unter zwei verschiedenen Betriebssystemversionen, drei verschiedenen Linux-PC's sowie einer IBM PowerPC-Workstation unter AIX.

3.4.3 Programmablauf

Beim Programmaufruf wird über die Kommandozeile das zu siebende Intervall übergeben und in `pg_scan_args` geprüft. Während der Initialisierungsphase werden die benötigten Speicherbereiche alloziert, die Primzahldifferenzen geladen und das Vorsieb vorbereitet. Danach beginnt innerhalb einer Schleife der eigentliche Siebvorgang. In `pg_sieve` (siehe auch Algorithmus 3.16) wird dabei jeweils ein Teilsegment gesiebt. Nach dem Sieben wird das Intervall in `pg_get_primes` durchlaufen und in Abhängigkeit der Definitionen aus `pg_defs.h` jeweils beim Auftreten einer Primzahl bzw. Primzahldifferenz entweder die in `pg_next_prime` oder `pg_next_gap` befindlichen Anweisungen abgearbeitet. Vor allem aus Testgründen kann optional ein effizient implementiertes Zählen der Primzahlen erfolgen.

Der Grundablauf ist Abbildung 14 zu entnehmen.

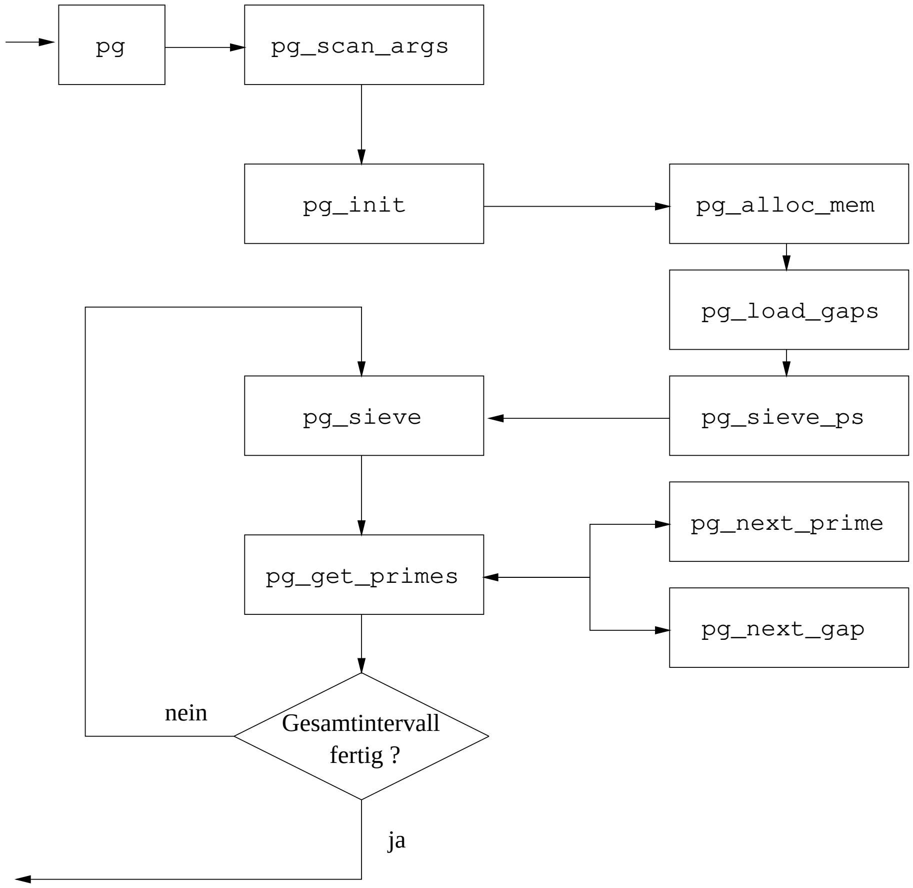


Abbildung 14: `pg`-Programmablauf

3.4.4 Die Siebprozedur `pg_sieve`

Die wesentliche Prozedur `pg_sieve` setzt sich aus zwei Teilen zusammen. Der in `pg` zu wählende Parameter `MUL_3_5_THRESHOLD` taucht hier als $m35$ auf. Bis zu diesem Wert finden keine Mehrfachmarkierungen von Vielfachen von 3 und 5 statt, danach wird normal gesiebt.

Algorithmus 3.16 *pg_sieve*

```
gaps ← input()   { Eingabe des Lückenfeldes }
iMin1 ← input()   { Eingabe der Segmentnummer }
l ← input()   { Eingabe des Segmentlänge }
l ← l ≫ 1   { Berechnung der halben Segmentlänge }
first ← 1 + l × iMin1   { Erste Zahl des Segments }
last ← first + l − 1   { Letzte Zahl des Segments }
sqrtLast ← sqrt(last)   { Bestimmung des größtmöglichen Faktors }
index ← 1   { Index für gaps-Feld }
while l%primes[index] = 0 do   { Bestimme erste zu siebende Primzahl }
  index ← index + 1
end while
p ← primes[index]   { Erste Primzahl, die die Segmentlänge nicht teilt }
while p ≤ sqrtLast do   { Durchlaufe Primzahlen ≤ √last }
  next ← first % p   { Bestimme ersten „Treffer“ im Segment }
  if next = 0 then   { Falls Division aufgegangen }
    next ← 1   { Erster Treffer ist first selbst }
  else
    next ← p − next + 1   { ansonsten addiere p + 1 auf p × (first ÷ p) }
  end if
  if next ∧ 1 = 0 then   { Treffer war gerade, ein p aufaddieren }
    next ← next + p
  end if
  if p < m35 then   { Vermeide doppeltes Markieren von Vielfachen von 3 und 5 }
    bigsteps()
  else
    next ← next ≫ 1
    while next ≤ l₂ do   { Markiere Vielfache von p }
      setBit(segment, next)   { Markiere zerlegbare Zahl }
      next ← next + p   { Nächstes Vielfaches }
    end while
  end if
  index ← index + 1   { Erhöhe gaps-Index }
  p ← p + gaps[index]   { Setze p auf nächste Primzahl }
end while
return segment
```

Algorithmus 3.17 beschreibt die Prozedur *bigsteps*, durch die die Vermeidung des wiederholten Markierens der Vielfachen von 3 und 5 realisiert wird:

Algorithmus 3.17 *bigsteps*

```
twop ← p + p   { p · 2 }
threep ← twop + p   { p · 3 }
nextmod15 ← next%15   { next muß Vielfaches von 15 werden }
twopmod15 ← twop%15   { Zur späteren, einfachen Berechnung von nextmod15 }
fifteenp ← (p ≪ 4) − p   { p · 15 }
while next ≤ l do  { Siebe mit p }
  if nextmod15 = 0 then  { Falls next Vielfaches von 15 }
    next ← (next ≫ 1) + p   { Halbiere next, addiere erstes p }
    if next ≤ l₂ then  { Überhaupt noch im Intervall? }
      setBit(segment, next)  { Markiere zerlegbare Zahl }
    else
      break
    end if
    while next + fifteenp ≤ l₂ do  { Addiere p, 2p, 3p, p, 3p, 2p, p, 2p }
      next ← next + p
      setBit(segment, next)
      next ← next + twop
      setBit(segment, next)
      next ← next + threep
      setBit(segment, next)
      next ← next + p
      setBit(segment, next)
      next ← next + threep
      setBit(segment, next)
      next ← next + twop
      setBit(segment, next)
      next ← next + p
      setBit(segment, next)
      next ← next + twop
      setBit(segment, next)
    end while
    while next ≤ l₂ do  { Restliche Zerlegbare markieren }
      setBit(segment, next)  { Markiere zerlegbare Zahl }
      next ← next + p
    end while
    break  { Vielfache aller p im Intervall markiert }
  else
    setBit(segment, next ≫ 1)  { Markiere zerlegbare Zahl }
    next ← next + twop
    nextmod15 ← nextmod15 + twopmod15
    if nextmod15 ≥ 15 then  { Vermindere gegebenenfalls nextmod15 um 15 }
      nextmod15 ← nextmod15 − 15
    end if
  end if
end while
return
```

Satz 3.15 Algorithmus 3.16 siebt die Primzahlen aus einem Intervall $[(i-1)\cdot l+1, i\cdot l]\cap [3,n], i\in[1,n/l]$

$$\begin{aligned}
t_{ADL}(i,l,m35) &= (33B+S(l/4B))\cdot N^*(m35,\sqrt{i\cdot l},l)/2+\\
&\quad +(598B+2\cdot D(B)+15\cdot S(l/4B))\cdot\pi(m35-1)+\\
&\quad +(232B+8\cdot S(l/4B))\cdot N^*(p_{k+1},m35,l)/30+O(1)\\
&= O(l\log l\log\log(i\cdot l))
\end{aligned}$$

unter Verwendung von $s_{ADL}(l)=l/2+O(1)$ Bits Speicher aus.

Beweis. Für Primfaktoren größer gleich $m35$ ist Algorithmus 3.16 identisch mit 3.14. Für $p\le m35$ werden im Durchschnitt 7,5 Justierungsschritte notwendig. Daher werden die Zeilen 39 – 44 insgesamt $7,5\cdot(\pi(\sqrt{i\cdot l})-\pi(m35-1))$-mal durchlaufen. Dasselbe gilt für die Zeilen 34 – 35. Die Zeilen 16 – 31 werden pro p $\lfloor l/(30\cdot p)\rfloor$-mal ausgeführt. Die asymptotische Laufzeit ändert sich wiederum nicht. Zur Korrektheit: Die Zeilen 6, 10 und 33 sorgen dafür, daß das Intervall nicht verlassen wird. Im Falle, daß $next$ kein Vielfaches von 15 ist, wird zunächst $next$ erhöht und jeweils markiert. Dies geschieht genau so lange, bis die Bedingung in Zeile 7 wahr wird. Dann wird zunächst $next/2+p$ markiert, falls dieser Wert noch im Intervall liegt. Danach wird solange die Sequenz $p,2p,3p,p,3p,2p,p,2p$ addiert und markiert wie $next+15\cdot p$ noch im Intervall liegen. Schließlich werden die möglicherweise noch zu markierenden, restlichen Zahlen durchlaufen. □

Die Aussage des Satzes 3.15 zeigt nicht offensichtlich, daß die Vermeidung des mehrfachen Ankreuzens von Vielfachen von 3 bzw. 5 eine Verbesserung darstellt.

Bemerkung. Durch die Zeilen 16 – 31 werden statt der normalerweise notwendigen 15 Schritte nur 8 ausgeführt. Allerdings wird dies nur dann zu einer echten Ersparnis führen, wenn diese Ausführung mindestens zweimal stattfindet, um die vorher notwendigen Justierungsschritte auszugleichen. Daher sollte der Wert von $m35$ etwa $\frac{1}{30}$ der Segmentlänge nicht überschreiten. Gerade bei den kleinen Primfaktoren lohnt sich der zusätzliche Aufwand jedoch merklich.

Im folgenden Abschnitt werden Ergebnisse von Laufzeitmessungen des Primzahlgenerators pg aufgelistet. Dabei sind einige Zeiten angegeben, die einmal mit $m35\approx l/30$ sowie einmal mit $m35=0$ gemessen wurden. Das Optimum von $m35$ wurde dabei für verschiedene Siebgrenzen und Maschinen experimentell bestimmt.

3.4.5 Laufzeitmessungen

Die Messungen zu den Tabellen 15 – 17 fanden sowohl auf den beiden bisher verwendeten Testmaschinen als auch aus Vergleichsgründen auf einem 200MHz Pentium PC unter Linux 2.0 statt. Alle Werte sind in CPU-Sekunden angegeben. Die erste Messung zeigt die minimalen Laufzeiten und die zugehörigen optimalen Parameter `MUL_3_5_THRESHOLD` und `SIEVELENLONGS` für Siebe bis 10^n, $n \in \{4, \ldots, 10\}$.

Intervallobergrenze	`MUL_3_5_THRESHOLD`	`SIEVELENLONGS`	Zeit
10^4	47500	$2^3 \cdot 5 \cdot 7 \cdot 11$	$<$ 0,01
10^5	47500	$2^3 \cdot 5 \cdot 7 \cdot 11$	$<$ 0,01
10^6	50000	$2^3 \cdot 5 \cdot 7 \cdot 11$	0,01
10^7	50000	$3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	0,150
10^8	52500	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	1,700
10^9	57500	$2^2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	19,5
10^{10}	85000	$2^3 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	226,2

Tabelle 15: Optimale pg-Laufzeiten Ultra-1, $10^4 - 10^{10}$

Intervallobergrenze	`MUL_3_5_THRESHOLD`	`SIEVELENLONGS`	Zeit
10^4	40000	$2^3 \cdot 5 \cdot 7 \cdot 11$	0,02
10^5	40000	$2^3 \cdot 5 \cdot 7 \cdot 11$	0,02
10^6	40000	$2^3 \cdot 5 \cdot 7 \cdot 11$	0,05
10^7	40000	$2^3 \cdot 5 \cdot 7 \cdot 11$	0,48
10^8	42500	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	5,62
10^9	57500	$2^2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	64,42
10^{10}	60000	$2^2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	796,22

Tabelle 16: Optimale pg-Laufzeiten SPARCstation-4, $10^4 - 10^{10}$

Intervallobergrenze	`MUL_3_5_THRESHOLD`	`SIEVELENLONGS`	Zeit
10^4	40000	$2^3 \cdot 5 \cdot 7 \cdot 11$	0,02
10^5	40000	$2^3 \cdot 5 \cdot 7 \cdot 11$	0,02
10^6	40000	$2^3 \cdot 5 \cdot 7 \cdot 11$	0,04
10^7	45000	$2^3 \cdot 5 \cdot 7 \cdot 11$	0,46
10^8	52500	$3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	5,4
10^9	52500	$3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	64,3
10^{10}	55000	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	861,97

Tabelle 17: Optimale pg-Laufzeiten PC Pentium 200, $10^4 - 10^{10}$

Die identischen Zeiten für 10^4 und 10^5 ergeben sich daraus, daß bereits ein Wert für `SIEVELENLONGS` von $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 = 9240$ ein Intervall der Länge 591360 repräsentiert und `pg` immer mindestens ein ganzes Segment siebt.

Die zweite Messung betrachtet die Intervalle $[10^n, 10^n + 10^9]$ für $n \in \{10, 11, 12, 13, 14, 15\}$. Diese Rechnung wurde nur auf der Ultra-1 durchgeführt. Optimal waren dabei für `MUL_3_5_THRESHOLD` 50000 sowie für `SIEVELENLONGS` $2^3 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$. In der dritten Spalte ist als Vergleichswert die Zeit für die Rechnung mit `MUL_3_5_THRESHOLD` $= 0$ angegeben:

n	Zeit `MUL_3_5_THRESHOLD` $= 50000$	Zeit `MUL_3_5_THRESHOLD` $= 0$
10	24,7	43,1
11	29,7	48,1
12	38,0	56,4
13	56,3	74,5
14	104,5	122,2
15	241,7	257,3

Tabelle 18: Optimale `pg`-Laufzeiten Ultra-1, $10^n - 10^n + 10^9$

Die deutliche Zunahme der Laufzeiten ist natürlich nicht verwunderlich, da die Anzahl der jeweils zu siebenden Primfaktoren in der Größenordnung $O(\sqrt{n}/\log\sqrt{n})$ wächst. Der Einfluß des Parameters `MUL_3_5_THRESHOLD` ist beachtlich und macht im ersten Intervall eine Ersparnis von immerhin 35% aus. Die fallende Tendenz liegt an der Tatsache, daß immer größere Primfaktoren in Betracht kommen, für die die Vermeidung der Mehrfachmarkierungen keinen Sinn mehr macht und nicht stattfindet.

3.5 Vergleich mit anderen Sieben

In [Dun96] und [Sor98] werden die Laufzeiten von Implementierungen verschiedener Siebalgorithmen miteinander verglichen.

Das beste Ergebnis erzielte dabei ein Verfahren, das nun beschrieben werden soll.

3.5.1 Pritchard's Linear Segmented Wheel Sieve

Das hier beschriebene Verfahren geht auf Paul Pritchard [Pri83] zurück. Es handelt sich dabei prinzipiell um eine segmentierte Version des Eratosthenes-Siebes. Eine grundlegende Rolle spielt dabei das Konzept der *Wheels*.

Definition 3.16 Es seien $W_k(m)$ und W_k definiert durch

$$W_k(m) = \{j \leq m : ggT(j, M_k) = 1\},$$

$$W_k = W_k(M_k)$$

W_k heißt auch k-tes $Wheel$, $W_k(y)$ k-tes $Wheel$ erweitert auf y.

Bemerkung. Es bezeichne $\varphi(n) = |\{1 \leq k \leq n : ggT(k, n) = 1\}|$ die Eulersche Funktion. Dann ist

$$|W_k| = \varphi(M_k) = O\left(\frac{M_k}{\log\log M_k}\right)$$

sowie

$$|W_k(m)| = O\left(\frac{m}{\log\log M_k}\right)$$

(siehe z.B. [Rib96])

Das Prinzip des Siebverfahrens ist nun das folgende:

1. Initialisiere jedes Segment, so daß zunächst alle Zahlen als zerlegbar gelten
2. Nehme diejenigen Zahlen als Primzahlkandidaten auf, die relativ prim zu den ersten k Primzahlen sind
3. Streiche nun diejenigen wieder heraus, die einen Primfaktor p mit $p_k < p \leq p_{\pi(\sqrt{n})}$ besitzen

Der folgende Algorithmus siebt ein Teilsegment (siehe auch [Sor98] sowie [Pri83]). Auf die explizite Formulierung des steuernden Algorithmus zur Segmentierung analog 3.11 wird hier verzichtet.

Dabei sei $W_k[x] = min\{y : y > x \wedge ggT(y, M_k) = 1\}$.

Algorithmus 3.18 *sieve1segp*

1: $primes \leftarrow input()$ { Eingabe der Primzahlen $\leq \sqrt{n}$ }
2: $pisqrtx \leftarrow input()$ { Eingabe von $\pi(\sqrt{n})$ }
3: $firstMin1 \leftarrow input()$ { Erste Zahl des Segments $-$ 1 }
4: $last \leftarrow input()$ { Letzte Zahl des Segments }
5: $k \leftarrow input()$ { Eingabe von k }
6: $mk \leftarrow input()$ { Eingabe von M_k }
7: $x \leftarrow firstMin1 + W_k[firstMin1\%mk]$ { Erster Zahl x mit $ggT(x, M_k) = 1$ }
8: **while** $x \leq last$ **do** { Nehme Kandidaten auf }
9: $\quad setBit(segment, x - firstMin1)$
10: $\quad x \leftarrow x + W_k[x\%mk]$ { Setze x auf nächste zu M_k teilerfremde Zahl }
11: **end while**
12: $i \leftarrow k + 1$
13: **while** $i \leq pisqrtx$ **do** { Durchlaufe Primzahlen $\leq \sqrt{n}$ }
14: $\quad p \leftarrow primes[i]$ { $p = p_i$ }
15: $\quad firstMin1p \leftarrow firstMin1 \div p$
16: $\quad factor \leftarrow firstMin1p + W_k[firstMin1p\%mk]$ { Erster Faktor von p_i }
17: $\quad lastp \leftarrow last \div p$
18: $\quad$ **while** $factor \leq lastp$ **do** { Streiche Zerlegbare }
19: $\quad\quad unsetBit(segment, p \times factor - firstMin1)$ { Setze Kandidaten zurück }
20: $\quad\quad factor \leftarrow factor + W_k[factor\%mk]$ { Ermittle nächsten Faktor }
21: $\quad$ **end while**
22: $\quad i \leftarrow i + 1$
23: **end while**
24: **return** *segment*

Satz 3.17 Es sei $l = last - firstMin1$ die Länge eines Segments. Algorithmus 3.18 siebt die Primzahlen aus einem Intervall $[firstMin1 + 1, last]$ in

$$\begin{aligned} t_{ADL}(n,l,k) &= (40B + D(B) + S(M_k/(2B))) \cdot \varphi(M_k) \cdot l/M_k + \\ &\quad +(30B + 3 \cdot D(B) + S(\pi(\sqrt{n})/2) + S(M_k/(2B))) \cdot (\pi(\sqrt{n}) - k) + \\ &\quad +\frac{\varphi(M_k)}{M_k} \cdot (40B + M(B) + D(B) + S(M_k/(2B))) \cdot N^*(p_k, \sqrt{n}, l) \end{aligned}$$

Bitoperationen unter Verwendung von $s_{ADL}(n,l,k) = \pi(\sqrt{n})B + l/2 + \varphi(M_k)B + O(1)$ Bits Speicher aus.

Beweis. Die erste **while**-Schleife in den Zeilen 8 – 11 nimmt diejenigen Zahlen als Primzahlkandidaten in das Feld auf, die relativ prim zu den ersten k Primzahlen sind. Es gibt $\varphi(M_k)$ zu M_k teilerfremde Zahlen, d.h. insgesamt werden $\varphi(M_k) \cdot l/M_k$ Schritte durchlaufen. Dabei wird für jedes p zunächst in Zeile 16 der erste Faktor zu einer durch p zerlegbaren Zahl im Segment bestimmt. Die Schleife 13 – 23 besteht aus $\pi(\sqrt{n}) - k$ Schritten. Schließlich werden in den Zeilen 18 – 21 alle Vielfachen der p im Segment gestrichen. Die Funktion *unsetBit* sei dabei analog *setBit* definiert. Diese Schleife benötigt für jedes p $\varphi(M_k) \cdot \left\lfloor \frac{l}{p} \right\rfloor / M_k$ Durchläufe. Die Konstruktion des Feldes W_k muß nur einmal durchgeführt werden und kann in $O(M_k)$ Schritten erfolgen (siehe [Dun96]). Zur Raumkomplexität: Das Feld W_k benötigt $M_k B$ Bits Speicher, die Primzahlliste bis $\sqrt{n}$ wiederum $\pi(\sqrt{n})B$ Bits. Hinzu kommt der Speicher für ein Segment, bestehend aus l Bits. □

Es bezeichne nun Algorithmus 3.19 (Pritchard's Linear Segmented Wheel Sieve) denjenigen Algorithmus, der unter Verwendung von 3.18 segmentweise aus einem Intervall $[1, n]$ die Primzahlen aussiebt.

Satz 3.18 Algorithmus 3.19 siebt die Primzahlen aus dem Intervall $[1, n]$ in

$$t_{ADL}(n) \quad = \quad O(n \log n)$$

Bitoperationen unter Verwendung von $s_{ADL}(n) = O(\sqrt{n})$ Bits Speicher aus.

Beweis. Durch Festlegung von $M_k = O(\sqrt{n})$ wird $p_k = O(log\sqrt{n})$. Mit 3.5.1 wird nun

$$S(M_k/(2B)) \cdot \varphi(M_k) \cdot l/M_k = O\left(\frac{l \log n}{\log \log n}\right),$$

$$S(M_k/(2B)) \cdot (\pi(\sqrt{n}) - k) = O(\sqrt{n})$$

sowie

$$\frac{\varphi(M_k)}{M_k} \cdot (40B + M(B) + D(B) + S(M_k/(2B))) \cdot N^*(p_k, \sqrt{n}, l) = O\left(\frac{l \log \log n \log n}{\log \log n}\right)$$

$$= O(l \log n)$$

Durch die Wahl von $l = O(\sqrt{n})$ ergibt sich für die Zeit zum Sieben eines Segments die Größenordnung $O(\sqrt{n} \log n)$ und daher insgesamt für 3.19 $O(n \log n)$. □

Bemerkung. Es ist also möglich, einen Faktor $1/\log \log n$ zu gewinnen. In den meisten Berechnungsmodellen wird $S(n) = O(1)$, woraus eine wirklich lineare Zeitkomplexität folgt. Allerdings zeigen sich in der Praxis doch entscheidende Nachteile, was insbesondere auf die notwendigen Divisionen, Multiplikationen und die zusätzlichen Speicherzugriffe zurückzuführen ist. Abbildung 15 zeigt einen Vergleich der ADL-Komplexitäten von 3.19 und 3.14 für die Ultra-1, die sowohl vom Speicher als auch von den anderen Operationen billiger als die SPARCstation-4 ist und sich daher eher vorteilhaft auf 3.19 auswirkt.

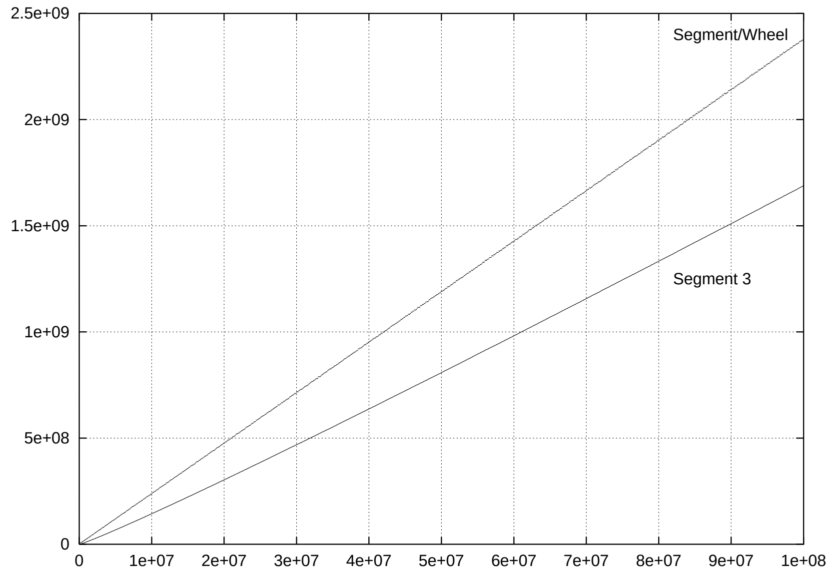


Abbildung 15: ADL-Zeitkomplexitäten 3.14 und 3.19 (Ultra1)

Tabelle 19 zeigt einen Vergleich der Laufzeitmessungen von `pg` mit den Ergebnissen der Implementierung von 3.19 in [Sor98]. Die dortigen Messungen fanden ebenfalls auf einem Pentium 200 PC unter Linux 2.0 statt, die Kompilation wurde gleichermaßen mit dem GNU-C-Compiler `gcc` in der Version 2.7.2.1 und der Optimierungsstufe `-O` durchgeführt. Der dort zur Verfügung stehende Hauptspeicher war allerdings sechsmal größer, was jedoch keinen oder höchstens einen positiven Einfluß auf die dortigen Messungen gehabt haben sollte.

Intervallobergrenze	Algorithmus 2.5 [Sor98]	`pg`
10^4	0,002	0,02
10^5	0,02	0,02
10^6	0,21	0,04
10^7	2,27	0,46
10^8	23,61	5,4
10^9	242	64,3

Tabelle 19: Laufzeitvergleich (Angaben in CPU-Sekunden)

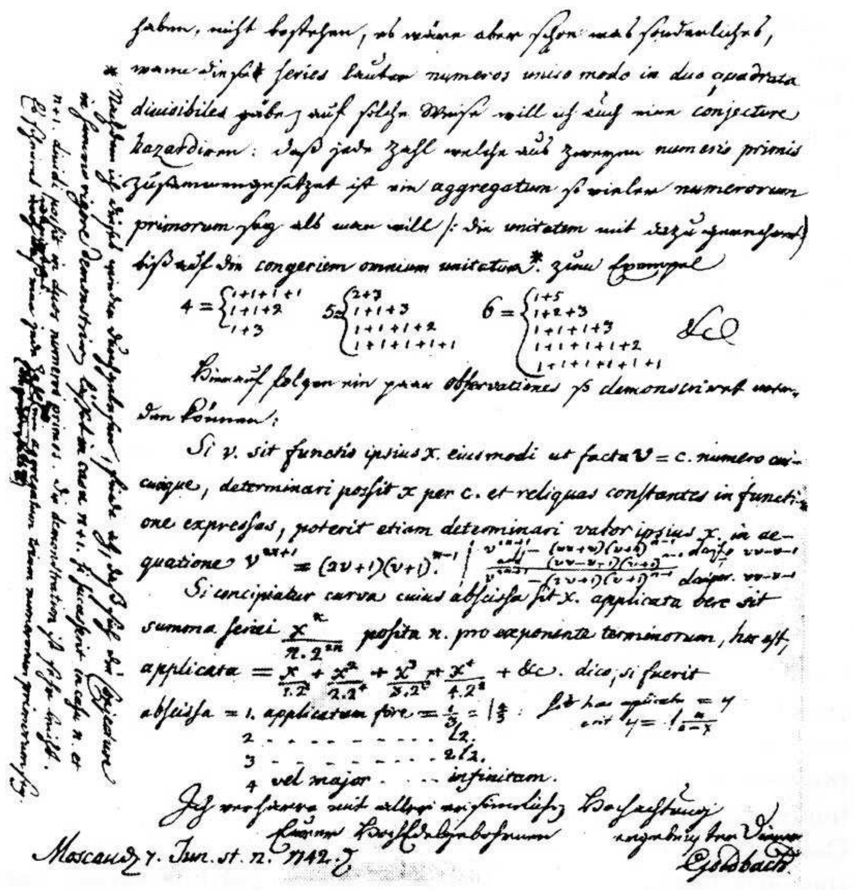
Es scheint keinen offensichtlichen Weg zu geben, die zusätzlichen Zugriffe von 3.19 zu vermeiden. Sorenson schlug in [Sor98] eine veränderte Version von 3.19 vor, die in den dort zu findenden Messungen 3.19 zwar leicht unterliegt. Es wird jedoch auf zusätzliche Optimierungsmöglichkeiten hingewiesen, die diese Situation verändern könnten.

Der Primzahlgenerator `pg` kommt im folgenden zur Anwendung.

4 Verifikation der Goldbachschen Vermutung

4.1 Einführung

Ursächlicher Gegenstand dieses Kapitels ist der folgende Brief, den *Christian Goldbach* am 7. Juni 1742 an *Leonhard Euler* schrieb:


haben, nicht bestehen, ob wäre aber schon mal sonderlich, wenn dieselbe series lauter numeros unico modo in duo quadrata divisibiles geben, auf solche Weise will ich auch eine conjecture hazardiren: daß jede Zahl welche aus zweyen numeris primis zusammengesetzt ist ein aggregatum so vieler numerorum primorum sey als man will (die unitatem mit dazu gerechnet) biß auf die congeriem omnium unitatum.* zum Exempel

$4 = \begin{cases}1+1+1+1\\1+1+2\\1+3\end{cases}$ $5 = \begin{cases}2+3\\1+1+3\\1+1+1+2\\1+1+1+1+1\end{cases}$ $6 = \begin{cases}1+5\\1+2+3\\1+1+1+3\\1+1+1+1+2\\1+1+1+1+1+1\end{cases}$ &c.

Hierauf folgen ein paar observationes so demonstrirt werden können:

Si v. sit functio ipsius x. eiusmodi ut facta v = c. numero cuicunque, determinari possit x per c. et reliquas constantes in functione expressas, poterit etiam determinari valor ipsius x. in aequatione ...

Si concipiatur curva cuius abscissa sit x. applicata vero sit summa seriei $\frac{x^n}{n.2^n}$ posita n. pro exponente terminorum, hoc est, applicata $= \frac{x}{1.2} + \frac{x^2}{2.2^2} + \frac{x^3}{3.2^3} + \frac{x^4}{4.2^4} +$ &c. dico, si fuerit abscissa = 1. applicatam fore $= \frac{1}{2}$ ...

2 . . . . l2.

3 . . . . 2.l2.

4 vel major . . . infinitam.

Ich verharre mit aller ersinnlichen Hochachtung

Ew. Hochedelgebohrnen ergebenster Diener

Goldbach

Moscau d. 7. Jun. st. n. 1742.

Abbildung 16: Goldbachs Brief vom 7. Juli 1742

Goldbach vermutet darin, daß jede Zahl, die als Summe zweier Primzahlen darstellbar ist, sogar die Summe „beliebig vieler Primzahlen“ sei. Dabei wird auch die 1 als Primzahl angesehen. Die Bemerkung am Rand enthält die eigentliche *Goldbachsche Vermutung.* Dort ist zunächst angemerkt, daß sich die obige Aussage für $n + 1$ beweisen läßt, falls sie für n gilt und sich $n + 1$ als Summe zweier Primzahlen darstellen läßt. Es heißt dann weiter:

Es scheinet wenigstens, daß eine jede Zahl, die größer ist als 2, ein aggregatum trium numerorum primorum sey.

Goldbach gibt die drei Beispiele 4, 5 und 6 an. Weitere Versuche müssen ihn dann schließlich zu seiner berühmten Vermutung geführt haben.

Euler antwortet am 30. Juni, daß ihm Goldbach schon früher einmal die Vermutung mitgeteilt habe, daß sich jede *gerade* Zahl als Summe *zweier* Primzahlen darstellen läßt. Er zeigt dann, daß sich damit die Aussage beliebig vieler Darstellungen beweisen läßt (siehe 4.2).

Weiter heißt es bei Euler:

Daß aber ein jeder numerus par eine summa duorum primorum sey, halte ich für ein ganz gewisses theorema, ungeachtet ich dasselbe nicht demonstriren kann.

1912 äußert Edmund Landau, daß es sich bei der Goldbachschen Vermutung um ein Problem handele, das *beim gegenwärtigen Stande der Wissenschaft unangreifbar* sei. Godefrey Harold Hardy schätzt die Lösung des Goldbachschen Problems als *probably as difficult as any of the unsolved problems in mathematics.* Die Goldbachsche Vermutung konnte bis heute nicht bewiesen werden.

Bemerkung. In heutigen Arbeiten zum Thema findet man fast durchweg Ungenauigkeiten oder Verwechselungen. Meistens wird behauptet, Goldbach habe in seinem Brief vom 7. Juni 1742 vermutet, daß sich jede gerade Zahl als Summe zweier Primzahlen darstellen ließe. Dies ist nicht der Fall. Allerdings läßt die Eulersche Bemerkung, daß Goldbach ihm genau dies bereits vorher einmal mitgeteilt habe, darauf schließen, daß Goldbach es tatsächlich früher schon vermutet hatte. Ein Brief solchen Inhalts ist allerdings nicht bekannt und wahrscheinlich nicht mehr vorhanden. Es heißt bei Euler auch nur, daß (Goldbach) *vormals mit mir communicirt haben*, was wohl auch eine mündliche Kommunikation nicht prinzipiell ausschließt und damit das Fehlen einer Aufzeichnung erklären könnte.

Auch wird häufig fälschlicherweise zitiert, Euler habe auf Goldbachs Vermutung, daß sich jede Zahl als Summe dreier Primzahlen darstellen läßt, die Aussage von Satz 4.1 des folgenden Abschnitts gezeigt, was nicht stimmt.

Darüber hinaus wird in den meisten Zitaten vergessen, daß Goldbach die 1 als Primzahl betrachtete, was heute unüblich ist. Die heute als *Goldbachsche Vermutung* bezeichnete Aussage ist, daß sich jede gerade Zahl größer oder gleich 4 als Summe zweier Primzahlen ($\neq 1$) darstellen läßt. Dies ist nicht etwa gleichwertig zur ursprünglichen Vermutung, sondern impliziert sie.

Bereits vor Goldbach hatte Descartes bemerkt, daß sich jede gerade Zahl als Summe von ein, zwei oder drei Primzahlen darstellen läßt. Dies wurde jedoch erstmals 1908 veröffentlicht.

In diesem Kapitel wird eine Verifikation der Vermutung bis $4 \cdot 10^{14}$ beschrieben. Im nächsten Abschnitt werden zunächst ein kurzer, mathematischer Hintergrund gegeben sowie bekannte Teilergebnisse erwähnt. Es folgt in Abschnitt 4.4 ein einfacher Algorithmus zur Verifikation. Danach werden die zwei wesentlichen, heute bekannten Vorgehensweisen vorgestellt sowie deren Vor- und Nachteile beschrieben.

In Abschnitt 4.4.2 wird der zweite der beiden Algorithmen weiterentwickelt und durch einige Optimierungen wesentlich verbessert. Die Implementierung des Ergebnisses dieser Optimierung wurde auf mehrere Rechner verteilt, die Verteilung selbst wird dabei in Abschnitt 4.5 erklärt. Die Rechnung führte schließlich zum heute höchsten Wert, für den die Goldbachsche Vermutung als zutreffend bekannt ist. In Abschnitt 4.5.1 wird eine Laufzeitanalyse vorgenommen, in 4.6 werden weitere Ergebnisse der Rechnung gezeigt.

4.2 Mathematischer Hintergrund

Euler zeigt zunächst, daß sich die von Goldbach in seinem Brief geäußerte Aussage zur Darstellbarkeit als Summe vieler Primsummanden leicht nachweisen läßt, falls gilt, daß jede gerade Zahl Summe zweier Primzahlen (wieder inklusive der 1) ist. Denn falls $2n = p + q$ sich immer finden läßt, gibt es auch p' und q' mit $2n - 2 = p' + q'$ und damit ist $2n$ auch Summe der drei Primzahlen $p' + q' + 2$. Im ungeraden Falle $2n + 1$ ist $2n$ gerade und besitzt daher nach Voraussetzung eine Darstellung $p + q$. Also ist $2n + 1 = p + q + 1$ und damit wieder als Summe dreier Primzahlen dargestellt. Der Rest folgt mit Induktion.

Unabhängig davon, ob man die 1 nun als Primzahl zählt oder nicht, gilt:

Satz 4.1 Die beiden folgenden Aussagen sind äquivalent:

(G_3') *Jede natürliche Zahl $n > 5$ ist Summe dreier Primzahlen.*
(G_2) *Jede gerade Zahl $2n \geq 4$ ist Summe zweier Primzahlen.*

Beweis. „$\Longrightarrow$“: Es sei $n \geq 2$ und $2n + 2 = p_1 + p_2 + p_3$. Dann muß mindestens eine der drei $p_i = 2$ sein, sonst wäre die Summe ungerade. Damit ist (z.B.) $2n = p_1 + p_2$ und $2n \geq 4$.
„$\Longleftarrow$“: Es sei $n \geq 3$ und $2n-2 = p_1+p_2$. Dann ist $2n = p_1+p_2+2$ und $2n+1 = p_1+p_2+3$. □

Im folgenden wird die 1 nicht als Primzahl betrachtet und die *Goldbachsche Vermutung* mit Aussage (G_2) identifiziert. Eine ähnliche, schwächere Vermutung ist die folgende:

(G_3) *Jede ungerade Zahl $2n + 1 \geq 7$ ist Summe dreier Primzahlen.*

Satz 4.2 $(G_2) \implies (G_3)$

Beweis. Es sei $n \geq 3$ und wegen (G_2) gelte $2n-2 = p+q$. Dann ist $2n+1 = p+q+3$. □

(G_3) wird auch als *ternäre Goldbachsche Vermutung* bezeichnet (G_2) auch als *binäre*.

(G_3) wurde von *Vinogradov* 1937 [Vin37] für alle $n \geq n_0$ bewiesen. Er bewies sogar eine asymptotische Aussage zur Anzahl der Zerlegungen (siehe Kapitel 5). 1956 zeigte *Borodzkin* [Bor56], daß $n_0 < 3^{3^{15}} \approx 10^{6900000}$. n_0 wurde 1989 in [Che89] auf 10^{43000} reduziert. Erst 1997 konnte unter der Annahme der *verallgemeinerten Riemannschen Vermutung* (*GRH*, siehe z.B. [Bac96]) n_0 auf 10^{20} verbessert werden [Zin97]. In [Sao98] wurde (G_3) bis 10^{20} experimentell verifiziert und wäre damit unter der Voraussetzung der Richtigkeit von *GRH* bewiesen.

Was (G_2) angeht, so ist die folgende Aussage, die *Jing-Run Chen* im Jahre 1966 bewies, wohl als bis heute bestes Resultat zu bezeichnen:

Jede genügend große gerade Zahl $2n$ ist Summe zweier Primzahlen oder Summe einer Primzahl und eines Produkts zweier Primzahlen.

In [Mon75] konnte gezeigt werden, daß es Konstanten $k, \delta > 0$ gibt, so daß die Anzahl der $n \leq x$, die sich *nicht* als Summe zweier Primzahlen darstellen lassen, kleiner als $k \cdot x^{1-\delta}$ ist.

Eine sehr gute Einführung zum Thema wird in [Yua84] gegeben, wo eine Sammlung verschiedener, historischer Arbeiten – teilweise aus dem Chinesischen und Russischen ins Englische übersetzt – zu finden sind.

4.3 Historische Berechnungen

In der Vergangenheit wurden häufiger Verifikationen der Goldbachschen Vermutung vorgenommen. Tabelle 20 zeigt nur solche, die den jeweils vorher abgedeckten Bereich erweiterten.

Die in der Tabelle zuletzt aufgeführte Rechnung lief dabei, zumindest teilweise und ohne Wissen voneinander, parallel zu der hier in Abschnitt 4.5 beschriebenen.

Name	Jahr	$2n \leq$
A. Desboves	1855	10000
N. Pipping	1940	100000
M.K. Shen	1964	$3,3 \cdot 10^7$
M.L. Stein, P.R. Stein	1965	10^8
A. Granville, J.v.d. Lune, H.J.J. te Riele	1989	$2 \cdot 10^{10}$
M.K. Sinisalo	1993	$4 \cdot 10^{11}$
J.M. Deshouillers, H.J.J. te Riele, Y. Saouter	1998	10^{14} sowie $2n \in [10^{5i}, 10^{5i} + 10^8], i = 20, \ldots, 30$

Tabelle 20: Historische Berechnungen

4.4 Algorithmen zur Verifikation

Die heute bekannten Verfahren zur Verifikation der Goldbachschen Vermutung folgen demselben Grundprinzip: Es sei $N = \{2n : n_1 \leq 2n \leq n_2\}$ das zu prüfende Intervall. Dann gilt es, möglichst minimale Mengen von Primzahlen P_1 und P_2 zu finden, so daß

$$N \subseteq P_1 + P_2 = \{p_1 + p_2 : p_1 \in P_1, p_2 \in P_2\}.$$

Dabei muß natürlich in Betracht gezogen werden, daß solche Mengen möglicherweise nicht existieren.

Es soll nun zunächst ein einfaches Verfahren vorgestellt werden, bei dem

$$P_1 = P_2 = \{p : 2 < p \leq n_2\}$$

und $n_1 = 6$, $n_2 = n$ gewählt wird. Dadurch sind sicher alle möglichen Kombinationen von Summanden p_1, p_2 mit $p_1 + p_2 \leq n_2 = n$ abgedeckt.

Algorithmus 4.1 verifiziert die Goldbachsche Vermutung für das Intervall $[6, n]$. Dabei sei wie auch im folgenden gefordert, daß die Segmentlänge die Intervallobergrenze teile.

Algorithmus 4.1 *G2verify0*

```
n ← input()   { Eingabe der Obergrenze }
gaps ← pdiff(sqrt(n))   { Erzeugung des Lückenfeldes }
P1 ← pg_sieve(gaps, 0, n)   { Erzeuge P1 (und P2) mit pg_sieve }
i ← 1
while i < (n >> 2) do   { Durchlaufe ungerade Primzahlen bis n/2 }
  if getBit(P1, i) = 0 then   { Falls 2i + 1 prim }
    j ← i
    while j < (n − (i << 1) − 1) >> 1 do   { Durchlaufe Primzahlen bis n − 2i − 1 }
      if getBit(P1, j) = 0 then   { Falls 2j + 1 prim }
        setBit(g2, (i + j))   { Markiere Summe in g2 }
      end if
      j ← j + 1
    end while
  end if
  i ← i + 1
end while
for i ← 2 to n >> 1 do   { Prüfe, ob alle Bits von g2 gesetzt sind }
  if getBit(g2, i) = 0 then   { Keine Zerlegung gefunden }
    return false
  end if
end for
return true
```

Algorithmus 4.1 ist zwar korrekt, aber zu aufwendig. Es werden alle möglichen Kombinationen von Summanden durchlaufen, was sich als nicht notwendig herausstellen wird. Darüber hinaus stößt 4.1 durch fehlende Segmentierung schnell an praktische Grenzen. Es werden nun zwei prinzipiell bessere Möglichkeiten zur Wahl der Mengen P_1 und P_2 vorgestellt. Bevor diese detailliert beschrieben werden, zunächst die Festlegung einiger Begriffe:

Definition 4.3 Es seien p_1, p_2 zwei Primzahlen. Dann heißt das Paar (p_1, p_2) mit $2n = p_1 + p_2$ *Goldbach-Partition* der Zahl $2n$.

Definition 4.4

1. Die *minimale Goldbach-Partition* einer geraden Zahl $2n$ ist das Paar (p_1, p_2), $p_1 \leq p_2$, derart daß für alle primen $q < p_1$ die Zahlen $2n - q$ zerlegbar sind.
2. p_1 von 1. sei im folgenden durch $p(2n)$ bezeichnet.
3. Es bezeichne p^* die auf alle $x \in \mathbb{R}$ erweiterte Funktion, die die Maxima von p beschreibt, also $p^*(x) = \max\{p(2n) : 2n \leq x\}$.

Die wesentlichen Punkte, die nun ausgenutzt werden, sind einerseits das zu beobachtende, sehr langsame Wachstum von p^*. Es hat den Anschein, als besäße jede gerade Zahl $2n$

bereits eine minimale Partition, deren kleinerer Summand $p(2n)$ gegenüber n sehr klein bleibt. Man vermutet, daß gilt: $p^*(x) = O(\log^2 x \log\log x)$ (siehe dazu auch [Gra89] sowie Abschnitt 4.6). Ebenso scheint auch die Anzahl der verschiedenen Partitionen einer Zahl $2n$ an sich recht schnell zu wachsen, so daß auch eine Suche nach nicht-minimalen Partitionen schnell zum Erfolg führt (siehe auch Kapitel 5).

Daraus ergeben sich zwei günstige Möglichkeiten für die Wahl der Mengen P_1 und P_2. Abbildung 17 zeigt diese zunächst graphisch. Dabei bestehe dann jeweils P_1 aus dem gesamten linken, grauen Bereich und P_2 aus dem gesamten grauen Bereich rechts.

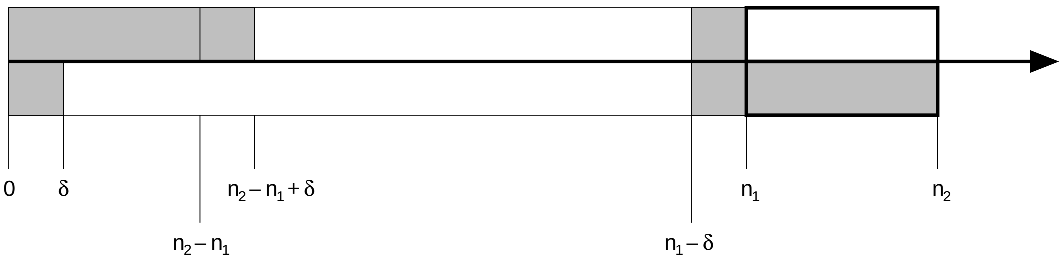


Abbildung 17: Wahl der Mengen P_1 und P_2

Die beiden Verfahren werden nun im einzelnen beschrieben.

4.4.1 Verwendung von Pseudoprimzahltests

Das hier vorgestellte Verfahren kam erstmals in [She64] zur Anwendung.

Es wird $P_1 = \{p : p \in [1, n_2 - n_1 + \delta]\}$ und $P_2 = \{p : p \in [n_1 - \delta, n_1]\}$ gewählt.

Das Prinzip ist nun das folgende:

Für jede gerade Zahl $2n \in N$ werden solange die Differenzen $2n - p_2$ mit $p_2 \in P_2$ gebildet, bis $2n - p_2$ prim ist. Wegen $n_1 - \delta \le p_2 \le n_1$ und $n_1 \le 2n \le n_2$ ist $2n - p_2 \in [1, n_2 - n_1 + \delta]$. Die relativ große Primzahlmenge P_1 muß dabei für alle Segmente aus dem Gesamtintervall nur einmal erzeugt werden, P_2 für jedes Testsegment neu. Da nach obiger Bemerkung zu erwarten ist, daß sich recht schnell eine Partition findet, muß P_2 wahrscheinlich nicht all zu mächtig gewählt sein.

Das Problem ist nun, die Menge P_2 zu erzeugen. Da n_1 normalerweise recht groß ist, werden zur effizienten Generierung der Primzahlen $p_2 \in [n_1 - \delta, n_1]$ *Strong Pseudoprime Tests* hinzugezogen.

Definition 4.5 Eine Zahl $n = 2^h \cdot d$, d ungerade und $h > 0$ heißt *strong pseudoprime* zur Basis $b \iff b^d \equiv 1 \mod n$ oder $b^{2^k} \equiv -1 \mod n$ für ein $0 \leq k < h$.

Es seien $p_1, p_2, \ldots, p_k$ die ersten k Primzahlen. Dann bezeichne ψ_k diejenige maximale Zahl, für die gilt: $\forall\ n < \psi_k$: n ist prim $\iff$ n ist *strong pseudoprime* zu den Basen $p_1, \ldots, p_k$.

Satz 4.6 $\psi_8 = 341550071728321$.

Beweis. Siehe [Jae93]. □

Die Bedeutung von Satz 4.6 ist, daß 8 Pseudoprimzahltests reichen, um die Primalität einer Zahl $n < 341550071728321$ nachzuweisen.

Bemerkung. ψ_j für $j > 8$ ist nicht bekannt.

Eine genauere Diskussion neben einer Implementierung des Tests findet sich z.B. in [Rie94].

Es folgt die ADL-Beschreibung des ersten effizienten Verfahrens zur Verifikation. Zunächst wird der steuernde Algorithmus zur Segmentierung, dann die Verifikation der einzelnen Segmente beschrieben. Dabei erzeuge $genP_2$ die $\pi(n_1) - \pi(n_1 - \delta)$ größten Primzahlen unterhalb von n_1 und übergebe diese an P_2 als Feld der Primzahlen selbst. Die Grenzen n_1 und n_2 seien als gerade vorausgesetzt und es gelte wieder $l \mid n$.

Algorithmus 4.2 *G2verify1*

$n \leftarrow input()$ { Eingabe der Obergrenze }
$l \leftarrow input()$ { Eingabe der Segmentlänge }
$\delta \leftarrow input()$
if $G2verify0(l) = false$ **then** { Erstes Segment mit 4.1 verifizieren }
  return $false$
end if
$P_1 \leftarrow pg\_sieve(pdiff(sqrt(l + \delta)), 0, (l \gg 1))$ { Erzeugung von P_1 }
$n_{segs} \leftarrow n \div l$ { Bestimme Anzahl der Segmente }
for $i \leftarrow 1$ **to** $n_{segs} - 1$ **do** { Restliche Segmente }
  $P_2 \leftarrow genP_2(i \times l, \delta)$ { Generierung des Feldes P_2 }
  if $G2segment1(i, l, P_1, P_2, pi(P_2)) = false$ **then** { Ausnahme }
    return $false$
  end if
end for
return $true$

Der folgende Algorithmus verifiziert ein einzelnes Segment:

Algorithmus 4.3 *G2segment1*

```
1:  i ← input()   { Eingabe der Segmentnummer }
2:  l ← input()   { Eingabe der Segmentlänge }
3:  P_1 ← input()   { Eingabe der ersten Primzahlmenge }
4:  P_2 ← input()   { Eingabe der zweiten Primzahlmenge }
5:  π_max ← input()   { Eingabe der Anzahl zu durchlaufender Primzahlen }
6:  n_1 ← twon + l   { Berechnung der unteren Segmentgrenze }
7:  n_2 ← n_1 + l   { Berechnung der oberen Segmentgrenze }
8:  twon ← n_1
9:  while twon ≤ n_2 do   { Durchlaufe gerade Zahlen }
10:   for j ← 0 to π_max − 1 do   { Durchlaufe P_2 }
11:     if getBit(P_1, ((twon − P_2[j])) ≫ 1) = 0 then   { Partition gefunden }
12:       break   { Nächstes twon }
13:     end if
14:   end for
15:   if j = π_max then   { Ausnahmesituation }
16:     return false   { Keine Partition gefunden }
17:   end if
18:   twon ← twon + 2
19: end while
20: return true
```

Satz 4.7 Algorithmus 4.3 verifiziert die Goldbachsche Vermutung für das Intervall $[n_1, n_2]$. Falls der Wert *true* zurückgeliefert wird, ist das Intervall positiv verifiziert. Falls das Ergebnis *false* ist, gibt es eine gerade Zahl $2n \in [n_1, n_2]$, so daß für alle $p \in P_2$ die Zahlen $2n - p$ zerlegbar sind.

Beweis. Sämtliche geraden Zahlen $2n$ des Intervalls werden durchlaufen und dabei die Differenzen $2n - p$ mit $p \in P_2$ gebildet. Die Differenzen werden in Zeile 10 auf Primalität geprüft. Falls für alle $2n$ eine prime Differenz gefunden werden konnte, ist das Intervall verifiziert. Falls eine Zahl $2n$ nicht durch Primzahlen $p \in P_2$ zerlegt werden konnte, wird dies in Zeile 16 durch negativen Abbruch festgestellt. □

Es handelt sich also in dieser Form um einen nach [Des99] „diplomatischen“ Test. Allerdings kann Algorithmus 4.3 prinzipiell durch die Wahl der Mengen P_1 und P_2 wie in Algorithmus 4.3 in einen deterministischen Test überführt werden.

Algorithmus 4.3 hat einen wesentlichen Vorteil, allerdings auch entscheidende Nachteile gegenüber dem Verfahren des nächsten Abschnitts.

Der Vorteil ist die bessere Laufzeit, da für jedes Testsegment nur eine relativ kleine Anzahl von Primzahlen generiert werden muß, wohingegen der im nächsten Abschnitt vorgestellte Algorithmus das Gesamtintervall komplett durchsiebt.

Der erste Nachteil ist die Tatsache, daß 4.3 im allgemeinen nicht die Werte für $p(2n)$ liefert. Es handelt sich also um einen reinen Verifikationsalgorithmus ohne die Möglichkeit zur Gewinnung weiterer Daten zum Problem. Der zweite Nachteil ist die Abhängigkeit von der Kenntnis des ψ-Wertes bis zur Obergrenze des zu prüfenden Intervalls. Zum Zeitpunkt der Erstellung dieser Arbeit war nur der Wert von ψ_8 bekannt. Damit ist Algorithmus 4.3 also auf das Intervall $[1, 341550071728321] \subset [1, 4 \cdot 10^{14}]$ beschränkt.

Aus den beiden genannten Gründen wurde nicht Algorithmus 4.3 implementiert, sondern das im nächsten Abschnitt beschriebene Verfahren. Auf eine Laufzeitanalyse von Algorithmus 4.3 wird hier verzichtet. In [Des98] wird die zu erwartende Laufzeit betrachtet, eine genauere Analyse ist in [Des99] zu finden. Ein praktikabler Wert für die Mächtigkeit von P_2 (bei $\max(n_2) = 10^{14}$) wird dabei mit 430 angegeben.

Bemerkung. Eigentlich hängt Algorithmus 4.3 nur dann von der Kenntnis der ψ-Werte ab, wenn er Verwendung von Pseudoprimzahltests macht. Die Alternative wäre wiederum das Sieben der Mengen P_2, was allerdings den einzigen Vorteil, nämlich den der besseren Laufzeit, zunichte machen würde.

Es sei noch darauf hingewiesen, daß die Optimierung der Subtraktionen in 4.3 analog des nun vorzustellenden Verfahrens geschehen kann.

4.4.2 Komplettes Sieben aller Testsegmente

Im zweiten Verfahren wird $P_1 = \{p : p \in [1, \delta]\}$ und $P_2 = \{p : p \in [n_1 - \delta, n_2]\}$. Das Prinzip ist das folgende:

Für jede gerade Zahl $2n \in N$ werden solange die Differenzen $2n - p$ mit $p \in P_1$ gebildet, bis $2n - p_1$ prim ist. Wegen $2n \leq n_2$ und $p \leq \delta$ ist $2n - p \in [n_1 - \delta, n_2]$. Die Primzahlmenge P_1 muß dabei nur einmal erzeugt werden. P_2 wird jedesmal durch Sieben des gesamten Segments und Verwendung des Teilstücks $[n_1 - \delta, n_1]$ des letzten Segments gebildet. Da – wie sich noch herausstellen wird – p^* tatsächlich recht langsam wächst, muß P_1 nicht allzu groß gewählt werden. Es folgt eine erste ADL-Beschreibung des Verfahrens. Es sei dazu P_1 als (relativ kleines) Feld der Primzahlen kleiner gleich δ gegeben. Es gelte $B \mid l$ und $l \mid n$ wobei diese Forderung keine besondere Einschränkung darstellt. P_2 wird im folgenden als Bitfeld der Länge $(l + \delta)/2$ aufgefaßt. Hier nur der steuernde Algorithmus, die segmentweise Verifikation kann erneut von 4.3 vorgenommen werden, wobei sich nur beim Aufruf etwas ändert.

Algorithmus 4.4 *G2verify2*

```
n ← input()   { Eingabe der Obergrenze }
l ← input()   { Eingabe der Segmentlänge }
if G2verify0(l) = false then  { Erstes Segment mit 4.1 verifizieren }
  return false
end if
n_segs ← n ÷ l   { Bestimme Anzahl der Segmente }
δ ← log(n) × log(n) × log(log(n)) ≪ 1   { Ermittlung von δ }
overlap ← δ ≫ (b + 1)   { Anzahl der überlappenden Wörter }
for i ← 1 to n_segs − 1 do  { Restliche Segmente }
  for k ← 0 to overlap − 1 do  { Kopiere relevante Wörter vom letztem Sieb }
    P_2[k] ← P_2[(l ≫ (b + 1)) + k]
  end for
  P_2 ← pg_sieve'(gaps, i × l, l)   { Generierung des Feldes P_2 }
  if G2segment1'(i, l, P_2, P_1, pi(P_1)) = false then  { Ausnahme }
    return false
  end if
end for
return true
```

pg_sieve' unterscheidet sich von pg_sieve nur dadurch, daß das Sieben von P_2 bei Bit $\delta/2$ beginnt. $G2segment1'$ unterscheidet sich von Algorithmus 4.3 nur dadurch, daß in Zeile 11 hinter $P_2[j]$ noch n_1 abgezogen wird. Der Aufruf erfolgt nun mit vertauschten Parametern P_1 und P_2. Zur folgenden Analyse noch eine Bemerkung:

Es sei $p^*(x) = O(\log^2 x \log\log x)$ angenommen (für den betrachteten Bereich wird sich diese Annahme bestätigen). Die O-Konstante in dieser Abschätzung ist relativ klein, es wird sich zeigen, daß sie im betrachteten Intervall bei etwa 1,6 liegt. Im folgenden sei $\delta = 2 \cdot \log^2 n_2 \log\log n_2$ gewählt.

Lemma 4.8 Algorithmus 4.3 verifiziert die Goldbachsche Vermutung für alle geraden Zahlen eines Segments der Länge l in

$$\begin{aligned} t_{ADL}(l) &= \frac{l}{2} \cdot \big(18B + \pi(log^2 l \log\log l) \cdot \big(50B + S(\pi(log^2 l \log\log l)/2) + \\ &\quad +S(l/4B)\big)\big) \\ &= O(l \log^3 l \log\log l) \end{aligned}$$

Bitoperationen und $s_{ADL}(n) = O(1)$ Bits zusätzlichem Speicherplatz. Falls der Wert *true* zurückgeliefert wird, ist das Intervall positiv verifiziert. Falls das Ergebnis *false* ist, gibt es eine gerade Zahl $2n \in [i \cdot l + 1, (i + 1) \cdot l]$, so daß für alle $p \in P_1$ die Zahlen $2n - p$ zerlegbar sind.

Beweis. (Für den Aufruf durch 4.4) Es werden wiederum sämtliche Differenzen $2n-p$ mit $p \in P_1$ gebildet und auf Primalität geprüft. Die **while**-Schleife wird $l/2$ mal durchlaufen, die **for**-Schleife jeweils maximal $\pi(\delta)$ mal. Mit $\delta = O(\log^2 n_2 \log\log n_2)$ folgt schließlich die Aussage zum Zeitaufwand. Algorithmus 4.3 benötigt nur konstant viel zusätzlichen Speicher, da die Felder P_1 und P_2 jeweils übergeben werden. □

Satz 4.9 Algorithmus 4.4 verifiziert die Goldbachsche Vermutung für alle geraden Zahlen kleiner gleich n in

$$\begin{aligned} t_{ADL}(n,l) &= n/l \cdot (19B + O(l \log l \log\log n) + O(l \log^3 l \log\log l) + \\ &\quad + \frac{\log^2 l \log\log l}{B} \cdot (14B + 2 \cdot S(\log^2 l \log\log l/2B))) \\ &= O(n(\log l \log\log n + \log^3 l \log\log l)) \end{aligned}$$

Bitoperationen und $s_{ADL}(n,l) = l/2 + \log^2 l \log\log l + \pi(2 \cdot \log^2 l \log\log l)B + O(1)$ Bits Speicherplatz. Es gilt wieder obige Bemerkung zum Rückgabewert $false$.

Beweis. Das erste Segment wird mit Algorithmus 4.1 verifiziert, die restlichen jeweils durch Aufrufe von 4.3. Dabei werden die dort benötigten Primzahlen zwischen $i \cdot l$ und $i \cdot l - \delta$ jeweils vor dem Sieben in den Zeilen 10–12 kopiert. Entscheidend sind die Aufrufe von *pg_sieve* und *G2segment1* in den Zeilen 13 und 14. Diese erzeugen schließlich die beiden O-Terme. Die Speicherkomplexität ergibt sich aus den $(l + 2\delta)/2$ Bits für das Feld P_2 und die $\pi(\delta)B$ Bits für das Feld P_1. □

Die Wahl von z.B. $l = \sqrt{n}$ zeigt, daß der zweite Term (aus der eigentlichen Verifikation eines Segments) überwiegt, was zunächst etwas überrascht. Das entscheidende Problem von Algorithmus 4.4 ist die große Anzahl von Subtraktionen in Algorithmus 4.3. Dies wird im nächsten Schritt verbessert. Statt viele Subtraktionen einzelner Zahlen durchzuführen, wird das gesamte Feld P_2 um jeweils $(p+1)/2$ (mit $p \in P_1$) Bits verschoben, was einer Addition aller in P_2 repräsentierten Primzahlen um p entspricht. Die $(n_2 - n_1)/2$ Bits ab Bit $(\delta - p + 1)/2$ werden dann jeweils mit dem zuvor auf Null initialisierten Feld $g2$ (der Länge $(n_2 - n_1)/2$) durch logisches Oder verknüpft. Dabei sei nun ein 1-Bit im Sieb gleichbedeutend mit einer Primzahl und ein 0-Bit stehe für eine zerlegbare Zahl. Dies ist in der Implementierung von pg tatsächlich durch eine einzige Anweisung definierbar. Das Feld P_2 sei nun insgesamt $(l + \delta)/2$ Bits lang. Schließlich enthält $g2$ genau dort eine 0, wo keine Summe $p_1 + p_2$ entstand, d.h. es müssen zur positiven Verifikation alle Bits von $g2$ gleich 1 sein.

Abbildung 18 zeigt die Vorgehensweise am Beispiel der Addition von 3 und 5.

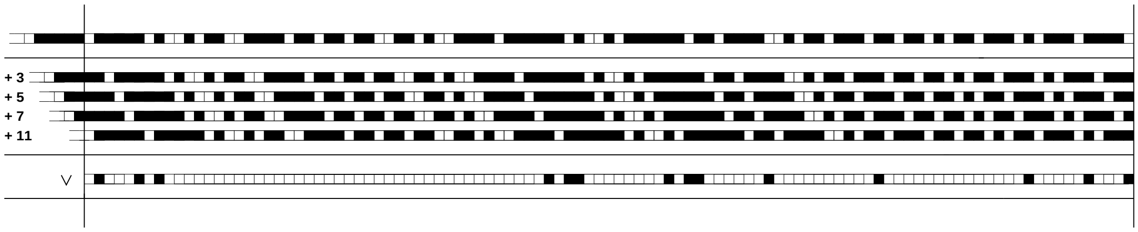


Abbildung 18: Ersetzung der Subtraktionen

Die *ADL*-Beschreibung ist in Algorithmus 4.5 gegeben:

Algorithmus 4.5 *G2segment2*

$l \leftarrow input()$ { Eingabe der Segmentlänge }
$P_1 \leftarrow input()$ { Eingabe der ersten Primzahlmenge }
$P_2 \leftarrow input()$ { Eingabe der zweiten Primzahlmenge }
$\pi_{max} \leftarrow input()$ { Eingabe der Anzahl der Primzahlen in $[1, \delta]$ }
for $j \leftarrow 0$ **to** $\pi_{max} - 1$ **do** { Durchlaufe P_1 }
  $shiftleft(P_2, (P_1[j] + 1) \gg 1)$
  for $k \leftarrow 0$ **to** $l \gg (b+1)$ **do** { Verknüpfe Ergebnis mit Feld $g2$ }
    $g2[k] \leftarrow g2[k] \vee P_2[(\delta \gg (b+1)) + k]$
  end for
end for
for $k \leftarrow 0$ **to** $l \gg (b+1)$ **do** { Durchlaufe $g2$, alle Bits müssen auf 1 gesetzt sein }
  if $g2[k] \neq \neg 0$ **then** { Teste auf Gleichheit mit 1111...1111 }
    return $false$ { Keine Partition gefunden }
  end if
end for
return $true$

Algorithmus 4.5 hat noch zwei wesentliche Nachteile.

Das Verschieben des Intervalls P_2 um k Bits durch *shiftleft* ist natürlich nicht durch eine Operation $P_2 \ll k$ zu erreichen, sondern muß durch teure Bitmaskierungen und Wortverschiebungen realisiert werden.

Darüber hinaus ist es vielleicht gar nicht notwendig und auch zu aufwendig, *alle* p aus P_1 zu durchlaufen und P_2 um $(p+1)/2$ Bits zu verschieben. Mit anderen Worten, die letzten Löcher im Feld $g2$ sind leichter zu stopfen, indem man wieder wie in Algorithmus 4.3 vorgeht und jeweils einzeln nach Partitionen sucht. Man muß abwägen, wieviele p zur Intervallverschiebung herangezogen werden.

Auf die Angabe einer Funktion zur Verschiebung des gesamten Feldes P_2 um k Bits wird verzichtet, da eine weitere Modifikation vorgenommen wird, die dies überflüssig macht.

Dazu folgende Vorüberlegung:

Für alle $(p+1)/2$, $p \in P_1$, die in derselben Restklasse $\mod B$ liegen, kann im Prinzip das gleiche, um $(p+1)/2 \mod B$ verschobene Segment P_2 benutzt werden. Es muß dazu nur bei Wort $\lfloor (p+1)/2B \rfloor$ mit der wortweisen, logischen Oder-Verknüpfung mit $g2$ begonnen werden. Diese Vorgehensweise reduziert die Anzahl der notwendigen, bitweisen Verschiebungen des gesamten Segments P_2 auf $B-1$.

Die folgenden Algorithmen beschreiben diese Modifikation. Darüber hinaus wird ein weiterer Parameter π_{lim} eingeführt, so daß nur noch für die ersten π_{lim} Primzahlen wortweise Verknüpfungen vom entsprechenden P_2 mit $g2$ vorgenommen werden.

Algorithmus 4.6 *G2segment3*

$l \leftarrow input()$ { Eingabe der Segmentlänge }
$\delta \leftarrow input()$ { Eingabe der Segmentlänge }
$P_1 \leftarrow input()$ { Eingabe der ersten Primzahlmenge }
$P_2 \leftarrow input()$ { Eingabe der zweiten Primzahlmenge }
$\pi_{lim} \leftarrow input()$ { Eingabe der maximalen Anzahl Verschiebungen }
$\pi_{max} \leftarrow input()$ { Eingabe der Anzahl der Primzahlen in $[1, \delta]$ }
$result \leftarrow true$ { Initialisierung des Rückgabewertes }
for $j \leftarrow 0$ **to** $B-1$ **do** { Verschiebe P_2 B mal um 1 Bit und verknüpfe mit $g2$ }
  $g2 \leftarrow P_2\_or\_g2(g2, P_1, P_2, \pi_{lim}, j, (l+\delta) \gg (b+1), \delta \gg (b+1))$ { Verknüpfung }
  $shiftleft1(P_2, (l+\delta) \gg (b+1))$ { Verschiebung }
end for
$result \leftarrow check0s(g2, P_1, P_2, p2n_{max}, \delta, \pi_{lim}, \delta)$ { Test der verbleibenden 0-Bits }
return $result$

Algorithmus 4.7 realisiert dabei die Verschiebung des Feldes P_2 um 1 Bit:

Algorithmus 4.7 *shiftleft1*

```
P_2 ← input()   { Eingabe des zu verschiebenden Feldes }
numwords ← input()   { Eingabe der Anzahl zu verschiebender Wörter }
overflowbit ← 1 ≪ (B − 1)   { Übertragsbit }
for j ← 0 to numwords − 1 do   { Durchlaufe Wörter von P_2 }
  thisword ← P_2[j]
  if thisword ∧ overflowbit then   { Möglicherweise Überlauf }
    overflow ← 1
  else
    overflow ← 0
  end if
  P_2[j] ← (thisword ≪ 1) + overflow   { Verschiebung mit gemerktem Überlauf }
end for
```

Die Verknüpfung der Felder P_2 und $g2$ ist durch 4.8 realisiert:

Algorithmus 4.8 *P_2_or_$g2$*

```
g2 ← input()
P_1 ← input()
P_2 ← input()
π_lim ← input()
j ← input()
numwords ← input()
overlap ← input()
for k ← 0 to π_lim − 1 do   { Durchlaufe P_1 }
  pmoves ← (P_1[k] + 1) ≫ 1   { Anzahl der zu verschiebenen Stellen für P_1[k] }
  if (pmoves ∧ (B − 1)) = i then   { Falls aktuelle Verschiebung (mod B) zutrifft }
    for m ← 0 to numwords − 1 do   { Verknüpfe Ergebnis mit Feld g2 }
      g2[m] ← g2[m] ∨ P_2[j + overlap − (pmoves ≫ b)]
    end for
  end if
end for
return g2
```

Algorithmus 4.9 übernimmt die Prüfung der letzten, noch nicht gefundenen Partitionen:

Algorithmus 4.9 *check0s*

```
g2 ← input()
P_1 ← input()
P_2 ← input()
p2n_max ← input()
δ ← input()
π_lim ← input()
for j ← 0 to l ≫ (b + 1) do  { Prüfe restliche 0-Bits in g2 wie in Algorithmus 4.3 }
  thisword ← g2[j]
  if thisword ≠ ¬0 then  { Falls nicht alle Bits gesetzt sind }
    for k ← 0 to B − 1 do  { Durchlaufe Bits des aktuellen Wortes }
      if thisword ∧ (1 ≪ k) = 0 then  { Falls Bit ungleich 1 }
        thisbit ← j ≪ b + k + δ
        for m ← π_lim to π_max − 1 do  { Durchlaufe restliche Primzahlen bis δ }
          if getBit(P_2, thisbit − ((P_1[m] + 1) ≫ 1) + B) = 1 then
            if P_1[m] > p2n_max then  { Neuer p*-Wert gefunden? }
              p2n_max ← P_1[m]  { Diesen merken }
            end if
            break  { Partition gefunden }
          end if
        end for
        if m = π_max − 1 then  { Keine Partition gefunden }
          return false
        end if
      end if
    end for
  end if
end for
return true
```

Schließlich muß der steuernde Algorithmus noch leicht modifiziert werden:

Das Kopieren der δ Bits des zuletzt bearbeiteten Segments muß ein Wort später beginnen, da um insgesamt B Bits verschoben worden ist. Eine Ausnahme bildet das erste Segment.

Darüber hinaus wird nun der maximale Wert von $p(2n)$ im Intervall ermittelt und schließlich ausgegeben. Dabei muß in einer Implementierung darauf geachtet werden, daß der neue, maximale Wert immer größer als die π_{lim}-te Primzahl ist, da p^* erst in 4.9 ermittelt wird.

Algorithmus 4.10 *G2verify3*

$n \leftarrow input()$ { Eingabe der Obergrenze }
$l \leftarrow input()$ { Eingabe der Segmentlänge }
$\pi_{lim} \leftarrow input()$
if $G2verify0(l) = false$ **then** { Falls erstes Segment negativ verifiziert }
  return $false$
end if
$n_{segs} \leftarrow n \div l$ { Bestimme Anzahl der Segmente }
$\delta \leftarrow log(n) \times log(n) \times log(log(n))$ { Ermittlung $\delta/2$ }
$overlap \leftarrow \delta \gg b$ { Anzahl der überlappenden Wörter }
for $k \leftarrow 0$ **to** overlap **do** { Kopiere relevante Wörter vom letztem Sieb }
  $P_2[k] \leftarrow P_2[(l \gg (b+1)) + k]$ { Wortweise kopieren }
end for
for $i \leftarrow 1$ **to** $n_{segs} - 1$ **do** { Restliche Segmente }
  for $k \leftarrow 0$ **to** overlap **do** { Kopiere relevante Wörter vom letztem Sieb }
    $P_2[k] \leftarrow P_2[(l \gg (b+1)) + k + 1]$ { +1, da P_2 um ein Wort verschoben ist }
  end for
  $P_2 \leftarrow pg\_sieve'(gaps, i \times l, \delta)$ { Generierung des Feldes P_2 }
  if $G2segment3(l, \delta, P_1, P_2, \pi_{lim}, pi(P_1)) = false$ **then** { Falls Segment negativ verifiziert }
    return $false$
  end if
end for
$output(p2n_{max})$
return $true$

Lemma 4.10 Algorithmus 4.7 verschiebt ein Segment um 1 Bit in

$$\begin{aligned} t_{ADL}(l,\delta) &= (l+\delta)/2B \cdot (33B + 2 \cdot S((l+\delta)/4B)) + O(1) \\ &= O((l+\delta)\log(l+\delta)) \end{aligned}$$

Bitoperationen und $s_{ADL}(l) = O(1)$ Bits zusätzlichem Speicherplatz.

Beweis. Auf jedes Wort wird einmal lesend und schreibend zugegriffen, Überläufe werden vermerkt und korrigiert. □

Lemma 4.11 Algorithmus 4.8 führt die Addition der ersten π_{lim} Primzahlen auf das Segment P_2 in

$$\begin{aligned} t_{ADL}(l,\pi_{lim},\delta) &= \pi_{lim} \cdot (32B + S(\pi(\delta)/2)) + \\ &\quad +\pi_{lim}/B \cdot (3B + (l+\delta)/2B \cdot (33B + 2 \cdot S((l+\delta)/4B)) + O(1) \\ &= O(\pi_{lim}(l+\delta)\log(l+\delta)) \end{aligned}$$

Bitoperationen und $s_{ADL}(l) = O(1)$ Bits zusätzlichem Speicherplatz durch.

Beweis. In Zeile 9 wird für jede möglicherweise zu addierende Primzahl zunächst die zu verschiebende Anzahl Bits modulo B berechnet. Falls die gerade verschobene Anzahl damit übereinstimmt, wird in 11 – 13 die Verknüpfung mit dem $g2$-Feld vorgenommen. Die Schleife in Zeile 8 wird π_{lim}-mal ausgeführt, die Zeilen 11 – 13 π_{lim}/B-mal. □

Lemma 4.12 Algorithmus 4.9 sucht die noch fehlenden Partitionen durch Subtraktion der restlichen Primzahlen aus P_1 in

$$\begin{aligned} t_{ADL}(l, \pi_{lim}, \delta) &= \#_0(\pi_{lim}) \cdot ((\pi(\delta) - \pi_{lim})/2 \cdot (35B + S(\pi(\delta)/2) + S((l+\delta)/4B)) + \\ &\quad +45B + S(\pi(\delta)/2)) + (l/2B) \cdot (21B + S(l/4B)) + O(1) \\ &= O(\#_0(\pi_{lim}) \cdot (\pi(\delta) - \pi_{lim}) \log(l+\delta)) + O(l \log l) \end{aligned}$$

Bitoperationen und $s_{ADL}(l) = O(1)$ Bits zusätzlichem Speicherplatz. Dabei bezeichnet $\#_0(\pi_{lim})$ die nach π_{lim} Additionen noch nicht gesetzten Bits in $g2$.

Beweis. Für jedes Wort, das ein noch nicht gesetztes Bit enthält, wird jedes einzelne Bit untersucht und im Falle, daß dieses nicht gesetzt ist, so lange die restlichen Primzahlen aufaddiert, bis eine Partition gefunden wurde, oder $\pi(\delta) - \pi_{lim}$ Versuche fehlschlugen. Die äußere **for**-Schleife wird l/B-mal ausgeführt. Die Zeilen 10 – 25 werden maximal $\#_0(\pi_{lim})$-mal durchlaufen, die innere Schleife in Zeile 8 $\pi(\delta) - \pi_{lim}$-mal. □

Die Funktion $\#_0(k)$ wird noch genauer untersucht.

Lemma 4.13 Algorithmus 4.6 verifiziert die Goldbachsche Vermutung für alle geraden Zahlen eines Segments der Länge l in

$$\begin{aligned} t_{ADL}(l, \pi_{lim}) &= B \cdot (t_{ADL}(shiftleft1, l, \delta) + t_{ADL}(P_2\_or\_g2, l, \pi_{lim}, \delta)) + t_{ADL}(l, \pi_{lim}, \delta) \\ &= O(\pi_{lim}(l+\delta)\log(l+\delta)) + O(\#_0(\pi_{lim}) \cdot (\pi(\delta) - \pi_{lim}) \log(l+\delta)) \end{aligned}$$

Bitoperationen und $s_{ADL}(l) = O(1)$ Bits zusätzlichem Speicherplatz.

Beweis. Alle Primsummanden $p \leq p_{\pi_{lim}}$ werden durch Verschiebung des gesamten Intervalls P_2 addiert und die Summen im Feld $g2$ vermerkt. Für alle nicht markierten $2n$ eines Segments werden durch Subtraktion der restlichen Primzahlen $p \leq \pi(\delta)$ Partitionen gesucht. Die Zeitkomplexitätsaussage ergibt sich aus der Summe der Komplexitäten von 4.7, 4.8 und 4.9. □

Satz 4.14 Algorithmus 4.10 verifiziert die Goldbachsche Vermutung für alle geraden Zahlen kleiner gleich n in

$$\begin{aligned} t_{ADL}(n, l, \pi_{lim}) &= n/l \cdot (t_{ADL}(pg\_sieve, l, i) + t_{ADL}(G2segment3, l, \pi_{lim})) + O(1) \\ &= O(n \log l \log\log n) + O(n \log(l+\delta)\pi_{lim}) + \\ &\quad +O\left(\frac{n \log(l+\delta)\#_0(\pi_{lim}) \cdot (\pi(\delta) - \pi_{lim}) \log(l+\delta)}{l}\right) \end{aligned}$$

Bitoperationen und $s_{ADL}(n,l) = l/2 + \log^2 l \log\log l + \pi(2 \cdot \log^2 l \log\log l)B + O(1)$ Bits Speicherplatz.

Beweis. Jedes Segment muß zunächst gesiebt werden, dann kann die Verifikation durch Algorithmus 4.6 vollzogen werden. Das Sieben der Segmente erzeugt nach 3.15 einen Beitrag von insgesamt $O(n \log l \log\log n)$. $\square$

4.4.3 Diskussion

Mit $l \approx \sqrt{n}$ und $\delta = 2\log^2 n \log\log n$ wird $t_{ADL}(G2verify3, n, \pi_{lim}) \approx$

$$O(n \log n \log\log n) + O(n \log n\ \pi_{lim}) + O(\sqrt{n} \log n\ \#_0(\pi_{lim}) \cdot (\pi(\delta) - \pi_{lim}))$$

wobei δ im hinteren Term der Übersichtlichkeit halber nicht ersetzt wurde. Denn π_{lim} liegt zwischen 1 und $\pi(\delta)$, wodurch der letzte Term unwesentlich wird, sobald π_{lim} nahe bei $\pi(\delta)$ liegt. Jedoch wird mit wachsendem π_{lim} der zweite Term größer, so daß dessen Einfluß gegenüber dem ersten wächst. Mit dem Ziel einer späteren Aussage über einen möglichst optimalen Wert für π_{lim} wird nun zunächst $\#_0(\pi_{lim})$, also die nach π_{lim} Verschiebungen und Oder-Verknüpfungen in einem Segment noch nicht gesetzten Bits abgeschätzt.

Im Feld P_2 läßt sich die Anzahl der Primzahlen etwa durch $l/\log l$ abschätzen. Bei der ersten Verschiebung werden also $l/\log l$ Bits im Feld $g2$ gesetzt. Bei der zweiten Verschiebung beträgt die Wahrscheinlichkeit, daß ein zu setzendes Bit bereits gesetzt ist, etwa $\frac{\frac{l}{\log l}}{l} = \frac{1}{\log l}$. Bei $l/\log l$ Bits werden also nur noch $l/\log l - l/\log^2 l$ neu gesetzt. Insgesamt sind nach der zweiten Verknüpfung daher $l/\log l + l/\log l - l/\log^2 l$ Bits auf 1 gesetzt. Allgemein läßt sich die folgende Rekursionsformel angeben:

$$\begin{aligned}\#_1(k) &= \#_1(k-1) + \frac{l}{\log l} \cdot \left(1 - \frac{\#_1(k-1)}{l}\right) \\ &= \#_1(k-1) \cdot \left(1 - \frac{1}{\log l}\right) + \frac{l}{\log l}\end{aligned}$$

Dabei sei $\#_1(k)$ die Anzahl der nach dem k-ten Schritt gesetzten Bits. Diese läßt sich auflösen gemäß

$$\#_1(k) = \left(1 - \frac{1}{\log l}\right)^{k-1} \frac{l}{\log l} + \left(1 - \frac{1}{\log l}\right)^{k-2} \frac{l}{\log l} + \ldots + \left(1 - \frac{1}{\log l}\right) \frac{l}{\log l} + \frac{l}{\log l}$$

und daher

$$\#_1(k) = \frac{l}{\log l} \sum_{m=0}^{k-1} \left(1 - \frac{1}{\log l}\right)^m = \frac{l}{\log l} \cdot \frac{1 - \left(1 - \frac{1}{\log l}\right)^k}{1 - \left(1 - \frac{1}{\log l}\right)} = l\left(1 - \left(1 - \frac{1}{\log l}\right)^k\right)$$

Daher ist die Anzahl noch nicht angekreuzter Bits nach π_{lim} Schritten etwa

$$\#_0(\pi_{lim}) = l\left(1 - \frac{1}{\log l}\right)^{\pi_{lim}}$$

Diese Aussage wird in der Laufzeitanalyse in Abschnitt 4.5.1 zur Anwendung kommen.

4.5 Implementierung und Verteilung

Die Implementierung basierte auf dem in 3.4 vorgestellten Primzahlgenerator `pg`. Der Wert von π_{lim} konnte dabei als weiterer Parameter angegeben werden. Die Segmentierung des Gesamtintervalls $[1, 4 \cdot 10^{14}]$ erfolgte nach dem in Abbildung 19 dargestellten Schema.

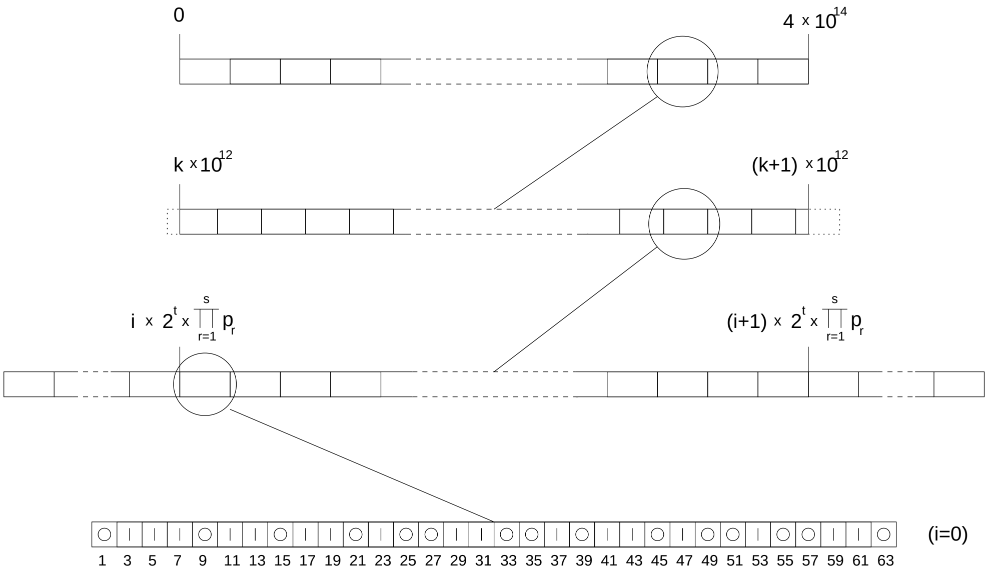


Abbildung 19: Segmentierung des Gesamtintervalls

Die entstehenden 400 Teilintervalle wurden unter Verwendung des in Kapitel 7 beschriebenen Programms zur Verteilung auf 7 Ultra-1- und 6 SPARCstation-4-Workstations sowie 2 PC's verteilt. Nach jeweils etwa 250-500 Segmenten wurde ein Logbucheintrag geschrieben, auf den im Falle eines Ausfalls aufgesetzt werden konnte. Dadurch wurde die maximal verlorene Rechenzeit auf etwa eine Stunde beschränkt. Bei Auftreten eines neuen lokalen Maximalwertes von $p(2n)$ wurde dieser sofort ausgegeben. Die globalen Maxima von $p(2n)$ wurden nach Beendigung der Rechnung aus den Logbuchdateien ermittelt. Es wurde darauf geachtet, daß der Wert von π_{lim} (siehe auch nächster Abschnitt) immer unterhalb des Initialwertes des Maximums von $p(2n)$ lag.

4.5.1 Laufzeitanalyse

Optimale Werte der entscheidenden Parameter $m35$ und der Sieblänge für `pg` sowie π_{lim} wurden für die jeweiligen Architekturen in Intervallen der Länge $2 \cdot 10^8$ im Abstand 10^{13} bestimmt. Als Ansatzpunkte dienten dabei einerseits die in 3.4 gemessenen Werte für $m35$ und die Sieblänge, andererseits Abbildung 20, die die Entwicklung der ADL-Zeitkomplexität von Algorithmus 4.6 in Abhängigkeit des Wertes von π_{lim} für eine Segmentlänge von $2^8 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 = 3843840$ und $n = 2 \cdot 10^{14}$ zeigt. Dabei wurde die oben ermittelte Funktion $\#_0$ als Annäherung für die Anzahl der nach π_{lim} Additionen noch nicht gefundenen Partitionen verwendet.

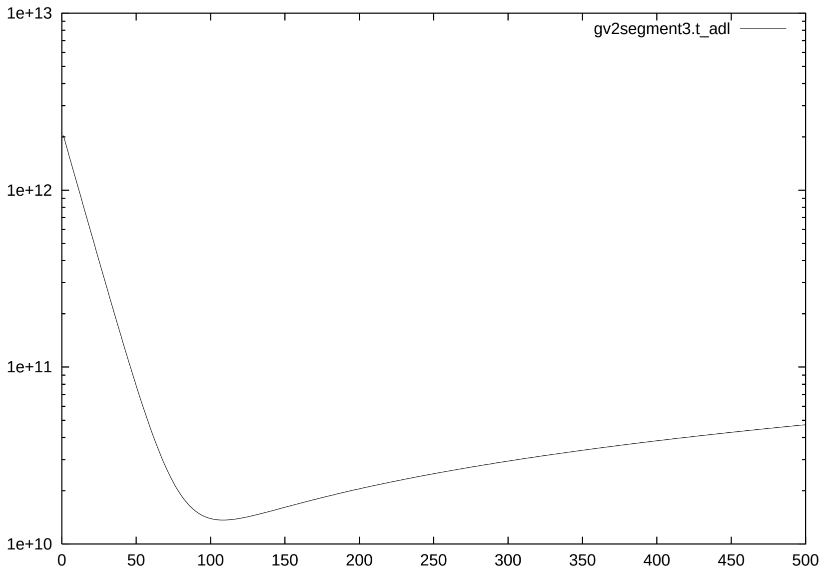


Abbildung 20: ADL-Zeitkomplexität von Algorithmus 4.6 in Abhängigkeit von π_{lim}

Der optimale Wert von π_{lim} scheint bei etwa 100 zu liegen, die tatsächlichen gemessenen Optima lagen darunter (≈ 70). In Abhängigkeit der entstandenen Tabelle der optimalen

Parameterkombinationen wurde die Quelle jeweils mit den optimalen Werten für das entsprechende Teilsegment neu kompiliert. Durch diese Vorgehensweise konnte die Gesamtlaufzeit um etwa 30% reduziert werden. Der Anteil des Siebens an der Gesamtrechnung variierte zwischen 45% bei etwa 10^{12} bis etwa 60% bei $4 \cdot 10^{14}$.

Die gesamte Rechenzeit betrug etwa 18 Wochen. Im Vergleich dazu läßt sich aus den Angaben in [Des98] hochrechnen, daß dort unter Verwendung einer Implementierung von 4.2 auf einem Cray C916 Vektorrechner etwa 11 Wochen benötigt worden wären. Ein direkter Vergleich der Rechenleistung ist nicht möglich. Allerdings wäre eigentlich ein noch größerer Unterschied zu erwarten gewesen, da 4.2 das komplette Sieben vermeidet und ein Strong-Pseudoprimzahltest einer Zahl n nur $O(\log n)$ Operationen benötigt. Der wesentliche Nachteil der Beschränkung von 4.2 auf das Intervall $[4, 341550071728321]$ wird dadurch natürlich nicht aufgehoben.

4.6 Ergebnisse

Die Goldbachsche Vermutung ist bis $4 \cdot 10^{14}$ richtig. Der größte Wert für $p(2n)$, der bis $4 \cdot 10^{14}$ auftritt, ist 5569 und wird für $2n = 389965026819938 = 389965026814369 + 5569$ benötigt.

Tabelle 21 zeigt die Maxima von $p(2n)$ für $2n \leq 4 \cdot 10^{14}$. Die bereits bekannten Werte stimmen mit denen in [Sin93] ermittelten überein, wo eine Verifikation nur bis $4 \cdot 10^{11}$ stattfand. Abbildung 21 zeigt die Entwicklung von $p^*(x)$ für alle $x \leq 4 \cdot 10^{14}$ und zum Vergleich die Funktion $1{,}603 \cdot \log^2 x \log\log x$.

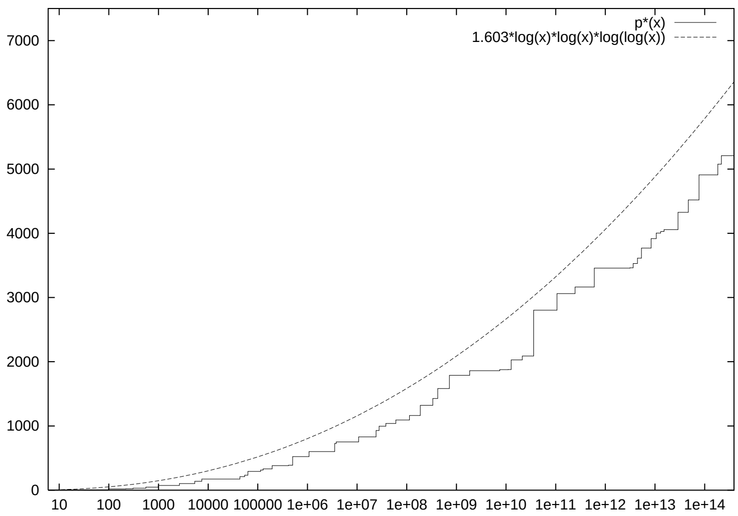


Abbildung 21: Die Funktion $p^*(x) = \max\{p(2n) : 2n \leq x\}$

n	$p(n)$	n	$p(n)$
6	3	113632822	1163
12	5	187852862	1321
30	7	335070838	1427
98	19	419911924	1583
220	23	721013438	1789
308	31	1847133842	1861
556	47	7473202036	1877
992	73	11001080372	1879
2642	103	12703943222	2029
5372	139	21248558888	2089
7426	173	35884080836	2803
43532	211	105963812462	3061
54244	233	244885595672	3163
63274	293	599533546358	3457
113672	313	3132059294006	3463
128168	331	3620821173302	3529
194428	359	4438327672994	3613
194470	383	5320503815888	3769
413572	389	8342945544436	3917
503222	523	10591605900482	4003
1077422	601	12982270197518	4027
3526958	727	15197900994218	4057
3807404	751	28998050650046	4327
10759922	829	46878442766282	4519
24106882	929	76903574497118	4909
27789878	997	184162477860248	5077
37998938	1039	217361316706568	5209
60119912	1093	389965026819938	5569

Tabelle 21: $p^*(n)$

Tatsächlich wächst die Funktion p^* nur sehr langsam. In [Gra89] wurde vermutet, daß die Maxima von $p(n)$ die Größenordnung $O(\log^2 n \log\log n)$ haben.

4.6.1 Konsequenzen

Wie aus Satz 4.1 hervorgeht, folgt die Korrektheit der ternären Vermutung sofort, falls die binäre nachgewiesen ist. Die Kenntnis der Korrektheit der binären Goldbachschen Vermutung für ein endliches Intervall kann folgendermaßen verwendet werden, um auch die ternäre Vermutung partiell zu verifizieren:

Satz 4.15 Es gelte G_2 für ein Intervall $[4, n]$. Falls man eine Folge von Primzahlen $p_0, p_1, \ldots, p_k$ finden kann, so daß gilt $p_0 < n$, $p_k > M$ und $4 < p_i - p_{i-1} < n - 2$ für alle $0 < i \leq k$, dann gilt G_3 für das Intervall $[7, M]$.

Beweis. Es sei $m = 2j + 1 \in [p_{i-1}, p_i] \subset [7, M]$. Dann liegt wegen $4 < p_i - p_{i-1} \leq n - 4$ für $j = i$ oder $j = i - 1$ eine der geraden Zahlen $m - p_j$ im Intervall $[4, n]$ und besitzt daher eine Darstellung $m - p_j = p + q$. Daraus folgt $m = p_j + p + q$. □

In [Sao98] wurde diese Tatsache zusammen mit der in [Sin93] durchgeführten Verifikation verwendet, um G_3 bis 10^{20} zu überprüfen. Mit der Kenntnis, daß G_2 bis $4 \cdot 10^{14}$ nachgewiesen ist, ließe sich diese Rechnung wesentlich vereinfachen. Eine Verifikation von G_3 mit ähnlichen Methoden wie in [Sao98] beschrieben dürfte eine obere Schranke von 10^{23} ermöglichen.

Satz 4.15 impliziert zusammen mit [Che89], daß G_3 nur dann falsch ist, wenn es bis 10^{43000} eine Primzahllücke der Länge $4 \cdot 10^{14}$ gäbe. Man vermutet (siehe z.B. [Rie94]), daß sich eine erste Lücke der Länge d etwa bei $g = e^{1,62\sqrt{d}}$ ergibt. Daher sollte eine erste Lücke der Länge $4 \cdot 10^{14}$ erst bei etwa $10^{14071141}$ auftauchen, also weit hinter 10^{43000}.

5 Goldbach-Partitionen

5.1 Einführung

Nachdem im letzten Kapitel eine Verifikation der Goldbachschen Vermutung vorgenommen wurde, soll nun der Frage nachgegangen werden, *wieviele* verschiedene Zerlegungen der Form (p, q) mit $2n = p + q$ es für eine gerade Zahl $2n$ gibt.

Zunächst die für dieses Kapitel wesentliche

Definition 5.1 Es seien p und q Primzahlen und $n \in \mathbb{N}$. Dann bezeichnet

$$g(2n) = |\{(p, q) : p \leq q, p + q = 2n\}|$$

die *Anzahl der Goldbach-Partitionen* einer Zahl $2n$.

Beispiel 5.2 Die folgende Tabelle zeigt einige Werte von $g(2n)$:

$2n$	$g(2n)$	$2n$	$g(2n)$
4	1	28	2
6	1	30	3
8	1	32	2
10	2	34	4
12	1	36	4
14	2	38	2
16	2	40	3
18	2	42	4
20	2	44	3
22	3	46	4
24	3	48	5
26	3	50	4

Tabelle 22: Einige Werte der Funktion $g(2n)$

Es hat den Anschein, daß die Anzahl der Partitionen wächst. Natürlich muß, solange die Goldbachsche Vermutung nicht bewiesen ist, prinzipiell sogar noch $\liminf_{n\to\infty} g(2n) = 0$ in Betracht gezogen werden. Im folgenden Abschnitt wird zunächst der mathematische Hintergrund beleuchtet. Dabei werden offene Fragen und Vermutungen angesprochen. Es folgt eine Auflistung historischer Berechnungen zum Problem. In Abschnitt 5.4 wird dann ein erster Algorithmus zur Berechnung der Anzahl der Goldbach-Partitionen vorgestellt. Es werden zwei Ansätze zur Segmentierung des Problems aufgezeigt, von denen einer implementiert und das resultierende Programm schließlich verteilt wurde. Ergebnisse dieser Rechnung und Analysen dazu finden sich in den Abschnitten 5.7 und 5.8.

5.2 Mathematischer Hintergrund

Es ist über die Anzahl der Goldbach-Partitionen relativ wenig bekannt. Häufig dienen probabilistische oder heuristische Methoden als Ansatz zur Annäherung an das Problem.

Anders verhält es sich bei der Partitionierung als Summe dreier Primzahlen. Hier bewies Vinogradov im Jahre 1937 [Vin37] die folgende asymptotische Aussage:

$$r_3(n) \sim \frac{n^2}{2(\log n)^3}\left(\prod_{p>2}\left(1+\frac{1}{(p-1)^2}\right)\prod_{\substack{p|n\\p>2}}\left(1-\frac{1}{p^2-3p+3}\right)\right)$$

Bemerkung. Dabei bedeutet

$$r_k(n) = |\{(p_1,\dots,p_k) : p_i \text{ prim und } n = \sum_{i=1}^{k} p_i\}|$$

Die beiden Definitionen sind über $g(2n) = \left\lceil \frac{r_2(2n)}{2} \right\rceil$ miteinander verknüpft. Im folgenden wird die Funktion $g(2n)$ betrachtet.

Die folgende Abschätzung für $g(2n)$ nach oben wird in [Hal74] bewiesen:

$$g(2n) \leq 8\prod_{p>2}\left(1-\frac{1}{(p-1)^2}\right)\prod_{\substack{p>2\\p|n}}\frac{p-1}{p-2}\frac{n}{\log^2 2n}\cdot\left(1+O\left(\frac{\log\log n}{\log n}\right)\right)$$

Dies führt wegen $\prod_{\substack{p>2\\p|n}}\frac{p-1}{p-2} = O(\log\log n)$ zu

$$g(2n) = O\left(\frac{2n}{\log^2 2n}\log\log 2n\right)$$

Deshoulliers et.al. [Des93] zeigen

$$g(2n) \leq \pi(2n-2) - \pi(n-1),$$

wobei in derselben Arbeit bewiesen wurde, daß die größte Zahl $2n$, für die Gleichheit besteht, 210 ist.

Die Anzahl der Goldbach-Partitionen einer geraden Zahl $2n$ hängt sehr stark von ihrer Primfaktorenzerlegung ab. So ist beispielsweise $g(120) = 12$, während für die Nachbarn gilt: $g(118) = 6$ und $g(122) = 4$. Die Primfaktorenzerlegungen sind jeweils $118 = 2\cdot 59$, $122 = 2\cdot 61$ und $120 = 2\cdot 2\cdot 2\cdot 3\cdot 5$. Demgegenüber besitzt die Zahl $2048 = 2^{11}$ genau 25 Goldbach-Partitionen, während im Vergleich $g(2046) = 75$ und $g(2050) = 42$ ist.

Wie im Abschnitt 5.8 auch graphisch deutlich sichtbar wird, setzt sich dieser Trend fort. Gewisse Zahlen scheinen mehr Partitionen zu besitzen als andere.

Dies läßt sich durch folgende, heuristische Betrachtung begründen:

Angenommen 3 teile $2n$, dann sind die Zahlen $2n-i$ für $i = 5, 7, 11, 13, 17, 19, 23, 25, \ldots$ nicht durch 3 teilbar. Da im Falle, daß die 3 $2n$ nicht teilt, jede zweite Zahl $2n-i$ durch 3 teilbar ist, ist die Wahrscheinlichkeit, daß sich im ersten Fall eine Goldbach-Partition ergibt, doppelt so groß wie im zweiten. Es gelte nun $5|2n$. Dann sind die Zahlen $2n-i$ für $i = 3, 7, 9, 11, 13, 17, 19, \ldots$ nicht durch 5 teilbar, im Falle $5 \not| 2n$ jedoch jede vierte, d.h. es kommen dann nur noch 3 von 4 für eine Partition in Frage. Die Anzahl der Partitionen sollte sich also bei $5|2n$ um den Faktor $\frac{4}{3}$ erhöhen. Im allgemeinen Fall müßte sie für jeden Primteiler p von $2n$ um einen Faktor $\frac{p-1}{p-2}$ erhöht werden. Daraus folgt, daß mit der Anzahl der verschiedenen Primfaktoren einer Zahl die Anzahl ihrer Goldbach-Partitionen steigt. Dies erklärt die deutlichen Unterschiede bei obigen Beispielen.

Die Wahrscheinlichkeit, daß zwei Summanden einer Zahl $2n$ beide prim sind, beträgt wegen A.7 etwa $\frac{1}{\log^2 2n}$. Die Anzahl der Partitionen sollte daher insgesamt etwa gleich $\frac{2n}{\log^2 2n}$ sein, wobei für jeden ungeraden Primteiler von $2n$ wegen obiger Betrachtung ein Faktor $\frac{p-1}{p-2}$ angefügt werden muß. Damit ergibt sich eine grobe Abschätzung von $g(2n) \approx \frac{2n}{\log^2 2n} \prod_{\substack{p>2 \\ p|2n}} \frac{p-1}{p-2}$. Dieser Ansatz stellt die Basis fast aller Vermutungen dar, die im folgenden Abschnitt vorgestellt werden, wobei jeweils ein korrigierender Faktor angefügt ist.

5.2.1 Vermutungen

In diesem Abschnitt werden verschiedene Vermutungen bezüglich des asymptotischen Verhaltens von g vorgestellt, die sämtlich auf heuristischen Argumenten basieren. Die jeweiligen Vermutungen sind mit den Kürzeln der Autoren indiziert, die sie getroffen haben, um sie später leichter referenzieren zu können. Es wird dabei jeweils vermutet, daß gilt: $g(2n) \sim g_{XY}(2n)$.

Die erste Aussage über das mögliche asymptotische Verhalten der Funktion g stammt von *Sylvester* [Syl71] aus dem Jahre 1871:

$$g_{Sy}(2n) = \frac{4n}{\log 2n} \prod_{\substack{2<k<\sqrt{2n} \\ k \not| 2n}} \frac{k-2}{k-1}$$

Hardy und *Littlewood* zeigen in [Har22], daß dies äquivalent zu

$$g_{Sy}(2n) = 8e^{-\gamma} c_2 \cdot \frac{n}{\log^2 2n} \prod_{\substack{p>2 \\ p|2n}} \frac{p-1}{p-2}$$

ist, wobei $\gamma = 1 - \int_1^\infty \frac{t-\lfloor t \rfloor}{t^2} dt \approx 0{,}577216\ldots$ die *Eulersche Konstante* und $c_2 = \prod_{p>2} \frac{p \cdot (p-2)}{(p-1)^2} \approx 0{,}660162\ldots$ die *Primzahlzwillingskonstante* ist.

Im Jahre 1896 vermutete *Stäckel* [Stä96]:

$$g_{St}(2n) = 4 \cdot \frac{n}{\log^2 2n} \cdot \prod_{\substack{p>2 \\ p|2n}} \frac{p}{p-1}$$

Landau korrigierte dies 1900 [Lan00] um den Faktor $0,772$:

$$g_{La}(2n) = 3,088 \cdot \frac{n}{\log^2 2n} \cdot \prod_{\substack{p>2 \\ p|2n}} \frac{p}{p-1}$$

Brun [Bru15] vermutete 1915 eine Formel, von der wiederum Hardy und Littlewood zeigen, daß sie äquivalent zu:

$$g_{Br}(2n) = 16e^{-2\gamma} c_2 \cdot \frac{n}{\log^2 2n} \cdot \prod_{\substack{p>2 \\ p|2n}} \frac{p-1}{p-2}$$

ist. Hardy und Littlewood selbst vermuten (ebenfalls in [Har22]), daß

$$g_{HL}(2n) = 4c_2 \cdot \frac{n}{\log^2 2n} \cdot \prod_{\substack{p>2 \\ p|2n}} \frac{p-1}{p-2}$$

Es gilt also

$$g_{HL}(2n) = \frac{e^{\gamma}}{2} \cdot g_{Sy}(2n) = \frac{e^{2\gamma}}{4} \cdot g_{Br}(2n)$$

(oder $g_{HL}(2n) \approx 0,8905 \cdot g_{Sy}(2n) \approx 0,793 \cdot g_{Br}(2n)$).

Die letzte Vermutung zum asymptotischen Verhalten der Funktion g stammt von *Selmer* aus dem Jahre 1942 ([Sel42]). Sie besagt, daß

$$g_{Se}(2n) = 2c_2 \cdot \prod_{\substack{p>2 \\ p|2n}} \left(\frac{p-1}{p-2}\right) \cdot \int_0^n \frac{dx}{\log(n+x)\log(n-x)}$$

In Abschnitt 5.8 werden diese mit den sich aus den Rechnungen ergebenden Werten von $g(2n)$ verglichen.

5.3 Historische Berechnungen

Es gibt nicht sehr viele veröffentlichte Arbeiten, in denen die Anzahl der Goldbach-Partitionen bestimmt wurde. Dies liegt wohl vor allem im Aufwand begründet, den diese Rechnung erfordert.

Tabelle 23 gibt einen Überblick über Ergebnisse der Vergangenheit, von denen jede jeweils frühere Berechnungen erweiterte.

Name	Jahr	$p+q \leq$
G. Cantor	1894	1000
R. Haussner	1896	5000
M.L. Stein, P.R. Stein	1965	200000
J. Bohman, C.E. Fröberg	1975	350000
D. Lavenier, Y. Saouter	1998	$1,28 \cdot 10^8$

Tabelle 23: Historische Berechnungen

In der zuletzt aufgeführten Rechnung wurde eine Methode benutzt, auf deren Prinzip im Abschnitt 5.5 noch näher eingegangen wird. Dort wurde erstmals eine speziell auf das Problem zugeschnittene Hardware entwickelt und eingesetzt.

Die Ergebnisse dieser Arbeit erweitern den bisherigen Bereich auf alle Goldbach-Partitionen (p,q) mit $p+q \leq 5 \cdot 10^8$.

5.4 Ein Algorithmus zur Anzahl der Goldbach-Partitionen

In diesem Abschnitt wird zunächst ein einfacher Algorithmus zur Berechnung der Anzahl der Goldbach-Partitionen vorgestellt. Es handelt sich dabei im Prinzip um eine leicht veränderte Version von Algorithmus 4.2. Der wesentliche Nachteil ist der immense Speicherbedarf, der in der Praxis schnell an Grenzen stößt.

Algorithmus 5.1 *part1*

```
n ← input()   { Eingabe der Intervallobergrenze }
sieve ← erat5(n)   { Siebe Intervall }
i ← 3
while i ≤ (n ≫ 1) do  { Durchlaufe ungerade Primzahlen bis n/2 }
  if getBit2(sieve, i) = 0 then  { Falls i prim }
    j ← i
    while j ≤ (n − i) do  { Durchlaufe alle Primzahlen von i bis n − i }
      if getBit2(sieve, j) = 0 then  { Falls j prim }
        g[(i + j) ≫ 1] ← g[(i + j) ≫ 1] + 1
      end if
      j ← j + 2
    end while
  end if
  i ← i + 2
end while
return g
```

Satz 5.3 Algorithmus 5.1 bestimmt die Anzahl der Goldbachpartitionen aller geraden Zahlen zwischen 6 und n in

$$\begin{aligned} t_{ADL}(n) &= (35B + S(n/4B)) \cdot n/2 + 10B \cdot \pi(n/2) + \sum_{i=2}^{n/2} \sum_{j=i}^{n-i} 28B + \\ &\quad + \sum_{p \leq n/2} \sum_{p \leq q \leq n-p} (23B + S(n/4)) + O(n \log n \log\log n) \\ &= (14B + S(n/4B)) \cdot n/2 + 10B \cdot \pi(n/2) + 21B \cdot n^2/4 + \\ &\quad + O(\log n \cdot \pi(n/2) \cdot \pi(n)) + O(n \log n \log\log n) \\ &= 21B \cdot n^2/4 + O(n^2/\log n) = O(n^2) \end{aligned}$$

Bitoperationen unter Verwendung von $s_{ADL}(n) = (B+1) \cdot n/2 + O(1)$ Bits Speicher.

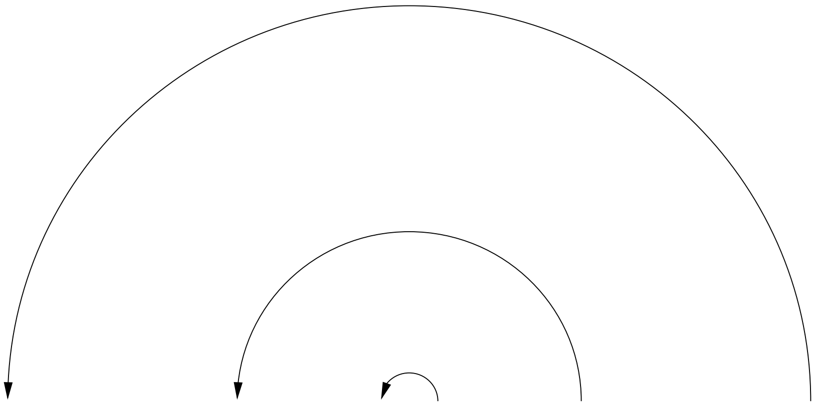
Beweis. Für jede Primzahl $p \leq n/2$ werden beginnend von p aus alle Primzahlen $q \leq n-p$ durchlaufen und der Inhalt des Feldes g jeweils an der Position $(p+q)/2$ erhöht. Den wesentlichen Anteil des benötigten Speichers nimmt das Feld g ein, das mit $nB/2$ beiträgt. □

Das Problem von Algorithmus 5.1 ist der immense Speicheraufwand. Ähnlich der nicht-segmentierten Versionen des Siebes des Eratosthenes muß das ganze Feld im Speicher vorhanden sein. Im nächsten Abschnitt wird dies – wenn auch mit nicht unerheblichem Zeitmehraufwand – vermieden.

5.5 Spiegelung von Segmenten

In diesem Abschnitt wird zunächst ein Verfahren beschrieben, daß den erheblichen Speicheraufwand von Algorithmus 5.1 zwar reduziert, auf normaler Hardware aber wiederum entscheidende Nachteile aufweist.

Um $g(2n)$ für ein einzelnes n zu bestimmen, kann man folgendermaßen vorgehen: Zunächst wird das Intervall $[1, 2n]$ in k Teilsegmente der Länge $\lfloor 2n/k \rfloor$ geteilt. Diese Segmente s_i werden nun jeweils paarweise beginnend mit $(s_1, s_{\lfloor 2n/k \rfloor})$ gesiebt. Für jedes Paar $(s_i, s_{\lfloor 2n/k \rfloor - i+1})$ wird dann s_i gespiegelt und das Ergebnis mit $s_{\lfloor 2n/k \rfloor - i+1}$ durch bitweises Und verknüpft. $g(2n)$ ergibt sich nun aus der Summe der verbleibenden 1-Bits nach Abarbeitung sämtlicher Segmente. Das entstehende Reststück wird im Grunde auf dieselbe Weise behandelt. Abbildung 22 zeigt die Vorgehensweise am Beispiel $2n = 72$:



1	0	1	0	0	1	1	0	0	1	0	0	1	0	1	1	0	1	0	0	1	1	0	1	1	0	1	1	0	1	1	0	1	1	1	0
71	69	67	65	63	61	59	57	55	53	51	49	47	45	43	41	39	37	35	33	31	29	27	25	23	21	19	17	15	13	11	9	7	5	3	1

Abbildung 22: Segmentweise Spiegelung

Auf eine *ADL*-Beschreibung wurde hier verzichtet, da eine Implementierung auf *RX-RAM*-ähnlichen Maschinen nicht sinnvoll ist. Dies liegt vor allem daran, daß die bitweisen Operationen meist Wortoperationen nach sich ziehen, was insgesamt sehr teuer wird. Allerdings ist in [Lav98] eine Hardwareimplementierung des Verfahrens beschrieben, wobei nicht nur ein einzelner g-Wert, sondern viele parallel berechnet werden. Dies wird durch das Durchschieben ganzer Segmente erreicht. Die Implementierung resultierte in der Berechnung der Anzahlen der Goldbach-Partitionen bis $1{,}28 \cdot 10^8$.

Es folgt die Beschreibung eines weiteren, für gewöhnliche Rechnerarchitekturen geeigneten, segmentierten Verfahrens, daß schließlich implementiert und eingesetzt wurde.

5.6 Segmentierung des Basisverfahrens

Es wird nun gezeigt, wie Algorithmus 5.1 segmentiert werden kann. Dabei bleibt das Grundprinzip von 5.1 erhalten.

Zunächst werden folgende Vorüberlegungen getroffen:

Das Sieben der Primzahlen zur Ermittlung der möglichen Summanden ist vergleichsweise billig. Wie bei Algorithmus 5.1 sichtbar wurde, stammt der wesentliche quadratische Anteil der Zeitkomplexität vom doppelten Durchlaufen des Intervalls. Die Abschätzung des Siebbeitrages zur Zeitkomplexität erfolgte relativ grob, die O-Konstante ist daher eher klein. Ein zwar redundantes, mehrfaches Sieben der Intervalle erscheint also zumindest bezahlbar, wenn auch nicht sofort offensichtlich hilfreich.

Es sei im folgenden das Intervall $[1, n]$ in k Segmente $s_0, \dots, s_{k-1}$ der Länge n/k geteilt, wobei zusätzlich $M_j \mid k$ und $k \mid n$ für ein $j \approx \sqrt{n}$ gelte.

Dann sind die zwei Summanden einer Partition (p, q) mit $p \leq q$ der geraden Zahlen eines Segments s_i jeweils in Segmenten laut Tabelle 24 enthalten.

$p \in$	$q \in$
s_0	$s_{i-1} \cup s_i$
s_1	$s_{i-2} \cup s_{i-1}$
s_2	$s_{i-3} \cup s_{i-2}$
$\dots$	$\dots$
s_j	$s_{i-j-1} \cup s_{i-j}$
$\dots$	$\dots$
$s_{\lfloor i/2 \rfloor}$	$s_{i-\lfloor i/2 \rfloor-1} \cup s_{i-\lfloor i/2 \rfloor}$

Tabelle 24: Mögliche Teilmengen zur Partitionierung

Das Prinzip von Algorithmus 5.2 ist nun das folgende: Für alle Segmente $s_i \subset [1, n]$ werden s_j und s_{i-j-1} sowie s_{i-j} für alle $j \in [0, \lfloor i/2 \rfloor]$ gesiebt. Dann werden die Summen $p + q$, $p \in s_j$, $q \in s_{i-j-1} \cup s_{i-j}$ gebildet und darauf geprüft, ob $p + q \in s_i$. Gegebenenfalls wird der Inhalt des Feldes g an der Position $p + q$ erhöht.

Dabei sind noch zwei Probleme zu beachten:

Für $j > i - j - 1$ darf keine Summation stattfinden, diese Situation ist schon bearbeitet.

Für $j = i - j - 1$ bzw. $j = i - j$ muß zusätzlich auf $p \leq q$ geachtet werden.

Algorithmus 5.2 beschreibt die Vorgehensweise für ein Intervall s_i, $i \geq 1$. Die wesentlichen zusätzlichen Funktionen add_{low}, add_{mid} und add_{high} zur Addition fertig gesiebter Segmente sind im Anschluß daran beschrieben.

Beispiel 5.4 Für $i = 5$ bzw. 6 werden die in Tabelle 25 aufgeführten Additions-Funktionen durchlaufen.

$i = 5$	$i = 6$
$add_{low}(g, l, s_0, s_4)$	$add_{low}(g, l, s_0, s_5)$
$add_{high}(g, l, s_0, s_5)$	$add_{high}(g, l, s_0, s_6)$
$add_{low}(g, l, s_1, s_3)$	$add_{low}(g, l, s_1, s_4)$
$add_{high}(g, l, s_1, s_4)$	$add_{high}(g, l, s_1, s_5)$
$add_{mid}(g, l, i, s_2)$	$add_{low}(g, l, s_2, s_3)$
$add_{high}(g, l, s_2, s_3)$	$add_{high}(g, l, s_2, s_4)$
	$add_{mid}(g, l, i, s_3)$

Tabelle 25: Beispielabläufe Algorithmus 5.2

Algorithmus 5.2 *seg1part1*

```
i ← input()   { Eingabe der Segmentnummer }
l ← input()   { Eingabe der Segmentlänge }
gaps ← input()   { Eingabe der Primzahllücken }
s_i ← pg_sieve(gaps, i, l)   { Siebe höchstes Segment }
if i ∧ 1 = 1 then   { Falls i ungerade }
  for j ← 0 to i ≫ 1 do   { Durchlaufe Segmente von 0 bis ⌊i/2⌋ }
    s_j ← pg_sieve(gaps, j, l)   { Siebe niedriges Segment }
    s_{i−j−1} ← pg_sieve(gaps, i − j − 1, l)   { Siebe zweithöchstes Segment }
    if j = i − j − 1 then   { Mitte erreicht }
      g ← add_mid(g, l, i, s_j)   { Addiere mittleres Segment }
    else
      g ← add_low(g, l, s_j, s_{i−j−1})   { Addiere niedriges und zweithöchstes Segment }
    end if
    g ← add_high(g, l, s_j, s_{i−j})   { Addiere niedriges und höchstes Segment }
    s_{i−j} ← s_{i−j−1}   { Zweithöchstes Segment wird höchstes }
  end for
else
  for j ← 0 to i ≫ 1 − 1 do   { Durchlaufe Segmente von 0 bis i/2 − 1 }
    s_j ← pg_sieve(gaps, j − 1, l)   { Siebe niedriges Segment }
    s_{i−j} ← pg_sieve(gaps, i − j − 1, l)   { Siebe zweithöchstes Segment }
    g ← add_low(g, l, s_j, s_{i−j−1})   { Addiere niedriges und zweithöchstes Segment }
    g ← add_high(g, l, s_j, s_{i−j})   { Addiere niedriges und höchstes Segment }
    if j = i ≫ 1 − 1 then   { Mitte erreicht }
      g ← add_mid(g, l, i, s_j)   { Addiere mittleres Segment }
      break
    end if
    s_{i−j} ← s_{i−j−1}   { Zweithöchstes Segment wird höchstes }
  end for
end if
return g
```

Algorithmus 5.3 add_{low}

```
g ← input()   { Eingabe des Partitionsfeldes }
l ← input()   { Eingabe der Segmentlänge }
s_low ← input()   { Eingabe des niedrigen Segments }
s_high ← input()   { Eingabe des höheren Segments }
lowwordcnt ← 0
while lowwordcnt < l ≫ b do  { Durchlaufe s_low wortweise }
  lowword ← s_low[lowwordcnt]
  if lowword ≠ ¬0 then  { Überhaupt eine Primzahl im aktuellen Wort? }
    lowbitcnt ← 0
    repeat  { Durchlaufe aktuelles Wort }
      if ((1 ≪ lowbitcnt) ∧ lowword) = 1 then  { Primzahl gefunden }
        hiwordcnt ← 0
        while hiwordcnt ≠ lowwordcnt do  { Durchlaufe Wörter von s_high }
          hiword ← s_high[l ≫ b − hiwordcnt − 1]
          hibitcnt ← 0
          repeat  { Durchlaufe aktuelles Wort }
            if ((1 ≪ hibitcnt) ∧ hiword) = 1 then  { Primzahl gefunden }
              pos ← (lowwordcnt − hiwordcnt − 1) << b + lowbitcnt + hibitcnt
              g[pos] ← g[pos] + 1
            end if
            hibitcnt ← hibitcnt + 1
          until hibitcnt = B
          hiwordcnt ← hiwordcnt + 1
        end while
        hiword ← s_high[l ≫ b − hiwordcnt − 1]
        hibitcnt ← 0
        while hibitcnt ≠ B do  { Durchlaufe kritische Zahlen mit Intervallprüfung
        }
          if ((1 ≪ hibitcnt) ∧ hiword) = 1 then  { Primzahl gefunden }
            cand ← hibitcnt + lowbitcnt − B
            if cand ≤ 0 then  { Im Zielintervall }
              g[cand] ← g[cand] + 1
            end if
          end if
          hibitcnt ← hibitcnt + 1
        end while
      end if
      lowbitcnt ← lowbitcnt + 1
    until lowbitcnt = B
  end if
  lowwordcnt ← lowwordcnt + 1
end while
return
```

Algorithmus 5.4 add_{high}

```
g ← input()   { Eingabe des Partitionsfeldes }
l ← input()   { Eingabe der Segmentlänge }
s_low ← input()   { Eingabe des niedrigen Segments }
s_high ← input()   { Eingabe des höheren Segments }
lowwordcnt ← 0
while lowwordcnt < l ≫ b do  { Durchlaufe s_low wortweise }
  lowword ← s_low[lowwordcnt]
  if lowword ≠ ¬0 then  { Überhaupt eine Primzahl im aktuellen Wort? }
    lowbitcnt ← 0
    repeat  { Durchlaufe aktuelles Wort }
      if ((1 ≪ lowbitcnt) ∧ lowword) = 1 then  { Primzahl gefunden }
        hiwordcnt ← 0
        while hiwordcnt ≠ l ≫ b − lowwordcnt − 1 do  { Durchlaufe Wörter von s_high }
          hiword ← s_high[hiwordcnt]
          hibitcnt ← 0
          repeat  { Durchlaufe aktuelles Wort }
            if ((1 ≪ hibitcnt) ∧ hiword) = 1 then  { Primzahl gefunden }
              pos ← (lowwordcnt + hiwordcnt) << b + lowbitcnt + hibitcnt
              g[pos] ← g[pos] + 1
            end if
            hibitcnt ← hibitcnt + 1
          until hibitcnt = B
          hiwordcnt ← hiwordcnt + 1
        end while
        hiword ← s_high[hiwordcnt]
        hibitcnt ← 0
        while hibitcnt ≠ B do  { Durchlaufe kritische Zahlen mit Intervallprüfung }
          if ((1 ≪ hibitcnt) ∧ hiword) = 1 then  { Primzahl gefunden }
            cand ← hibitcnt + lowbitcnt
            if cand < B then  { Im Zielintervall }
              pos ← (l ≫ b − 1) ≪ b + cand
              g[pos] ← g[pos] + 1
            end if
          end if
          hibitcnt ← hibitcnt + 1
        end while
      end if
      lowbitcnt ← lowbitcnt + 1
    until lowbitcnt = B
  end if
  lowwordcnt ← lowwordcnt + 1
end while
return
```

Algorithmus 5.5 add_{mid}

```
g ← input()   { Eingabe des Partitionsfeldes }
l ← input()   { Eingabe der Segmentlänge }
i ← input()   { Eingabe der Segmentnummer }
s_low ← input()   { Eingabe des mittleren Segments }
s_high ← s_low   { Zweiter Verweis auf mittleres Segment }
lowwordcnt ← 0
il ← i × l   { Vorberechnung }
lowl ← low × l   { Vorberechnung }
while lowwordcnt < l ≫ b do  { Durchlaufe s_low wortweise }
  lowword ← s_low[lowwordcnt]
  if lowword ≠ ¬0 then  { Überhaupt eine Primzahl im aktuellen Wort? }
    lowbitcnt ← 0
    repeat  { Durchlaufe aktuelles Wort }
      if ((1 ≪ lowbitcnt) ∧ lowword) = 1 then  { Primzahl gefunden }
        hibitcnt ← lowbitcnt
        while hibitcnt < B do  { Durchlaufe aktuelles Wort bis zum Ende }
          if (1 ≪ hibitcnt) ∧ loword = 1 then  { Primzahl gefunden }
            cand ← (lowwordcnt ≪ (b + 1) + lowbitcnt + hibitcnt + lowl) ≪ 1
            if cand > il and cand ≤ il + l then  { Im Zielintervall }
              g[(cand − il) ≫ 1] ← g[(cand − il) ≫ 1] + 1
            end if
          end if
          hibitcnt ← hibitcnt + 1
        end while
        hiwordcnt ← lowwordcnt + 1
        while hiwordcnt < l ≫ b do  { Durchlaufe restliche Wörter }
          hibitcnt ← 0
          hiword ← s_low[hiwordcnt]
          while hibitcnt < B do  { Durchlaufe aktuelles Wort bis zum Ende }
            if (1 ≪ hibitcnt) ∧ loword = 1 then  { Primzahl gefunden }
              cand ← ((lowwordcnt + hiwordcnt) ≪ b + lowbitcnt + hibitcnt +
              lowl) ≪ 1
              if cand > il and cand ≤ il + l then  { Im Zielintervall }
                g[(cand − il) ≫ 1] ← g[(cand − il) ≫ 1] + 1
              end if
            end if
            hibitcnt ← hibitcnt + 1
          end while
          hiwordcnt ← hiwordcnt + 1
        end while
      end if
      lowbitcnt ← lowbitcnt + 1
    until lowbitcnt = B
  end if
  lowwordcnt ← lowwordcnt + 1
end while
return
```

Auf einen Beweis der Korrektheit der Additionsalgorithmen wird aus Platzgründen verzichtet. Allerdings soll das Prinzip verdeutlicht werden.

Abbildung 23 zeigt eine Beispielsituation.

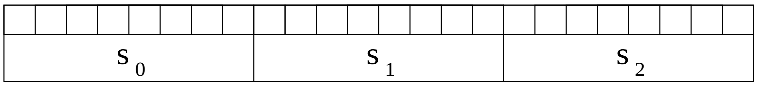


Abbildung 23: Beispielsegmente

Dabei bestehe jedes Segment aus acht Wörtern. Der Ablauf der Additionen ist hier wie folgt:

$add_{low}(g, l, s_0, s_1)$
$add_{mid}(g, l, i, s_1, s_1)$
$add_{high}(g, l, s_0, s_2)$

Die Algorithmen 5.3, 5.4 und 5.5 durchlaufen die Segmente jeweils wortweise. Für jedes Wort wird jedes Bit geprüft und gegebenenfalls die Addition eingeleitet. Dabei können relativ aufwendige **if**-Konstrukte zur Überprüfung, ob gefundene Summen $p + q$ sich tatsächlich im Zielintervall befinden (hier s_2), weitestgehend vermieden werden.

Algorithmus 5.3 addiert die Primzahlen jeweils wie in Abbildung 24 dargestellt. Summen aus schwarzen Wortbereichen liegen auf jeden Fall im Zielsegment. Daher können Abfragen auf Prüfung von $p \leq q$ gespart werden. Summanden aus grau unterlegten Wörtern können noch zum Zielsegment beitragen, hier sind jedoch Abfragen notwendig. Entsprechendes gilt für das Addieren durch 5.4 wie in Abbildung 25 dargestellt.

Die Addition der ja eher selten auftretenden mittleren Segmente durch 5.5 funktioniert etwas aufwendiger, da das zu addierende Segment sich je nach gerader oder ungerader Segmentanzahl unterscheidet. Allerdings müssen für jede gefundene Primzahl p nur noch $l - p$ Bits durchlaufen werden.

Eine weitere Optimierungsmöglichkeit besteht darin, Wörter blockweise (z.B. byteweise) zu durchlaufen und jeweils zu prüfen, ob der Block nur aus Einsen besteht. Solche Blöcke können übersprungen werden. In der Implementierung findet sich diese Vorgehensweise wieder, in obigen Algorithmen werden nur ganze Wörter auf Inhalt geprüft.

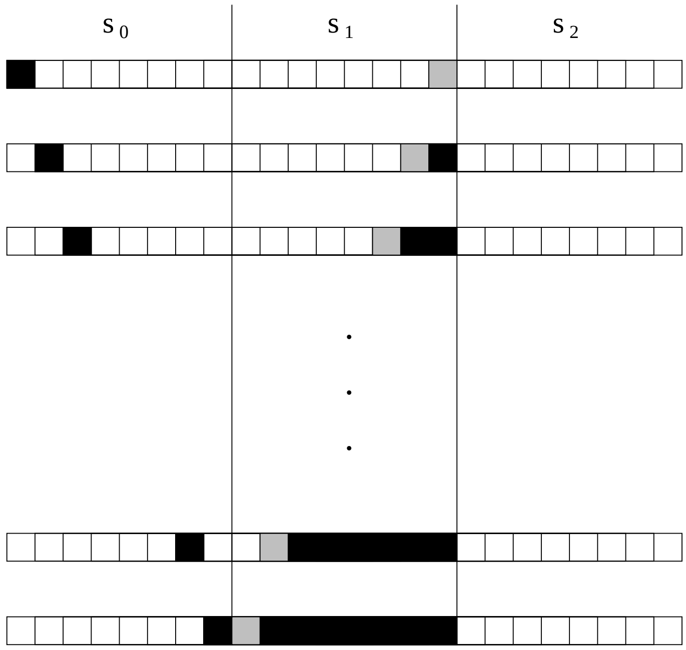


Abbildung 24: Beispieladdition add_{low}

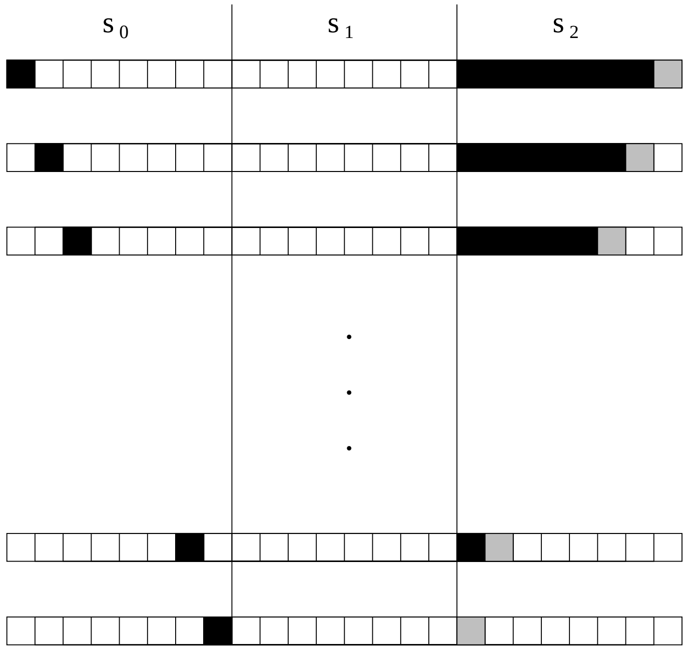


Abbildung 25: Beispieladdition add_{high}

Lemma 5.5 Algorithmus 5.3 summiert die Primzahlen zweier Intervalle der Länge l in

$$t_{ADL}(l) \quad = \quad O(l^2)$$

Bitoperationen und $s_{ADL}(l) = O(1)$ Bits (zusätzlichem) Speicher.

Beweis. (Komplexitäten) Die Schleife in Zeile 6 trägt mit $l/B \cdot (24B + S(l/B))$ Operationen bei, Zeile 11 mit $11B \cdot l$, die Zeilen 12–35 mit $\pi(l) \cdot (28B + S(l/B))$. Die Zeilen 14–23 kosten $\pi(l) \cdot l/2B \cdot (31B + S(l/B))$, 16–22 $\pi(l) \cdot l/2B \cdot 21B$ Operationen. Das Erhöhen des g-Feldes in Zeile 19 ergibt zusammen mit Zeile 18 $\pi(l) \cdot \pi(l) \cdot (28B + S(l/2B))$. Schließlich wird die Schleife in Zeile 27 B mal durchlaufen, dies jeweils $\pi(l)$ mal, also: $\pi(l) \cdot B \cdot (45B + S(l/2B))$. Insgesamt ergibt sich

$$\frac{l}{B} \cdot \big(11B^2 + 24B + 11 + S(l/B)\big) + \pi(l) \cdot \big(45B^2 + 28B + S(l/B) + B \cdot S(l/B)\big) +$$

$$+\pi(l) \cdot \frac{l}{2B} \cdot (21B^2 + 31B + S(l/B)) + \pi(l)^2 \cdot (28B + S(l/2B)) =$$

$$O(l \log l) + O(l) + O(l^2) + O(l^2/\log l) + O(l^2/(\log l)^2) = O(l^2)$$

Zur Raumkomplexität: Die Felder werden jeweils übergeben, 5.3 benötigt nur konstant viel zusätzlichen Speicher. □

Lemma 5.6 Algorithmus 5.4 summiert die Primzahlen zweier Intervalle der Länge l in

$$t_{ADL}(l) \quad = \quad O(l^2)$$

Bitoperationen und $s_{ADL}(l) = O(1)$ Bits (zusätzlichem) Speicher.

Beweis. (Komplexitäten) Analog 5.3, es ergibt sich hier

$$\frac{l}{B} \cdot \big(11B^2 + 24B + 11 + S(l/B)\big) + \pi(l) \cdot \big(59B^2 + 37B + S(l/B) + B \cdot S(l/B)\big) +$$

$$+\pi(l) \cdot \frac{l}{2B} \cdot (21B^2 + 22B + S(l/B)) + \pi(l)^2 \cdot (26B + S(l/2B)) = O(l^2)$$

An der Raumkomplexität hat sich nichts verändert. □

Lemma 5.7 Algorithmus 5.5 summiert die Primzahlen zweier Intervalle der Länge l in

$$t_{ADL}(l) \quad = \quad O(l^2)$$

Bitoperationen und $s_{ADL}(l) = O(1)$ Bits (zusätzlichem) Speicher.

Beweis. (Komplexitäten) Es ergeben sich hier

$$\frac{l}{B} \cdot (23B + 11 + S(l/B)) + \pi(l) \cdot (77B^2 + S(l/2B)) +$$

$$+\pi(l) \cdot \frac{l}{2B} \cdot (21B^2 + 13B + S(l/2B)) + \pi(l)^2 \cdot (56B + S(l/2B)) = O(l^2)$$

Operationen bei wiederum konstanten (zusätzlichem) Speicher. □

Satz 5.8 Algorithmus 5.2 bestimmt die Anzahl der Goldbach-Partitionen aller geraden Zahlen eines Segments s_i, $(i \geq 1)$ der Länge l in

$$t_{ADL}(i, l) \quad = \quad O(i \cdot l^2)$$

Bitoperationen und $s_{ADL}(i, l) = (B + 1) \cdot l/2 + \pi(\sqrt{i \cdot l})B + O(1)$ Bits Speicher.

Beweis. Die Funktionen add_{low} und add_{high} werden jeweils maximal $i/2$ mal ausgeführt, add_{mid} jeweils einmal. Das Sieben von Segmenten findet i mal statt und erzeugt damit einen Beitrag von $O(i \cdot l \log l \log\log i \cdot l)$. Insgesamt ergeben sich

$$i \cdot \Bigg(\frac{l}{B} \cdot (11B^2 + 24B + 11 + S(l/B)) + \pi(l) \cdot (52B^2 + 32,5B + (B + 1) \cdot S(l/B))) +$$

$$+\pi(l) \cdot \frac{l}{2B}(21B^2 + 26,5B + S(l/B)) + \pi(l)^2 \cdot (28B + S(l/2B)) \Bigg) +$$

$$\frac{l}{B} \cdot (23B + 11 + S(l/B)) + \pi(l) \cdot (77B^2 + S(l/2B)) +$$

$$+\pi(l) \cdot \frac{l}{2B}(21B^2 + 13B + S(l/B)) + \pi(l)^2 \cdot (56B + S(l/2B)) + O(i \cdot l \cdot \log l \log\log i \cdot l)$$

$$= O(i \cdot l^2)$$

Bitoperationen. Benötigt werden für das Feld g $l/2$ Wörter, zusammen mit dem Sieb also $(B+1) \cdot l/2$ Bits. Das Feld der Primzahldifferenzen benötigt zusätzlich $\pi(\sqrt{i \cdot l})B$ Bits. □

Nachdem die segmentweise Ermittlung der Anzahl der Goldbach-Partitionen zur Verfügung steht, ergibt sich für ein Intervall $[4, n]$:

Algorithmus 5.6 *segpart*

$n \leftarrow input()$ { Eingabe der Intervallobergrenze }
$l \leftarrow input()$ { Eingabe der Segmentlänge }
$gaps \leftarrow pdiff(sqrt(n))$ { Berechnung der notwendigen Primzahldifferenzen }
$output(part1(l))$ { Berechne erstes Segment mit Basisalgorithmus }
for $i \leftarrow 1$ **to** $n/l - 1$ **do** { Durchlaufe Segmente s_i von $i = 1$ bis $n/l - 1$ }
  $g_i \leftarrow seg1part1(i, l, gaps)$
  $output(g_i)$
end for
return

Satz 5.9 Algorithmus 5.6 bestimmt die Anzahl der Goldbach-Partitionen aller geraden Zahlen zwischen 4 und n in

$$t_{ADL}(n, l) \quad = \quad O\left(n^2 \log \frac{n}{l}\right)$$

Bitoperationen und $s_{ADL}(n) = (B + 1) \cdot l/2 + \pi(\sqrt{n})B + O(1)$ Bits Speicher.

Beweis. Es ist $l = \frac{n}{i}$ und daher ergibt sich die Anzahl der Operationen aus

$$\sum_{i=1}^{\frac{n}{l}} i \cdot l^2 = \sum_{i=1}^{\frac{n}{l}} i \cdot \frac{n^2}{i^2} = n^2 \cdot \sum_{i=1}^{\frac{n}{l}} \frac{1}{i} = n^2 \cdot \log \frac{n}{l}$$

Zur Raumkomplexität: Sieb, Differenzen- und g-Feld werden jeweils wiederverwendet, insgesamt also nur einmal berechnet. □

5.6.1 Diskussion

Durch die Segmentierung ist kein wesentlicher Nachteil entstanden. Die quadratische Laufzeit hat sich zwar erwartungsgemäß nicht verbessert sondern sogar um einen logarithmischen Faktor verschlechtert, der enorme Speicherbedarf von Algorithmus 5.1 ist aber deutlich reduziert, was das Verfahren auch für größere n implementierbar macht.

Der folgende Abschnitt beschreibt die Implementierung und Verteilung von Algorithmus 5.6.

5.7 Implementierung und Verteilung

Die Zeitkomplexitätsaussage zu Algorithmus 5.6 legt zunächst den Schluß nahe, die Segmentlänge l möglichst groß zu wählen. Allerdings ist l durch zwei Faktoren eingeschränkt. Zum einen benötigt 5.6 $l/2 \cdot B$ Bits Speicher. Dabei wurde in den letzten Abschnitten immer vorausgesetzt, daß ein Maschinenwort tatsächlich ausreicht, um die Anzahl der Goldbach-Partitionen einer geraden Zahl zu fassen. Nach der Abschätzung 5.2 gilt dies auf jeden Fall (bei $B = 32$) bis $n \leq 10^9$. Bei einem zur Verfügung stehenden Hauptspeicher von mindestens $8MB$ pro Rechner ergäbe sich ein Ansatzpunkt von etwa $l = 4 \cdot 10^6$.

Allerdings gibt es noch eine zweite wichtige Einschränkung für l. Die Berechnung eines einzelnen Segments s_i beträgt nach 5.8 $O(i \cdot l^2)$ Schritte, wächst also quadratisch mit l. Tatsächlich benötigten die letzten Segmente der Rechnung auf kleinen Maschinen etwa 3 Tage. Nach Abwägung der beiden Bedingungen an l erwies sich ein Wert von $l = 2^5 \cdot 3 \cdot 5 \cdot 7 \cdot 11 = 480480$ als praktikabel, was einer Intervallänge von etwa 960960 pro Segment entspricht. Daraus ergeben sich bis $5 \cdot 10^8$ 521 Segmente, wobei für jedes Segment ein realer Speicheraufwand von nur ca. 2MB notwendig ist. Der Vorteil dieser Wahl war zudem, daß Daten, die auf einem Rechner ohne Netzanbindung errechnet wurden, einfach transportiert werden konnten, da die Größe der jeweils komprimierten Dateien mit etwa 1,4 MB recht genau dem zum Zeitpunkt der Erstellung dieser Arbeit verfügbaren Diskettenspeicherplatz entsprach.

Algorithmus 5.6 wurde auf insgesamt 12 Workstations und 4 PCs verteilt. Die Implementierung erfolgte unter Berücksichtigung der bereits erwähnten Optimierung beim Durchlaufen der gesiebten Segmente, d.h. es wurde byteweise auf Existenz von Primzahlen geprüft. Diese Vorgehensweise lohnt sich, da bereits im Bereich von 10^8 die durchschnittliche Lücke zwischen zwei Primzahlen etwa 20 beträgt, d.h. es können tatsächlich recht viele Bytes übersprungen werden und müssen daher nicht bitweise durchlaufen werden.

5.7.1 Laufzeit

Abbildung 26 zeigt die ADL-Zeitkomplexität von Algorithmus 5.6 bei einer Segmentlänge von $l = 480480$ (in Bitoperationen). Der quadratische Charakter von Algorithmus 5.6 wird daran gut sichtbar.

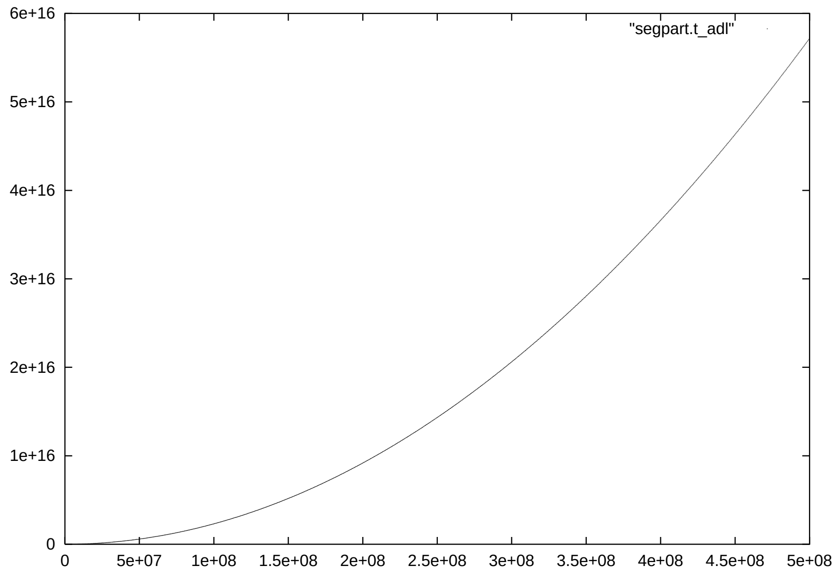


Abbildung 26: ADL-Zeitkomplexität von Algorithmus 5.6

Der Beitrag durch die Funktion S war hier nicht entscheidend. Auch sind keine wesentlichen Multiplikationen oder Divisionen vorhanden. Die ADL-Laufzeiten für die verschiedenen Maschinen war daher im Grunde gleich.

Die reale Gesamtlaufzeit betrug etwa 9 Wochen. Die bisherige Obergrenze von $1{,}28 \cdot 10^8$ war bereits nach etwa 10 Tagen erreicht.

5.8 Ergebnisse

Die neu berechneten Werte sind bis zu dieser Grenze mit den vorhandenen Daten verglichen worden, wobei sich keine Diskrepanzen ergaben.

In diesem Abschnitt werden die Ergebnisse der Berechnung der Anzahl der Goldbach-Partitionen vorgestellt. Dabei zeigen sich zunächst erstaunliche Eigenschaften der Funktion g.

Abbildung 27 zeigt den Verlauf von $g(2n)$ für $2n \leq 10^6$, Abbildung 28 denjenigen von $g(2n)$ nahe $5 \cdot 10^8$.

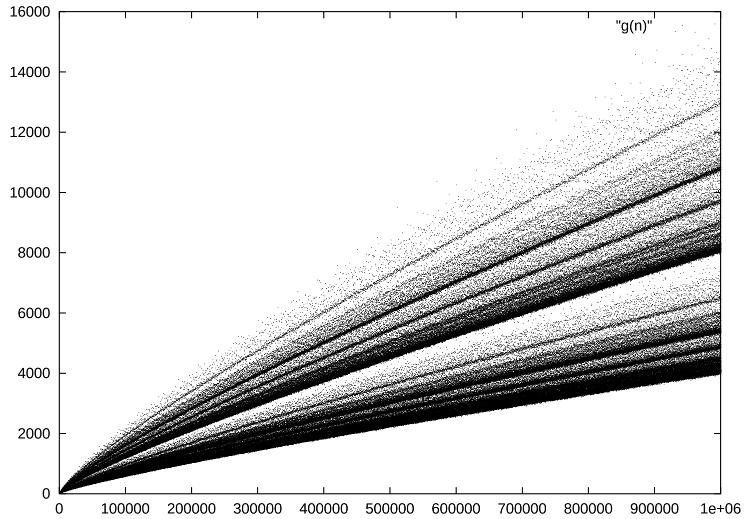


Abbildung 27: $g(2n), 2n \leq 10^6$

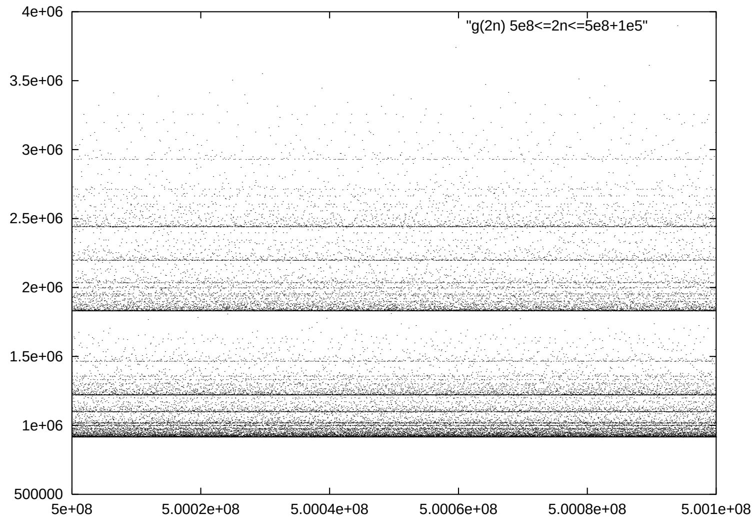


Abbildung 28: $g(2n), 2n \in [5 \cdot 10^8, 5 \cdot 10^8 + 10^5]$

Die deutlichen Konzentrationen in den Abbildungen 27 und 28 korrespondieren mit dem in Abschnitt 5.2 heuristisch begründeten Faktor $\prod_{p|2n}(p-1)/(p-2)$.

Abbildung 29 zeigt die g-Werte der Teilmengen von $[6, 10^6]$, die sich jeweils durch 3 aber nicht durch 5 und 7 bzw. $3 \cdot 5 \cdot 7$ teilen lassen bzw. weder durch 3 noch 5 noch 7 teilbar sind.

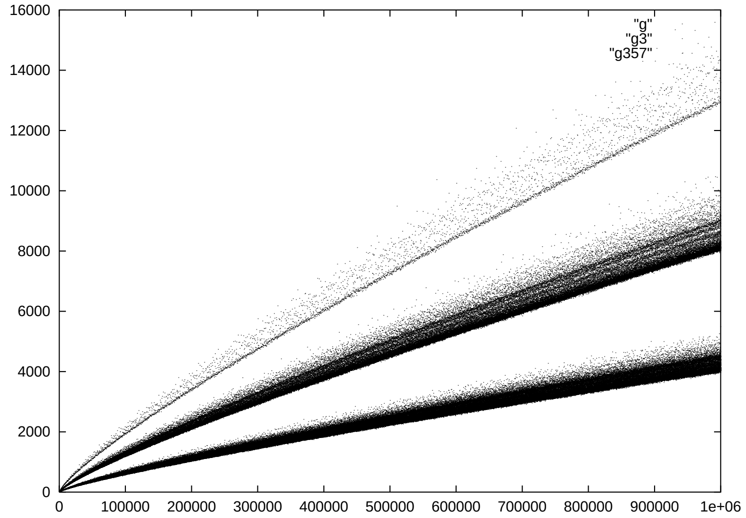


Abbildung 29: g-Werte von Teilmengen aus $[6, 10^6]$

Der maximale Wert von $g(2n)$ im betrachteten Intervall betrug 3977551 und wurde für $2n = 497668710$ $(= 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 17 \cdot 19 \cdot 23 \cdot 29)$ angenommen. Abbildung 30 zeigt die Entwicklung der Maxima der Funktion g, also $\{(2n, g(2n)) : g(2k) < g(2n)\ \forall\ k < n\}$.

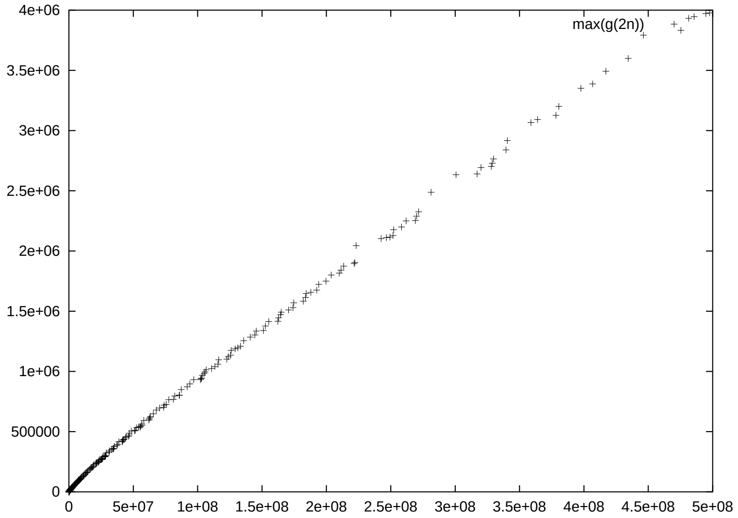


Abbildung 30: Maxima der Funktion g

In [Ste65] wurde vermutet, daß g alle natürlichen Zahlen als Werte annimmt. Die kleinste Zahl, die bis $2n = 5 \cdot 10^8$ nicht angenommen wird, ist 2166940. Abbildung 31 zeigt die Anzahl, wie oft eine Zahl im Intervall $[1, 5 \cdot 10^8]$ von g angenommen wurde. Betrachtet wird also $f(m) = |\{m : g(2n) = m\}|$ mit $m < 500000$.

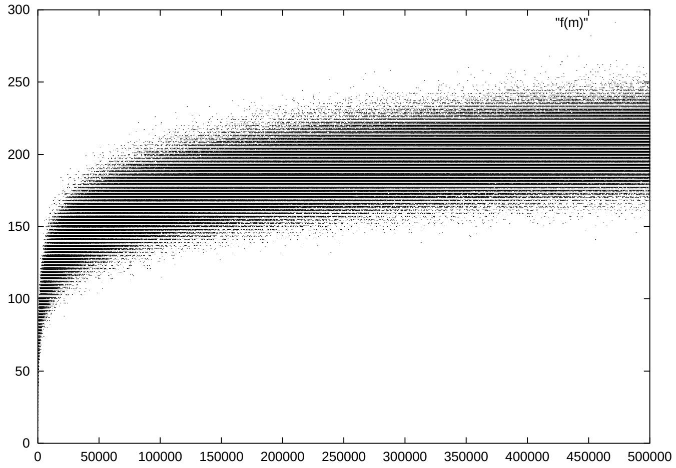


Abbildung 31: Die Funktion $f(m), m < 5 \cdot 10^5$

Es sieht so aus, als wachse $f(m)$, was nicht unbedingt verwunderlich ist, da mit wachsendem n die Möglichkeit, daß $g(2n)$ einen Wert m annimmt, wächst. Nicht offensichtlich ist die Form des notwendigen Abfallens von f, sofern man sich auf ein endliches Intervall beschränkt (experimentell natürlich beschränken muß). Dies ist in Abbildung 32 zu sehen.

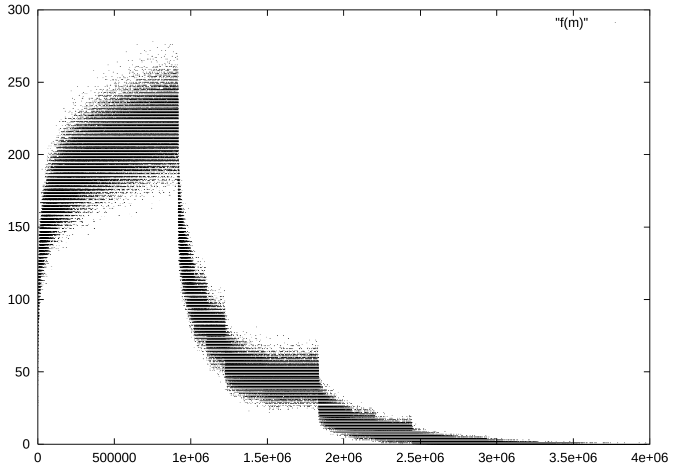


Abbildung 32: Die Funktion $f(m), m < 5 \cdot 10^8$

Die Funktion f ist – soweit bekannt – noch nicht im Speziellen studiert worden.

Es folgt der Vergleich mit den Vermutungen aus Abschnitt 5.2.1. Dazu wurde in Abständen der Länge 10^4 die Entwicklung des arithmetischen Mittels der jeweiligen Funktion g_{XY} bestimmt. Die Selmersche Abschätzung wurde durch numerische Integration der Funktion $1/(\log(n-x) \cdot \log(n+x))$ unter Verwendung des Programpakets *Mathematica* bestimmt, wobei die Annäherung an die singulären Stellen jeweils $x \cdot 10^{-5}$ betrug.

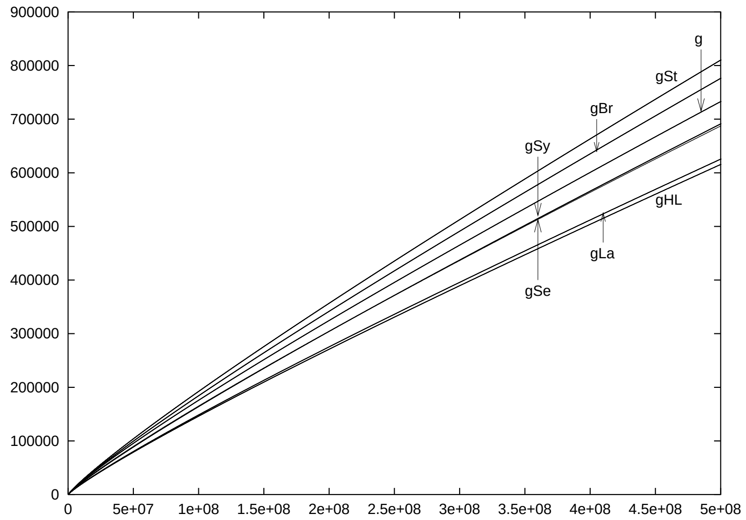


Abbildung 33: Vergleich der Vermutungen mit $g(2n)$

Wie Abbildung 33 zeigt, liegen Brun und Selmer ebenso wie Sylvester relativ gut beim tatsächlichen Wert von g, während die Abweichung von Hardy/Littlewood recht groß ist. Allerdings verhält sich dies beim absoluten Fehler anders. Tabelle 26 zeigt die mittlere, absolute Abweichung der Voraussagen bis $5 \cdot 10^8$ (gemessen in Schritten der Länge 10^5 beginnend bei 6).

Voraussage	Mittlere Abweichung
g_{Se}	349.2
g_{Sy}	4466.8
g_{HL}	71557.4
g_{Br}	89136.4
g_{La}	89383.9
g_{St}	124747.0

Tabelle 26: Mittlere Abweichung der Vermutungen

Hier zeigt sich die Stärke der Selmerschen Voraussage. Aber auch das ja älteste Ergebnis von Sylvester ist den anderen Vermutungen noch deutlich überlegen.

5.9 Ausblick

Kurz vor Fertigstellung dieser Arbeit wurde von Y. Saouter eine Arbeit zur Publikation eingereicht, in der ein neues Verfahren zur Berechnung von $g(2n)$ vorgestellt wird [Sao99]. Die Laufzeit verringert sich dabei deutlich. Allerdings ist der Speicherbedarf noch zu hoch, jedoch nicht aussichtslos, so daß eine Implementierung in wenigen Jahren möglich scheint.

6 Fermatsche Quotienten

6.1 Einführung

In diesem Kapitel werden Berechnungen zu *Fermatschen Quotienten* vorgestellt. Im ersten Abschnitt wird der mathematische Hintergrund zur Problemstellung beleuchtet. Es folgt ein Überblick über historische Berechnungen. Ausgehend von einem Basisalgorithmus wird dann ein Verfahren vorgestellt und weiterentwickelt, dessen 64-Bit-optimierte Implementierung schließlich auf mehrere Maschinen verteilt wurde.

Das resultierende Programm zeigt eine praktische Grenze für die idealisierte Annahme des in Kapitel 2 definierten Berechnungsmodells *RX-RAM* auf, daß sämtliche Rechnungen innerhalb der Register stattfinden. In diesem Fall reicht die Anzahl der Register genau aus. Das Resultat dieser verteilten Rechnung stellt eine umfangreiche Erweiterung früherer Arbeiten dar. Die während der Rechnungen gefundenen Lösungen werden in Abschnitt 6.7.2 vorgestellt.

Hier nun zunächst die für dieses Kapitel entscheidende

Definition 6.1 (Fermatscher Quotient) Es sei p eine ungerade Primzahl und $a \in \mathbb{Z}$ mit $p \not| a$. Dann heißt

$$q_p(a) = \frac{a^{p-1} - 1}{p} \tag{2}$$

Fermatscher Quotient von p zur Basis a.

6.2 Mathematischer Hintergrund

Im Jahre 1828 stellte *Niels Abel* [Abe28] die folgende Aufgabe:

„*Kann $\alpha^{\mu-1} - 1$, wenn μ eine Primzahl und α eine ganze Zahl und kleiner als μ und größer als 1 ist, durch μ^2 theilbar sein?*“

Noch im gleichen Band [Jac28] gibt *Carl Jacobi* die ersten Lösungsbeispiele an:

μ	α
11	3, 9
29	14
37	18

Beispiel 6.2 Es gilt $3^5 = 243 = 2 \cdot 11^2 + 1$ und damit $3^{10} = (3^5)^2 \equiv 1 \mod 11^2$.

Die Kongruenz $a^{p-1} \equiv 1 \mod p^2$ steht in folgendem Zusammenhang mit (2):

Satz 6.3

$$q_p(a) \equiv 0 \; mod \; p \Longleftrightarrow a^{p-1} \equiv 1 \; mod \; p^2$$

Beweis. $p \mid q_p(a) = \frac{a^{p-1}-1}{p} \iff \frac{a^{p-1}-1}{p} = k \cdot p \iff a^{p-1} - 1 = k \cdot p^2 \iff a^{p-1} \equiv 1 \mod p^2$.
für ein $k \in \mathbb{N}$. □

Es folgen zunächst ein paar wichtige Eigenschaften Fermatscher Quotienten (siehe auch [Rib91] oder [Bri71]):

Satz 6.4 $q_p(a) \in \mathbb{Z}$.

Beweis. Da p a nach Voraussetzung nicht teilt, folgt mit dem kleinen Fermatschen Satz, daß $a^{p-1} \equiv 1 \mod p$, also teilt p $a^{p-1} - 1$. □

Satz 6.5 $q_p(a) \equiv 0 \mod p \Longleftrightarrow a^{\frac{p-1}{2}} \equiv \pm 1 \mod p^2$

Beweis. „$\Longrightarrow$“:
$q_p(a) \equiv 0 \mod p \iff a^{p-1}-1 = k \cdot p^2 \implies (a^{\frac{p-1}{2}}-1)(a^{\frac{p-1}{2}}+1) = k \cdot p^2 \implies p^2 \mid a^{\frac{p-1}{2}}-1$ oder $+1$, da $p \mid a^{\frac{p-1}{2}} - 1$ und $p \mid a^{\frac{p-1}{2}} + 1$ nicht beide teilen kann (sonst müßte p auch die Summe $a^{\frac{p-1}{2}}$ teilen).
„$\Longleftarrow$“:
Da $p^2 \mid a^{\frac{p-1}{2}} + 1$ oder $p^2 \mid a^{\frac{p-1}{2}} - 1$, teilt p^2 auch das Produkt $a^{p-1} - 1$. □

Satz 6.6 Es gelte $p \not| \, ab$, $a, b \in \mathbb{Z}$. Dann: $q_p(a \cdot b) = q_p(a) + q_p(b) \mod p$

Beweis. Wegen $p \not| \, ab$ und Fermat ist $a^{p-1} - 1 = mp$ und $b^{p-1} - 1 = np$, $m, n \in \mathbb{Z}$. Daraus $(a^{p-1}-1)(b^{p-1}-1) = mnp^2 \implies (ab)^{p-1}+1 = a^{p-1}+b^{p-1}+mnp^2 \implies q_p(ab) = \frac{(ab)^{p-1}}{p} = \frac{a^{p-1}-1}{p} + \frac{b^{p-1}-1}{p} + mnp = q_p(a) + q_p(b) + kp$ mit $k = mn \in \mathbb{Z}$ □

Fermat-Quotienten verhalten sich also „Logarithmus-ähnlich“. Aus der Kenntnis der Reste für prime Basen lassen sich sofort die Reste für zerlegbare Basen (und daher ggfs. auch Lösungen von $q_p(n) \equiv 0 \mod p$) ermitteln.

Satz 6.7 Es sei $p \not| \, n$. Dann gilt:
$q_p(a^n) \equiv 0 \mod p \iff q_p(a) \equiv 0 \mod p$.

Beweis. siehe z.B. [Bri71] □

Im folgenden werden nur prime Basen a und Exponenten p betrachtet.

6.2.1 Der Zusammenhang mit Fermats letztem Satz

Fermat-Quotienten erlangten besonderes Interesse, nachdem im Jahre 1909 *Wieferich* den folgenden Satz bewies:

Satz 6.8 (Wieferich) Es sei p eine Primzahl, $a, b, c \in \mathbb{Z}$ und $p \nmid abc$. Dann gilt:

$$a^p + b^p = c^p \implies 2^{p-1} \equiv 1 \mod p^2$$

Beweis. siehe [Wie09] □

Die Bedingung $p \nmid abc$ und p prim wird als 1. Fall von Fermats letztem Satz bezeichnet. Lösungen von $2^{p-1} \equiv 1 \mod p^2$ heißen auch *Wieferich*-Primzahlen. Die einzigen Wieferich-Primzahlen, die man bis heute kennt, sind 1093 und 3511 (siehe dazu auch [Rib83], [Mei13], [Bee22]). Bis $4 \cdot 10^{12}$ konnte keine weitere gefunden werden ([Cra97]). Eine zum Zeitpunkt der Erstellung dieser Arbeit noch nicht veröffentliche Rechnung ([Zim99b]) lief bis $2 \cdot 10^{13}$ (ohne weiteres Ergebnis).

Es ist heute bekannt, daß die Aussage des Wieferichschen Satzes neben der 2 für alle primen Basen $a \leq 89$ gilt [Rib96]. Mit dem Beweis von Fermats letztem Satz ist dieser Zusammenhang zwar nicht mehr von so entscheidender Bedeutung. Nach wie vor ist aber über die Lösungen von $a^{p-1} \equiv 1 \mod p^2$ bzw. ihrer Struktur wenig bekannt.

6.2.2 Offene Probleme

Die folgenden Probleme im Zusammenhang mit Fermatschen Quotienten sind ungelöst:

1. Gibt es zu einem gegebenen $a \geq 2$ unendlich viele Primzahlen p mit
$$a^{p-1} \equiv 1 \mod p^2 \qquad ?$$
2. Gibt es zu einem gegebenen $a \geq 2$ unendlich viele Primzahlen p mit
$$a^{p-1} \not\equiv 1 \mod p^2 \qquad ?$$
3. Gibt es zu jedem $p > 2$ ein $2 \leq a < p$ mit
$$a^{p-1} \equiv 1 \mod p^2 \qquad ?$$

Einige quantitative Abschätzungen zu diesen Aussagen sind z.B. in [Rib91] zu finden.

Im folgenden Abschnitt werden zunächst historische Berechnungen zum Problem aufgelistet.

6.3 Historische Berechnungen

Seitdem Jacobi die ersten Lösungen präsentierte, sind sehr viele Arbeiten entstanden, die verschiedene Bereiche für a und p betrachteten. Tabelle 27 gibt dazu einen Überblick. Die letzte große Rechnung wurde dabei von *Wilfrid Keller* [Kel97] durchgeführt und umfaßte die Intervalle $a < 1000$ und $p < 8 \cdot 10^9$ (a, p prim) sowie $p < 2^{38}$ für $a = 3$ und $a = 5$.

Name	Jahr	$a \in$	$p \in$
Jacobi, Busch	1828	$\{1, \dots, 36\}$	[3,37]
Desmarest	1852	$\{10\}$	$\{2,997\}$
Hertzer	1908	$\{2, \dots, p\}$ $\{2, \dots, \sqrt{p}\}$	[3,307] [307,751]
Grawe	1909	$\{2\}$	[3,1000]
Cunningham	1910	$\{2\}$	[3,1000]
Tarry	1911	$\{2\}$	[3,1009]
Meissner	1913	$\{2\}$	[3,2000]
Beeger	1914	$\{2, \dots, \sqrt{p}\}$	[3,199]
Beeger	1922	$\{2\}$	[3,14000]
Beeger	1940	$\{2\}$	[3,16000]
Fröberg	1958	$\{2\}$	[3,50000]
Kravitz	1960	$\{2\}$	[3,100000]
Pearson	1963	$\{2\}$	[3,200183]
Hausner, Sachs	1963	$\{2\}$	[3,1000000]
Brillhart, Tonascia, Weinberger	1963	$\{2, \dots, 10\}$	$[3, 2^{22}]$
Riesel	1964	$\{2, \dots, 10\}$ $\{11, \dots, 150\}$	[3,500000] [3,10000]
Kloss	1965	$\{2,3,5,7, \dots, 43\}$	zwischen $5 - 31 \cdot 10^6$
Sachs	1965	$\{2,3,5,6,7,10, \dots, 99\}$	[3,1000000]
Brillhart, Tonascia, Weinberger	1971	$\{3,5,6,7,10, \dots, 99\}$	versch. $2^k, k \in \{25, \dots, 30\}$, $3 \cdot 10^9$ für $a = 2$
Lehmer	1981	$\{2\}$	$[3, 6 \cdot 10^9]$
Aaltonen, Inkeri	1991	$\{3,5,7, \dots, 997\}$	$[3, 10^4]$
Montgomery	1991	$\{2, \dots, 99\}$	$[3, 2^{32}]$
Crandall, Dilcher, Pomerance	1996	$\{2\}$	$[3, 4 \cdot 10^{12}]$
Ernvall, Metsänkylä	1997	$[2, p-1]$	$[2, 10^6]$
Keller	1997	$\{3,5, \dots, 997\}$	$[3, 8 \cdot 10^9]$

Tabelle 27: Historische Berechnungen

6.4 Ein Algorithmus zur Berechnung von $a^{p-1} \mod p^2$

In diesem Abschnitt wird zunächst ein einfacher Algorithmus zur Berechnung des Restes von a^{p-1} nach Division durch p^2 vorgestellt, der allerdings in der Praxis gewisse Nachteile aufweist. Mit einigen – relativ aufwendigen – Erweiterungen werden die auftretenden Probleme dann gelöst.

6.4.1 Basisalgorithmus

Algorithmus 6.1 *fermatb*

```
amax ← input()   { Eingabe der größten Basis }
pmax ← input()   { Eingabe des größten Exponenten }
l ← input()   { Eingabe der Segmentlänge }
l_½ ← l ≫ 1   { Ermittle halbe Segmentlänge }
gaps ← pdiff(sqrt(pmax))   { Ermittle Primzahldifferenzen bis √pmax }
i ← 0
while true do   { Durchlaufe {1, ... ,pmax} segmentweise }
  segment ← pg_sieve(gaps, i, l)   { Siebe Segment }
  cntl ← 0   { Initialisiere Zähler für segment }
  while cntl < l_½ do   { Durchlaufe segment }
    if getBit(segment, cntl) = 0 then   { Falls Primzahl gefunden }
      pmin1 ← i × l + 2 × cntl   { Berechne p − 1 }
      p ← pmin1 + 1   { Berechne p }
      if p > pmax then   { Intervall verlassen? }
        return
      end if
      sqrp ← p × p   { Quadriere p }
      pmin1div2 ← pmin1 ≫ 1   { Berechne (p − 1)/2 }
      cnta ← 0   { Initialisiere Zähler für Basisfeld }
      a ← aprimes[cnta]   { Erste Basis }
      while a ≤ amax do   { Durchlaufe Basen aprimes }
        res ← binexpmod(a, pmin1div2, sqrp)   { Berechne a^((p−1)/2) mod p^2 }
        if res = −1 or res = 1 then   { Lösung gefunden? }
          output(a)   { Ausgabe der Basis }
          output(p)   { Ausgabe des Exponenten + 1 }
        end if
        cnta ← cnta + 1   { Erhöhe Zähler für aprimes }
        a ← aprimes[cnta]   { Bestimme nächste Basis }
      end while
    end if
    cntl ← cntl + 1   { Erhöhe Zähler für segment }
  end while
  i ← i + 1   { Erhöhe Segmentnummer }
end while
return
```

Das Primzahlfeld *aprimes* sei dabei als konstantes Feld kleiner Primzahlen gegeben. Dies stellt in der Praxis keine Einschränkung dar, da man gewöhnlich relativ kleine Basen betrachtet. Im Falle größerer Basen ließe sich beispielsweise das erste gesiebte Segment zur Bestimmung der notwendigen a nutzen.

Der Algorithmus zur binären, modularen Exponentiation *binexpmod* ist wie folgt definiert:

Algorithmus 6.2 *binexpmod*

1: $a \leftarrow input()$ { Eingabe der Basis }
2: $pmin1div2 \leftarrow input()$ { Eingabe des Exponenten }
3: $sqrp \leftarrow input()$ { Eingabe des Moduls }
4: $n \leftarrow a$ { Initialisiere Resultat }
5: $b \leftarrow 1 \ll (B-1)$ { Setze b auf höchstwertiges Bit }
6: **while** $b \wedge pmin1div2 = 0$ **do** { Justiere b auf höchstwertiges Bit im Exponenten }
7:     $b \leftarrow b \gg 1$ { Nächstniedrigstwertiges Bit }
8: **end while**
9: **while** $b \neq 0$ **do** { Durchlaufe Bits }
10:     $n \leftarrow (n \times n)\%sqrp$ { Berechne $n^2 \mod p^2$ }
11:     **if** $b \wedge pmin1div2$ **then** { Falls Bit in $pmin1div2$ gesetzt }
12:         $n \leftarrow (n \times a)\%sqrp$ { Berechne $a \cdot n \mod p^2$ }
13:     **end if**
14:     $b \leftarrow b \gg 1$ { Nächstniedrigstwertiges Bit }
15: **end while**
16: **return** n

Lemma 6.9 Es sei $p < 2^{\frac{B}{4}}$ und $a \leq p$. Algorithmus 6.2 berechnet den Rest von $a^{\frac{p-1}{2}}$ nach Division durch p^2 in

$$\begin{aligned} t_{ADL}(p) &= 31B + 13B \cdot (B - log_2 p) + \\ &\quad +(16B + 2 \cdot D(B) + 2 \cdot M(B)) \cdot \log_2 p \\ &= 13B^2 + 31B + (3B + 2 \cdot D(B) + 2 \cdot M(B)) \cdot \log_2 p \\ &= O(\log p) \end{aligned}$$

Bitoperationen unter Verwendung von $s_{ADL}(p) = 7B$ Bits Speicher.

Beweis. Korrektheit: siehe z.B. [Rie94]. Die Voraussetzung für p ist entscheidend, da ansonsten Überläufe entstehen (siehe auch 6.4.2). Zur Zeitkomplexität: Die erste **while**-Schleife wird $(B - \log_2 p)$-mal durchlaufen, die zweite beginnend in Zeile 9 benötigt $\log_2 p$ Schritte. □

Satz 6.10 Algorithmus 6.1 ermittelt die Lösungen der Kongruenz $a^{p-1} \equiv 1 \mod p^2$ für alle $a \leq a_{max}$ und $p \leq p_{max}$ in

$$t_{ADL}(a_{max}, p_{max}) \quad = \quad O\left(\frac{p_{max} \cdot a_{max}}{\log a_{max}}\right)$$

Bitoperationen unter Verwendung von $s_{ADL}(a_{max}, p_{max}) = \pi(a_{max})B + l/2 + \pi(\sqrt{p_{max}})B + O(1)$ Bits Speicher.

Beweis. Zur Korrektheit: Die Primzahlen werden segmentweise gesiebt, für jedes gefundene p werden sämtliche a-Basen durchlaufen und die Reste von $a^{(p-1)/2} \mod p^2$ in *binexpmod* ermittelt. In Zeile 23 wird wegen 6.5 auf Rest 1 oder -1 geprüft.
Der **if**-Rumpf in den Zeilen 11–30 wird $\pi(p_{max})$-mal durchlaufen. Die **while**-Schleife in 21–29 jeweils $\pi(a_{max})$-mal. Der Beitrag ohne *binexpmod* im **while**-Rumpf beträgt $(3 \cdot M(B) + 37B + S(\pi(a_{max})/2) + (26B + S(\pi(a_{max})/2)) \cdot \pi(a_{max})) \cdot \pi(p_{max}) = O(\log \pi(a_{max}) \cdot \pi(p_{max}) \cdot \pi(a_{max})) = O(\frac{p_{max} \cdot a_{max}}{\log p_{max}})$ Operationen. Hinzu kommen $O(p_{max} \log p_{max} \log\log p_{max})$ Operationen durch das Sieben. *binexpmod* wird für jedes p $\pi(a_{max})$-mal ausgeführt. Es ergeben sich dafür

$$\pi(a_{max}) \cdot \sum_{p \leq p_{max}} \left(13B^2 + 31B + (3B + 2 \cdot D(B) + 2 \cdot M(B)) \cdot \log_2 p\right) =$$

$$\pi(a_{max}) \cdot \pi(p_{max}) \cdot (13B^2 + 31B) + \pi(a_{max}) \cdot \frac{3B + 2 \cdot D(B) + 2 \cdot M(B)}{\log 2} \cdot \sum_{p \leq p_{max}} \log p =$$

$$O(\pi(a_{max})\pi(p_{max})) + O(\pi(a_{max}) \cdot p_{max}) =$$

$$O\left(\frac{p_{max} \cdot a_{max}}{\log a_{max}}\right)$$

Operationen. Wegen $a_{max} < p_{max}$ folgt die Aussage des Satzes. Zur Speicherkomplexität: Benötigt wird neben dem Platz für die Basen ein Segment der Länge $l/2$ sowie die Primzahldifferenzen für Primzahlen kleiner gleich $\sqrt{p_{max}}$. □

Die im Prinzip einzige Optimierungsmöglichkeit liegt in der Verbesserung des Faktors vor $\sum_{p \leq p_{max}} \log p$.

Dazu werden im folgenden die aus *binexpmod* stammenden Operationen – zumindest was die Divisionen angeht – noch weitgehend eliminiert. Neben der Tatsache, daß multiplikative Operationen im allgemeinen ungünstig sind, ergibt sich hier noch ein Problem, auf das nun eingegangen wird.

6.4.2 Diskussion

Algorithmus 6.1 unter Verwendung von 6.2 besitzt einen großen Nachteil in der Praxis. Die Zwischenergebnisse der Rechnung in 6.2 haben vor Reduktion modulo p^2 in Zeile 10 die Größenordnung p^4, in Zeile 12 immer noch $a \cdot p^2$. Für $p > 2^{32} \approx 4 \cdot 10^9$ ergeben sich Zahlen größer als 2^{128}. Die Maschinenwortlängen der Beispielmaschinen betragen jedoch nur 32 Bits (SPARCstation-4) bzw. 64 Bits (Ultra-1), d.h. es sind bis zu 5 Maschinenworte notwendig, um eine einzige Zahl darzustellen, was bereits einfache Operationen teuer macht.

Ein weiterer schwerwiegender Nachteil sind die auftretenden Divisionen in Form der Reduktion modulo p^2, die nicht nur an sich sehr teuer sind, sondern zudem wieder eine Rechnung mit überlangen Zahlen erfordern.

Eine Vereinfachung – zumindest was die Handhabung überlanger Zahlen angeht – ist die Verwendung einer geeigneten Langzahlarithmetik-Software. Das Paket *gmp* (GNU Multiple Precision Package) der Free Software Foundation ([Gra95]) eignet sich hier hervorragend. Tatsächlich basierte das für die 32-Bit-Maschinen entwickelte Programm auf der *gmp*-Routine `mpz_powm`, die im Prinzip Algorithmus 6.2 implementiert. Die wesentliche Sequenz der *gmp*-Routinen ist im Abschnitt 6.6 aufgelistet.

Die Verwendung einer solchen Software sollte natürlich nicht darüber hinwegtäuschen, daß das Problem dadurch nur verschoben wird. Die Rechnung an sich wird nicht weniger aufwendig oder schneller. Die relativ teuren Langzahloperationen führten schließlich dazu, daß der Anteil der kleinen Rechner in der verteilten Rechnung insgesamt nur etwa 20% betrug.

Im folgenden Abschnitt werden die Probleme von 6.2 beseitigt. Dabei wird eine auf das noch vorzustellende Zielintervall zugeschnittene 64-Bit-Routine entwickelt.

6.5 Ein modulares, fast divisionsfreies 64-Bit-Verfahren

Das hier vorgestellte Verfahren basiert auf [Cra97], wo eine umfangreiche, letztendlich erfolglose Suche nach Wieferich-Primzahlen stattfand. Im folgenden wird dieses Verfahren für beliebige Basen adaptiert und für die Zielintervalle optimiert. Divisionen überlanger Zahlen werden dabei vermieden.

6.5.1 Zahldarstellung zur Basis p

Ein wichtiger Schritt zur Verkleinerung der auftretenden, überlangen Zahlen besteht in der Zahldarstellung zur Basis p modulo p^2: Dazu bezeichne im folgenden ein Tupel (x, y) eine Zahl $n = x + yp \mod p^2$ (mit $x, y < p$).

Satz 6.11 Es sei $n = (x, y)$. Dann gilt

$$a \cdot n \mod p^2 = (ax \mod p, (ay + \lfloor ax/p \rfloor) \mod p) \quad \text{sowie}$$

$$n^2 \mod p^2 = (x^2 \mod p, (2xy + \lfloor x^2/p \rfloor) \mod p)$$

Beweis. 1. Es ist $an = ax + ayp = ax \mod p + \lfloor ax/p \rfloor p + ayp$. Also wird $an \mod p^2 = (ax \mod p, (ay + \lfloor (ax)/p \rfloor) \mod p)$.
2. $n^2 \mod p^2 = (x + yp)^2 = x^2 + 2xyp + y^2p^2 = x^2 \mod p + \lfloor x^2/p \rfloor p + 2xyp + y^2p^2$. Daraus folgt $n^2 \mod p^2 = (x^2 \mod p, (2xy + \lfloor x^2/p \rfloor) \mod p)$. □

Der Vorteil ist offensichtlich: Die Zwischenresultate bleiben für $p \leq 2^{63}$ kleiner als 2^{128}, d.h. es wird nur noch eine Arithmetik für doppelt lange Zahlen benötigt. Allerdings sind nach wie vor Divisionen durch p im Spiel. Im nächsten Abschnitt wird jedoch gezeigt, wie auch diese vermieden werden können.

6.5.2 Vermeidung von Divisionen

Das nachstehend beschriebene Vorgehen ersetzt Divisionen einer Zahl $0 \leq n < p^2$ durch p durch eine Multiplikation und einfache Shiftoperationen. Dabei wird ein einziges Mal ein Quotient berechnet und dieser wiederholt eingesetzt. Das Verfahren ist dann günstig, wenn Quotienten n/p für viele verschiedene n bei konstantem p berechnet werden müssen. Genau diese Situation ist aber in 6.1 gegeben, für jedes p werden viele Basen a durchlaufen.

Als Vorbereitung:

Satz 6.12 Es sei $r = \lfloor 2^s/p \rfloor$ mit $2^s > p^2$. Dann gilt

$$\lfloor n/p \rfloor = \lfloor rn/2^s \rfloor \text{ oder } \lfloor rn/2^s \rfloor + 1$$

Beweis. Es ist

$$\left\lfloor \frac{\left\lfloor \frac{2^s}{p} \right\rfloor n}{2^s} \right\rfloor = \left\lfloor \frac{\left(\frac{2^s}{p} - \delta \right) n}{2^s} \right\rfloor = \left\lfloor \frac{n}{p} - \frac{n\delta}{2^s} \right\rfloor, 0 \leq \delta < 1$$

Nun wird wegen $2^s > p^2$ und $0 \leq \delta < 1$ die rechte Seite entweder gleich $\lfloor n/p \rfloor$ oder gleich $\lfloor n/p \rfloor - 1$. □

Es folgt nun der Algorithmus zur Berechnung der Quadrate $n^2 \mod p^2$.

Algorithmus 6.3 *sqrmodp*

$x \leftarrow \mathit{input}()$ { Eingabe des Restes $\mod p$ }
$y \leftarrow \mathit{input}()$ { Eingabe des Restes $\mod p^2$ }
$r \leftarrow \mathit{input}()$ { Eingabe des Quotienten $\lfloor 2^s/p \rfloor$ }
$s \leftarrow \mathit{input}()$ { Eingabe von s }
$p \leftarrow \mathit{input}()$ { Eingabe von p }
$y \leftarrow x \times y$ { Berechne $x \cdot y$ }
$\mathit{tmp} \leftarrow (y \times r) \gg s$ { $\mathit{tmp} = \lfloor x \cdot y/p \rfloor$ oder $\mathit{tmp} = \lfloor x \cdot y/p \rfloor - 1$ }
$y \leftarrow y - p \times \mathit{tmp}$ { $y = x \cdot y \mod p$ oder $y = x \cdot y \mod p + p$ }
if $y \geq p$ **then** { Falls $+p$ }
  $y \leftarrow y - p$ { $y = x \cdot y \mod p$ }
end if
$x \leftarrow x \times x$ { Quadriere x }
$\mathit{tmp} \leftarrow x \times r \gg s$ { $\mathit{tmp} = \lfloor x^2/p \rfloor$ oder $\mathit{tmp} = \lfloor x^2/p \rfloor - 1$ }
$x \leftarrow x - p \times \mathit{tmp}$ { $x = x^2 \mod p$ oder $x = x^2 \mod p + p$ }
if $x \geq p$ **then** { Falls $+p$ }
  $x \leftarrow x - p$ { $x = x^2 \mod p$ }
  $\mathit{tmp} \leftarrow \mathit{tmp} + 1$ { Merke Überlauf }
end if
$y \leftarrow y + y + \mathit{tmp}$ { Addiere (ggfs. mit Überlauf) }
while $y \geq p$ **do** { Reduziere ggfs. y }
  $y \leftarrow y - p$ { $y = (2xy + \lfloor x^2/p \rfloor) \mod p$ }
end while
$\mathit{output}(x)$ { Ausgabe von x }
$\mathit{output}(y)$ { Ausgabe von y }
return

6.5.3 Modulares Quadrieren ohne Division

Algorithmus 6.3 berechnet für ein $n = (x, y)$ das Quadrat $n^2 \mod p^2$.

Das Verfahren lohnt sich natürlich nicht mehr, wenn hinreichend lange Maschinenworte zur Verfügung stehen.

Bevor zwei Algorithmen *slog2p* sowie *twosdivp* zur Berechnung der Parameter s und r vorgestellt werden, hier zunächst die Festlegung der Zielparameter für die spätere Implementierung:

Nach der Rechnung von Keller [Kel97] wurde das Ziel $a < 1000$, $p < 10^{11}$ gesetzt. Es ist $10^{11} < 2^{37}$, was im folgenden noch eine wichtige Rolle spielen wird. Darüber hinaus wird unter Rücksichtnahme auf spätere Operationen eine weitere Einschränkung vorgenommen: Es sei $p > 2^{32}$, woraus folgt, daß $s > 64$. Wegen $s \leq 2\log_2 p + 2$ wird $s \leq 74$, insgesamt also $64 < s < 75$. Es gilt somit auch: $2^{27} < r < 2^{43}$.

Es wird nun $s > 2\log_2 p$ (nahe bei $2\log_2 p$) berechnet:

Algorithmus 6.4 *slog2p*

$p \leftarrow input()$ { Eingabe von p }
$cnt \leftarrow 1$ { Beginne Suche nach erstem Bit in p bei 2^{B-1} }
$tmp \leftarrow 1 \ll (B-1)$
while $tmp \wedge p = 0$ **do** { Solange erstes Bit nicht gefunden }
  $cnt \leftarrow cnt + 1$ { Erhöhe cnt }
  $tmp \leftarrow tmp \gg 1$ { Halbiere tmp }
end while
$s \leftarrow (B - cnt) \ll 1$ { $s \geq 2\log_2 p$ }
return $s + 1$ { Rückgabe von $s > 2\log_2 p$ }

Lemma 6.13 Es sei $p < 2^B$. Algorithmus 6.4 berechnet $2\log_2 p < s < 2\log_2 p + 2$ in

$$t_{ADL}(p) = 29B + 15B \cdot (B - (\log_2 p))$$

Bitoperationen unter Verwendung von $s_{ADL}(p) = 6B$ Bits Speicher. Es treten dabei keine Zwischenresultate größer gleich 2^B auf.

Beweis. Korrektheit: Vom höchstwertigen Bit eines Maschinenwortes wird nach dem höchstwertigen Bit in p gesucht und dabei gezählt. Nun hat p^2 maximal die doppelte Anzahl an Bits. Um zu garantieren, daß wirklich $s > 2\log_2 p$ gilt, wird 1 addiert. Die **while**-Schleife wird dabei $B - \log_2 p$-mal durchlaufen. $\square$

Algorithmus 6.5 berechnet $r = \lfloor 2^s/p \rfloor$. Satz 6.12 beinhaltet noch ein Problem: Für $2^s > B$ reicht ein Maschinenwort für die Rechnung nicht aus. Dieses Problem kann aber umgangen werden. In Algorithmus 6.5 muß dabei der Parameter k so gewählt werden, daß $0 < s - k < B$, also $s - B < k < s$ gilt.

Lemma 6.14 Algorithmus 6.5 berechnet den Quotienten $r = \lfloor 2^s/p \rfloor$ in

$$t_{ADL}(p, s) \quad = \quad 51B + 2 \cdot D(B) + M(B)$$

Bitoperationen unter Verwendung von $s_{ADL}(p, s) = 6B$ Bits Speicher. Dabei treten keine Zwischenergebnisse größer gleich 2^B auf.

Beweis. Die Korrektheit ist den Kommentaren zu entnehmen, wobei z.B. mit $B = 64$ und $k = 20$ wegen $s < 75$ alle Zwischenergebnisse kleiner 2^{64} sind. Darüber hinaus gilt wegen $s > 64$: $s - k > 0$. Bleibt: Nach Zeile 10 ist $r = \left\lfloor \frac{2^s}{p} - \left\lfloor \frac{2^{s-k}}{p} \right\rfloor \cdot 2^k p \right\rfloor + \left\lfloor \frac{2^{s-k}}{p} \right\rfloor \cdot 2^k$ $= \left\lfloor \left\lfloor \frac{2^s}{p} \right\rfloor + t - \left\lfloor \frac{2^{s-k}}{p} \right\rfloor \cdot 2^k p \right\rfloor + \left\lfloor \frac{2^{s-k}}{p} \right\rfloor \cdot 2^k$, wobei $0 \leq t < 1$ einziger nichtganzzahliger Anteil ist. Daher wird $r = \left\lfloor \frac{2^s}{p} \right\rfloor$. Es sind dabei zwei Divisionen und eine Multiplikation

Algorithmus 6.5 *twosdivp*

$p \leftarrow input()$ $\{$ Eingabe von p $\}$
$s \leftarrow input()$ $\{$ Eingabe von s $\}$
$r \leftarrow 1 \ll (s-k)$ $\{ r = 2^{s-k} \}$
$r \leftarrow r \div p$ $\{ r = \left\lfloor \frac{2^{s-k}}{p} \right\rfloor \}$
$tmp \leftarrow r \ll k$ $\{ tmp = \left\lfloor \frac{2^{s-k}}{p} \right\rfloor \cdot 2^k \}$
$r \leftarrow r \times p$ $\{ r = \left\lfloor \frac{2^{s-k}}{p} \right\rfloor \cdot p \}$
$r \leftarrow (1 \ll (s-k)) - r$ $\{ r = 2^{s-k} - \left\lfloor \frac{2^{s-k}}{p} \right\rfloor \cdot p \}$
$r \leftarrow r \ll k$ $\{ r = \left(2^{s-k} - \left\lfloor \frac{2^{s-k}}{p} \right\rfloor \cdot p\right) \cdot 2^k \}$
$r \leftarrow r \div p$ $\{ r = \left\lfloor \frac{\left(2^{s-k} - \left\lfloor \frac{2^{s-k}}{p} \right\rfloor \cdot p\right) \cdot 2^k}{p} \right\rfloor \}$
$r \leftarrow r + tmp$ $\{ r = \left\lfloor \frac{\left(2^{s-k} - \left\lfloor \frac{2^{s-k}}{p} \right\rfloor \cdot p\right) \cdot 2^k}{p} \right\rfloor + \left\lfloor \frac{2^{s-k}}{p} \right\rfloor \cdot 2^k \}$
return r

nötig. Durch eine direkte Implementierung in *RX-RAM*-Maschineninstruktionen wird die Speicherkomplexität deutlich. □

Es verbleibt das Problem der überlangen Zwischenergebnisse in den Zeilen 6, 7, 12 und 13 von Algorithmus 6.3: Für $x \cdot y$ in Zeile 6 gilt $0 \leq x \cdot y \leq 2^{75}$. In Zeile 7 bleibt daher wegen $r < 2^{43}$: $y \cdot r < 2^{118}$, insbesondere also kleiner 2^{128}. Selbiges gilt für die Zeilen 12 und 13. Trotz der Tatsache, daß die Ergebnisse der Multiplikationen zwei Maschinenworte nicht übertreffen, können dabei in der Praxis Probleme auftreten. So rechnet die SPARC V9 64-Bit-Instruktion `mulx` modulo 2^{64}, berechnet also die höherwertigen Bits nicht. Ein Ausweg aus dem Dilemma ist die Aufspaltung in 32-Bit-Teilworte, was jedoch zusätzlich Zeit kostet. Allerdings kann dabei im Falle der Berechnung in Algorithmus 6.3 nach dem in Abbildung 34 dargestellten Schema zumindest eine Operation eingespart werden.

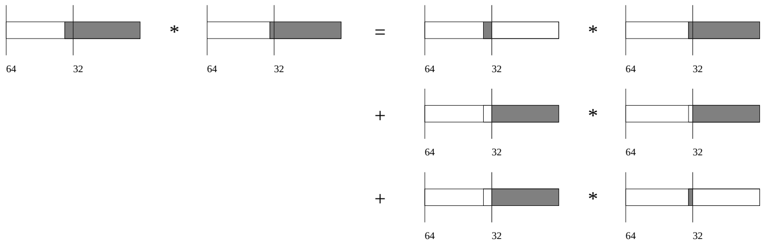


Abbildung 34: 64-Bit-Multiplikation für *sqrmodp*

Die Teilworte werden multipliziert, dann werden die Teilergebnisse addiert. Relevant sind wegen der Operation $\gg s$ in 6.3 nur die Beiträge größer gleich 2^{64}.

Satz 6.15 Algorithmus 6.3 berechnet $n^2 \mod p^2$ in $t_{ADL}(n,p) = 77B + 6 \cdot M(B)$ Bitoperationen unter Verwendung von $s_{ADL}(n,p) = 7B$ Bits Speicher.

Beweis. Die Korrektheit ergibt sich aus den Kommentaren. Die **while**-Schleife wird dabei maximal dreimal ausgeführt. □

Mit den nun geschaffenen Hilfsmitteln muß Algorithmus 6.2 nur noch folgendermaßen modifiziert werden:

Algorithmus 6.6 *binexpmod2*

```
1:  a ← input()   { Eingabe der Basis }
2:  pmin1div2 ← input()   { Eingabe des Exponenten }
3:  p ← input()
4:  r ← input()   { Eingabe von ⌊2^s/p⌋ }
5:  s ← input()   { Eingabe von s > 2 log_2 p }
6:  x ← a   { Initialisiere n = (x, y) }
7:  y ← 0
8:  b ← 1 ≪ (B − 1)   { Setze b auf höchstwertiges Bit }
9:  while b ∧ pmin1div2 = 0 do   { Justiere b auf höchstwertiges Bit im Exponenten }
10:     b ← b ≫ 1
11: end while
12: while b ≠ 0 do   { Durchlaufe Bits }
13:     (x, y) ← sqrmodp(x, y, r, s, p)   { (x, y) ⇐ (x, y)^2 mod p^2 }
14:     if b ∧ pmin1div2 then   { Falls Bit in pmin1div2 gesetzt, berechne a · n mod p^2 }
15:         tmp ← a × x
16:         x ← tmp%p
17:         y ← (a × y + tmp ÷ p)%p
18:     end if
19:     b ← b ≫ 1
20: end while
21: output(x)
22: output(y)
23: return
```

Lemma 6.16 Algorithmus 6.6 berechnet den Rest von $a^{\frac{p-1}{2}}$ nach Division durch p^2 in

$$t_{ADL}(p) \quad = \quad 13B^2 + 66B + (86B + 8 \cdot M(B) + 3 \cdot D(B)) \cdot log_2 p \;=\; O(\log p)$$

Bitoperationen unter Verwendung von $s_{ADL}(p) = 7B$ Bits Speicher.

Beweis. Die Korrektheit ergibt sich aus der Korrektheit von 6.2 sowie 6.3. In den Zeilen 14 – 18 wird die möglicherweise notwendige Operation $a \cdot n \mod p^2$ ausgeführt. Dabei treten wegen $a < 2^{10}$ keine überlangen Zwischenresultate auf, so daß sich eine Vorgehensweise analog Algorithmus 6.3 nicht lohnt. Der Zeitaufwand ergibt sich aus der Ersetzung der Quadrierung in Zeile 10 von 6.2 durch den Aufruf von 6.3 sowie der Ersetzung der elementaren Berechnung von $a \cdot n \mod p^2$ (Zeile 12 in 6.2) durch die Zeilen 14–18. □

6.5.4 Algorithmus *fermatadv*

Nachdem alle notwendigen Funktionen zur Verfügung stehen, kann Algorithmus 6.1 zu *fermatadv* verändert werden:

Algorithmus 6.7 *fermatadv*

```
amax ← input()   { Eingabe der größten Basis }
pmax ← input()   { Eingabe des größten Exponenten }
l ← input()   { Eingabe der Segmentlänge }
l_{1/2} ← l ≫ 1   { Ermittle halbe Segmentlänge }
gaps ← pdiff(sqrt(pmax))   { Ermittle Primzahldifferenzen bis √pmax }
i ← 0   { Segmentnummer }
while true do   { Durchlaufe {1, ... ,pmax} segmentweise }
  segment ← pg_sieve(gaps, i, l)   { Siebe Segment }
  cntl ← 0   { Initialisiere Zähler für segment }
  while cntl < l_{1/2} do   { Durchlaufe segment }
    if getBit(segment, cntl) = 0 then   { Falls Primzahl gefunden }
      pmin1 ← i × l + 2 × cntl   { Berechne p − 1 }
      p ← pmin1 + 1   { Berechne p }
      if p > pmax then   { Intervall verlassen? }
        return
      end if
      r ← twosdivp(slog2p(p))
      pmin1div2 ← pmin1 ≫ 1   { Berechne (p − 1)/2 }
      cnta ← 0   { Initialisiere Zähler für Basisfeld }
      a ← aprimes[cnta]   { Erste Basis }
      while a ≤ amax do   { Durchlaufe Basen aprimes }
        (x, y) ← binexpmod2(a, pmin1div2, p, r, s)   { Berechne a^{(p−1)/2} mod p^2 }
        if x = 1 and y = 0 or x = pmin1 and y = pmin1 then   { Lösung ? }
          output(a)   { Ausgabe der Basis }
          output(p)   { Ausgabe des Exponenten + 1 }
        end if
        cnta ← cnta + 1   { Erhöhe Zähler für aprimes }
        a ← aprimes[cnta]   { Bestimme nächste Basis }
      end while
    end if
    cntl ← cntl + 1   { Erhöhe Zähler für segment }
  end while
  i ← i + 1   { Erhöhe Segmentnummer }
end while
return
```

Satz 6.17 Algorithmus 6.7 ermittelt die Lösungen der Kongruenz $a^{p-1} \equiv 1 \mod p^2$ für alle $a \leq a_{max}$ und $2^{B/2} < p \leq p_{max}$ mit $a_{max} \cdot p_{max} < 2^B$ in

$$t_{ADL}(a_{max}, p_{max}) = O\left(\frac{p_{max} \cdot a_{max}}{\log a_{max}}\right)$$

Bitoperationen unter Verwendung von $s_{ADL}(a_{max}, p_{max}) = \pi(a_{max})B + l/2 + \pi(\sqrt{p_{max}})B + O(1)$ Bits Speicher.

Beweis. Die Korrektheit folgt aus der Korrektheit der Algorithmen 6.1 und 6.6 sowie der Tatsache, daß für $n = (x, y)(= x + yp \mod p^2)$ die Zahl $(1, 0)$ gleichbedeutend mit 1 und $(p-1, p-1) = p - 1 + p^2 - p \mod p^2$ mit $-1 = p - 1 \mod p$ übereinstimmt. Zur Zeitkomplexität: An der asymptotischen Laufzeit hat sich nichts geändert. In Zeile 17 muß für jedes $p < p_{max}$ jeweils ein Aufruf von Algorithmus 6.3 und 6.4 berechnet werden, zusammen $\sum_{p \leq p_{max}} (80B + 15B \cdot (B - \log_2 p) + 2 \cdot D(B) + M(B)) = O(\frac{p_{max}}{\log p_{max}})$. In Zeile 22 wird 6.2 durch 6.6 ersetzt. In Zeile 23 kommen zwei Operationen hinzu. Insgesamt ergibt die wesentliche **while**-Schleife in den Zeilen 21 – 29

$$\pi(a_{max}) \cdot \sum_{p \leq p_{max}} \big(13B^2 + 107B + (86B + 8 \cdot M(B) + 3 \cdot D(B)) \cdot \log_2 p + S(\pi(a_{max})/2)\big) =$$

$$\pi(a_{max}) \cdot \pi(p_{max}) \cdot (13B^2 + 107B + S(\pi(a_{max}/2))) +$$

$$+ (86B + 8 \cdot M(B) + 3 \cdot D(B)) \cdot \sum_{p \leq p_{max}} \log_2 p =$$

$$O(\pi(a_{max}) \cdot \pi(p_{max}) \cdot \log \pi(a_{max})) + O(\pi(a_{max}) \cdot p_{max}) =$$

$$O\left(\frac{a_{max} \cdot p_{max}}{\log a_{max}}\right)$$

Bitoperationen. Die Speicherkomplexität ergibt sich analog Algorithmus 6.1. □

Der Vorteil von Algorithmus 6.7 gegenüber 6.1 liegt nicht etwa in der besseren Zeitkomplexität, sondern in der Fähigkeit, auch große Exponenten verarbeiten zu können.

Ein Laufzeitvergleich zwischen 6.7 und 6.1 findet im nächsten Abschnitt statt.

6.6 Implementierung und Verteilung

Die Verteilung der Gesamtrechnung erfolgte auf 7 Ultra-1, 6 SPARCstation-4 sowie 2 PCs. Mangels 64-Bit-Registerlängen wurde auf den SPARCstation-4 sowie den PCs Algorithmus 6.1 implementiert, während 6.7 auf den Ultra-1 zum Einsatz kam. Ein Problem stellte dabei die fehlende Unterstützung des Sun-C-Compilers für die langen Register dar. Auch aufgrund dieser Tatsache wurde eine Assembler-Routine implementiert, die von einem C-Programm aufgerufen wurde. Die Anzahl der Register für die eigentliche Rechnung (ohne Zugriffe auf die Felder für Basis und Exponenten) reichte dabei exakt aus, ohne auf Speicher zurückgreifen zu müssen.

Die folgende Sequenz von *gmp*-Routinen wurde für jedes p und a ausgeführt und stellte die Basis für die Rechnung auf den 32-Bit-Maschinen dar:

```
mpz_set_ui (lp_1, (unsigned long) (p >> 32));
mpz_mul_2exp (lp_1, lp_1, 32);
mpz_add_ui (lp_1, lp_1, ((unsigned long) (p & 0x00000000ffffffffULL)));
mpz_set (lp2, lp_1);
mpz_mul (lp2,lp2,lp2);
mpz_sub_ui (lp2_1,lp2,1UL);
mpz_sub_ui (lp_1,lp_1,1UL);
mpz_set (lp_1_2,lp_1);
mpz_tdiv_q_2exp (lp_1_2, lp_1_2, 1UL);
mpz_set_ui (la, a );
mpz_powm (lr, la, lp_1_2, lp2);
if ((mpz_cmp_ui (lr, 1UL) == 0) || (mpz_cmp(lr, lp2_1) == 0))
{
   sprintf(Msg,"solution: %llu",p);
   LogEntry(Msg);
}
```

Optimierungsversuche wie Vermeidung von Division und Darstellung mod p ergaben auf den 32-Bit-Maschinen keine Verbesserung. Die Intervalle wurden gemäß Tabelle 28 verteilt:

$a \in$	$p \in$	Rechnertyp
$[2, 1000]$	$[3, 2^{32}]$	PC
$[2, 200]$	$[2^{32}, 10^{11}]$	SPARCstation-4/PC
$[200, 1000]$	$[2^{32}, 10^{11}]$	Ultra-1

Tabelle 28: Intervall-Aufteilung Fermatquotienten

Die Berechnung der kleinen Exponenten erfolgte auf den 32-Bit-Maschinen, da Algorithmus 6.7 keine Exponenten kleiner gleich 2^{32} verarbeiten kann. Die restliche Zeit wurde dann für die Basen kleiner 200 verwendet.

6.6.1 Laufzeit

Es ergaben sich überraschend große Unterschiede auf den verschiedenen Maschinen. Während 6.1 auf den PCs sehr schnell lief, zeigten sich deutliche Schwächen der Workstations. Dies ist wohl vor allem auf die unterschiedliche *gmp*-Implementierung zurückzuführen.

Tabelle 29 zeigt die Laufzeiten für die Intervalle $a < 168$, $p \in [5 \cdot 10^{10}, 5 \cdot 10^{10} + k]$ für $k = 10^4, 10^5$. Im Vergleich zeigt die zweite Zeile die Laufzeit von 6.7.

Rechner	$k = 10^4$	$k = 10^5$
Ultra-1 6.1	39,1	386,0
Ultra-1 6.7	6,7	66,7
SPARCstation-4 6.1	45,3	442,2
PC 6.1	11,3	110,9

Tabelle 29: Laufzeiten Fermatquotient (CPU-Sekunden)

Es ist bei 6.7 wesentlich, die Intervalle in der richtigen Reihenfolge zu durchlaufen, um unnötige Aufrufe von 6.4 sowie 6.5 einzusparen. Die Sieblänge und der Wert des Parameters `MUL_3_5_THRESHOLD` wurden dabei gemäß den in Kapitel 3 ermittelten Optima gewählt. Abbildung 35 zeigt die Entwicklung der ADL-Zeitkomplexität in den betrachteten Intervallen ($a < 1000, p < 10^{11}$ in ADL-Bitoperationen). Die reale Gesamtlaufzeit betrug ca. 18 Wochen.

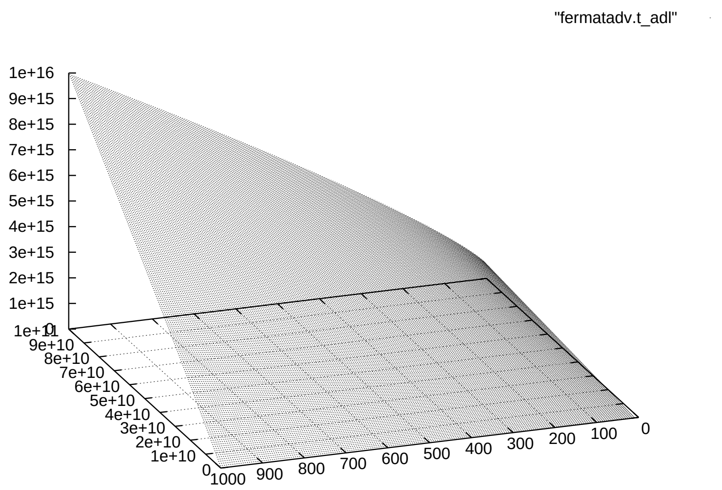


Abbildung 35: ADL-Zeitkomplexität Fermatquotienten

6.7 Ergebnisse

6.7.1 Betrachtung zur erwarteten Lösungsanzahl

Zunächst eine heuristische Betrachtung zur Anzahl der Paare (a, p) mit $a^{p-1} \equiv 1 \mod p^2$ im Intervall $[10^{10}, 10^{11}]$: Bei zufälliger Verteilung der Reste von $q_p(a) \mod p$ erwartet man für ein festes a für jeden p-ten Exponenten eine Lösung, also mit A.1 für jedes a

$$\sum_{10^{10}<p<10^{11}} \frac{1}{p} \approx \log\log 10^{11} - \log\log 10^{10} \approx 0,09531 \quad \text{Lösungen.}$$

Bei $\pi(1000) = 168$ verschiedenen Basen a ergibt sich eine erwartete Anzahl von 16,012.

6.7.2 Lösungen

Tabelle 30 zeigt die Lösungen mit $p > 10^{10}$, die neu ermittelt werden konnten.

a	p	a	p
23	15546404183	397	13315373041
37	76407520781	499	24117560837
97	76704103313	613	18419352383
151	15697215641	619	52649183399
239	12502228667	647	15266862761
269	65684482177	691	10843045487
271	12145092821	881	94626144313
353	17283818861	929	62199604679

Tabelle 30: Lösungen (a, p) mit $p > 10^{10}$

Zwischen $8 \cdot 10^9$ und 10^{10} ergab sich keine weitere Lösung. Das Paar $(929, 62199604679)$ stellt die erste Lösung für $a = 929$ dar. Keines der neu ermittelten Lösungspaare (a, p) erfüllte zusätzlich $p^{a-1} \equiv 1 \mod a^2$. Dies ist natürlich nicht von vornherein ausgeschlossen. W. Keller fand die Lösung $5^{188748146800} \equiv 1 \mod 188748146801$, es gilt auch $188748146801^4 \equiv 1 \mod 5^2$.
Keine der neuen Lösungen erfüllte zusätzlich die Kongruenz $a^{p-1} \equiv 1 \mod p^3$. Die zusätzliche Erfüllung einer der beiden weiteren Eigenschaften war natürlich auch sehr unwahrscheinlich.

Die Lösungen aus Tabelle 30 finden sich in Tabelle 31 der Vollständigkeit halber wieder. Die neuen Lösungen mit $p > 8 \cdot 10^9$ aus Tabelle 30 sind mit einem * gekennzeichnet. Kellers Lösungspaar $(5, 188748146801)$ wurde ebenso aufgenommen.

a	p	a	p
2	1093 3511	439	31 79 170899693
3	11 1006003	443	5 3406223
5	20771 40487 53471161 1645333507 6692367337 188748146801	449	3 5 1789
7	5 491531	457	5 11 919 1589513
11	71	461	1697 5081
13	863 1747591	463	1667
17	3 46021 48947	467	3 29 743 7393
19	3 7 13 43 137 63061489	479	47 2833 500239
23	13 2481757 13703077 15546404183*	487	3 11 23 41 1069
29		491	7 79 661763933
31	7 79 6451 2806861	499	5 109 81307 24117560837*
37	3 77867 76407520781*	503	3 17 229 659 6761
41	29 1025273 138200401	509	7 41 7215975149
43	5 103	521	3 7 31 53 8938997
47		523	3 9907 19289
53	3 47 59 97	541	3
59	2777	547	31 1691778551
61		557	3 5 7 23 39829
67	7 47 268573	563	18920521
71	3 47 331	569	7 263 25359067
73	3	571	23 29 308383
79	7 263 3037 1012573 60312841	577	3 13 17 71 1381277
83	4871 13691 315746063	587	7 13 31 22091 6343317671
89	3 13	593	3 5
97	7 2914393 76704103313*	599	5 35771
101	5 1050139	601	5 61
103	24490789	607	5 7 40303229
107	3 5 97 613181	613	3 4073 81371669 18419352383*
109	3 20252173	617	101 1087 6007
113		619	7 73 11682481 52649183399*
127	3 19 907 13778951	631	3 1787 5741
131	17 754480919	641	43 24481
137	29 59 6733 18951271 4483681903	643	5 17 307 859 460609 7354807
139		647	3 23 15266862761*
149	5 29573 121456243 2283131621	653	13 17 19 1381 22171 637699
151	5 2251 14107 5288341 15697215641*	659	23 131 2221 9161 65983
157	5 122327 4242923 5857727461	661	441583073
163	3 3898031	673	61
167	64661497	677	13 211
173	3079 56087	683	3 1279
179	3 35059 126443	691	37 509 1091 9157 84131 10843045487*
181	3 101	701	3 5
191	13 379133	709	17 199 1663
193	5 4877	719	3 41 4414200313
197	3 7 653 6237773	727	11
199	3 5 77263 1843757	733	17
211	279311	739	3 9719 5681059
223	71 349	743	5
227	7 40277	751	5 151 409
229	31	757	3 5 17 71 242789
233	3 11 157 86735239	761	41 907
239	11 13 74047 212855197 361552687 12502228667*	769	1305827821
241	11 523 1163 35407	773	3 787711 26259199 142719149
251	3 5 11 17 421 395696461	787	37 41 427541
257	5 359 49559 648258371	797	8273 14607661
263	7 23 251 267541 159838801	809	3 59 448110371
269	3 11 83 8779 65684482177*	811	3 211
271	3 168629 16774141 235558417 12145092821*	821	19 83 233 293 1229 37871 209140301
277	1993	823	13 2309
281	3443059	827	3 17 29 9323
283	46301	829	3 17
293	5 7 19 83	839	5227 11840951
307	3 5 19 487	853	1125407
311		857	5 41 157 1697 32478247
313	7 41 149 181 1259389	859	71
317	107 349 2227301	863	3 7 23 467 12049
331	211 359 6134718817	877	78926821
337	13 30137417	881	3 7 23 22385723 94626144313*
347		883	3 7
349	5 197 433 7499	887	11 607 60623
353	8123 465989 17283818861*	907	5 17 3497891
359	3 23 307 24350087	911	127 318917
367	43 2213	919	3
373	7 113	929	62199604679*
379	3	937	3 41 113 853 22343 500861 1031299 258469889
383	28067251	941	11 1499
389	19 373 29569 211850543	947	5021
397	3 279421 13315373041*	953	3 513405611
401	5 83 347 115849	967	11 19 4813 44830663
409	34583 1894600969	971	3 11 401 9257 401839 7672759
419	173 349 983 3257 22891217	977	11 17 109 239 401 37589
421	101 1483 350677	983	
431	3 2393 12755833	991	3 13 431 26437
433	3 129497 244403	997	197 1223

Tabelle 31: Lösungen (a, p), $a < 1000$, $p < 10^{11}$

Abbildung 36 zeigt die Lösungen aus Tabelle 31 graphisch.

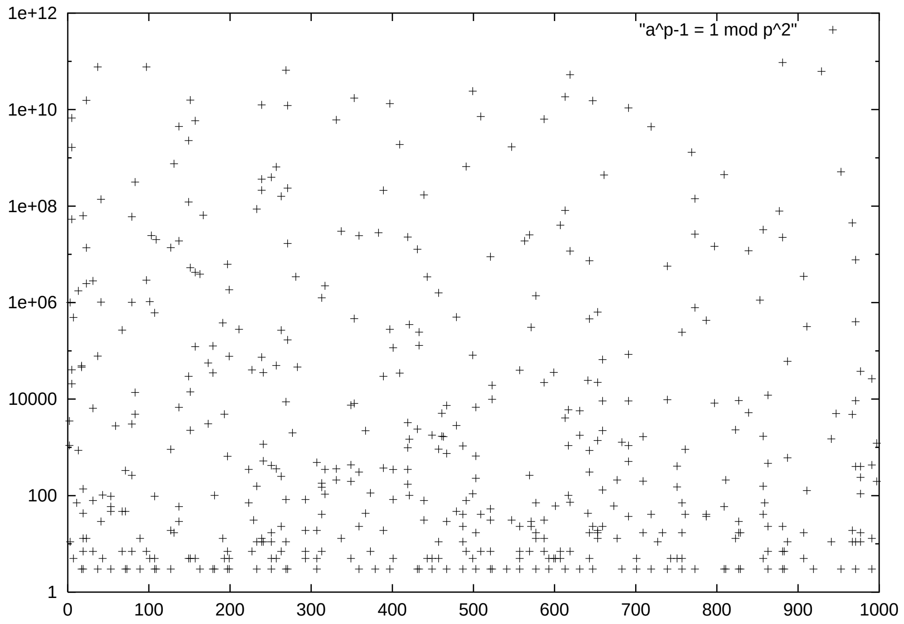


Abbildung 36: Lösungen (a, p), $a < 1000$, $p < 10^{11}$

Abbildung 37 zeigt die Entwicklung der Anzahl der Lösungspaare (a, p).

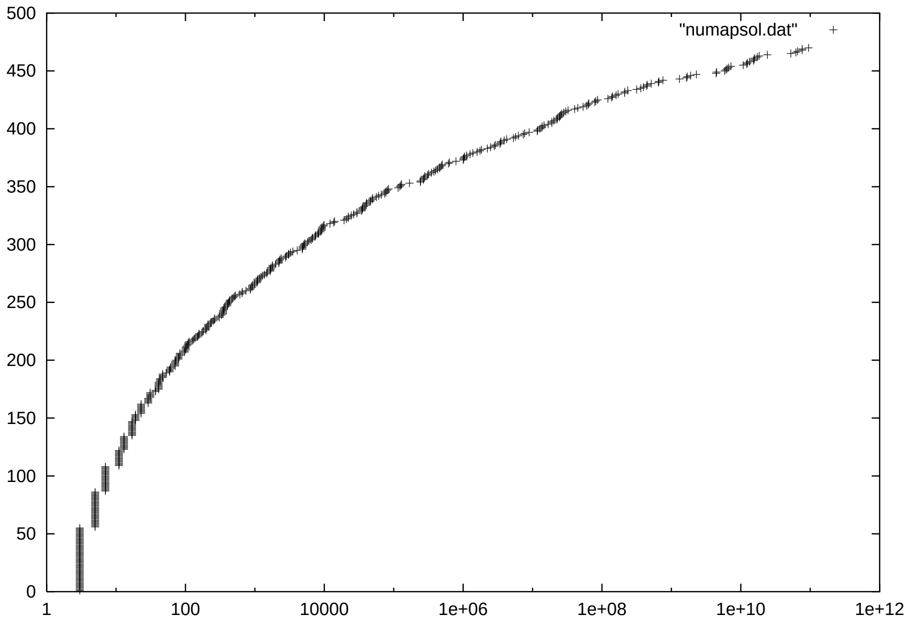


Abbildung 37: Entwicklung der Lösungsanzahl für $a < 1000$, $p < 10^{11}$

7 Verteilung

7.1 Einführung

Sämtliche in dieser Arbeit betrachteten Probleme erforderten einen Rechenaufwand, der für einzelne Maschinen nicht in vernünftiger Zeit hätte bewältigt werden können. Durchweg wären zum Erreichen der gesteckten Ziele mehrere Jahre bis zur Vollendung nötig gewesen. Durch eine Aufteilung der jeweiligen Gesamtintervalle auf mehrere Rechner konnte diese Zeit auf 4,5 Monate begrenzt werden.

Dieses Kapitel stellt ein System vor, das die Verteilung der Intervalle und die Kontrolle der laufenden Rechnung bis hin zu fast vollständiger Automatisierung steuerte. Ziel war es dabei, die einzelnen, an der Gesamtrechnung beteiligten Maschinen möglichst unabhängig voneinander zu belassen. Der Einsatz eines speziellen, zentralen Steuerungsprozesses wurde bewußt vermieden.

Für die Realisierung der einzelnen Rechnungen stand zunächst der Workstation-Cluster des Instituts für Informatik zur Verfügung, später kamen noch mehrere PCs an verschiedenen Standorten hinzu.

Um sämtliche zur Verfügung stehenden Rechnerkapazitäten nutzen zu können, darf keine Beschränkung auf eine spezielle Konfiguration stattfinden. Im günstigsten Fall sind alle Systeme vernetzt und verfügen über einen gemeinsamen Festplattenspeicher. Allerdings trifft dies eben nicht immer zu. Auch Rechner ganz ohne Netzanbindung bzw. solche im Netz ohne gemeinsamen Speicherzugriff wurden eingesetzt. Verschiedene Rechner können dabei unter Umständen auch verschiedenen Nutzungseinschränkungen unterworfen sein. Etwa kann eine stärker belastete Server-Maschine nur nachts und am Wochenende genutzt werden, andere zwar auch zu normalen Arbeitszeiten, aber mit möglicherweise geringerer Priorität der Rechnung gegenüber konkurrierenden Prozessen.

Es folgt nun eine Beschreibung der allgemeinen Vorgehensweise. Auf die einzelnen Punkte wird danach näher eingegangen. In Abschnitt 7.3 wird die Implementierung beschrieben, durch die eine möglichst optimale Verteilung und automatische Kontrolle auch unter heterogenen Bedingungen erzielt wird.

7.2 Prinzipielle Vorgehensweise

Die verschiedenen Rechnungen folgten demselben Grundprinzip:

1. Möglichst günstige Aufspaltung des Gesamtintervalls in Teilintervalle
2. Möglichst günstige Verteilung des Ergebnisses von 1. auf die verschiedenen Rechnersysteme
3. Zu gewissen Zeitpunkten automatischer Start eines dezentralen Kontrollprogramms auf jedem beteiligten Rechner
4. Starten des Programms zur eigentlichen Berechnung durch das Kontrollprogramm
5. Automatisierte regelmäßige Selbstkontrolle und entsprechende Reaktion
6. Fehlertoleranz bei Ausfall eines Rechners und ggfs. Benutzer-Benachrichtigung
7. Abschließende Sammlung der Teilergebnisse

Die einzelnen Punkte sollen nun näher beschrieben werden.

7.2.1 Aufspaltung der Gesamtintervalle

Das größte Intervall trat während der Berechnung zur Verifikation der Goldbachschen Vermutung (siehe Kapitel 4) auf und umfaßte alle natürlichen geraden Zahlen bis $4 \cdot 10^{14}$. Die Rechnung zur Anzahl der Goldbach-Partitionen wurde bis $5 \cdot 10^8$ durchgeführt. Modulo p verschwindende Fermatquotienten wurden für alle Basen $a < 1000$ und $p < 10^{11}$ gesucht. Alle behandelten Probleme wurden dabei so implementiert, daß Teilrechnungen unabhängig voneinander laufen konnten, also keine Kommunikation stattfinden mußte. Erst das abschließende Sammeln zur Gesamtinformation erforderte eine Synchronisation, also ein Warten auf Beendigung der letzten, noch ausstehenden Teilrechnung. Das Grundziel war dabei die Minimierung der Gesamtlaufzeit. Zur optimalen Aufteilung des Gesamtintervalls ist auf folgende, nicht unabhängige Punkte zu achten:

1. Unterschiedliche Rechnerleistung
2. Minimierung von Plattenzugriffen
3. Minimierung der Anzahl entstehender Logbuchdateien
4. Einschränkungen durch Bedingungen der Rechnung selbst
5. Minimierung der Zeit zur Nachbearbeitung beim Aufsetzen nach temporärem Rechnerausfall

Eine Minimierung der Gesamtrechenzeit bei gleichzeitiger Minimierung der Anzahl der Logbuchdateien erfolgte durch Abschätzung des maximal möglichen Zeitverlustes bei Einsatz des langsamsten Rechners für das letzte Teilintervall. Hierbei mußte insbesondere darauf geachtet werden, daß Rechner, die außerhalb jedes Netzzugriffes rechneten, im voraus weder zu große noch zu kleine Teilabschnitte erhielten, um spätere manuelle Neueinteilungen zu vermeiden. In der Praxis wurde die Intervallgröße jeweils experimentell bestimmt, der resultierende Verlust betrug dabei maximal wenige Tage.

Das Maximum der Anzahl entstandener Logbuchdateien war 500 und entstand während der Berechnung der Anzahl der Goldbach-Partitionen. Diese Zahl kann zwar auch schon nicht mehr als übersichtlich bezeichnet werden, die Dateien wurden allerdings wiederum durch ein Programm ausgewertet, so daß eine manuelle Kontrolle nicht notwendig war. Der Grund für das Auftreten dieses Maximums war die Tatsache, daß jedes Teilintervall von Goldbach-Partitionen zunächst im Hauptspeicher abgebildet werden mußte und daher in seiner Größe beschränkt war, was wiederum die Gesamtanzahl mitbestimmte. Eine Größenordnung von etwa einigen hundert Teilintervallen zeigte sich bei einer maximal verwendeten Rechneranzahl von 17 als sehr praktikabel. Die im einzelnen verwendete Anzahl hängt natürlich sehr von der eigentlichen Rechnung ab, das Optimum kann daher variieren.

Mit temporären Ausfällen von Maschinen muß zu jeder Zeit gerechnet werden. Da die Teilintervalle relativ groß sein können, mußten jeweils Aufsatzpunkte zwischengespeichert werden. Die Minimierung der Nacharbeitungszeit nach temporären Ausfällen wurde im wesentlichen durch die einzelnen Programme selbst erreicht. Dabei wurden die Zwischenergebnisse der Rechnung in der eigentlichen Logdatei verzeichnet, während eine zweite Datei den kompletten nächsten Programmaufruf mit den korrekten Aufsatzpunkten als Kommandozeilenparameter enthielt. Das möglicherweise nötige Aufsetzen wurde dabei durch einfaches Lesen des Inhalts der zweiten Datei durch das in Abschnitt 7.3.1 vorgestellte Kontrollprogramm realisiert.

7.2.2 Verteilung der Teilintervalle

Eine möglichst optimale Verteilung der Teilintervalle auf die verschiedenen Rechnersysteme erfolgte durch Abschätzung der zu erwartenden Gesamtleistung der einzelnen Maschinen. Ein ganz wesentlicher Vorteil ergab sich dabei durch die Möglichkeit des gemeinsamen Plattenzugriffs durch die meisten verwendeten Maschinen. Eine einzige große Datei mit Teilintervallen wurde von sämtlichen Rechnern im Cluster genutzt. Die Optimierung ergibt sich dabei praktisch automatisch. Eine Ausnahme war die Berechnung zu den Fermatquotienten, die aufgrund der verschiedenen Algorithmen je nach Architektur aufgeteilt werden mußte. Ein Nachteil des gemeinsam genutzten Plattenspeichers ist einerseits die Notwendigkeit der Implementierung eines Lock-Mechanismus, andererseits die Anfälligkeit gegenüber Ausfällen des Servers. Eine spezielle Zuteilung von Teilintervallen muß also nur für Rechner vorgenommen werden, für die kein gemeinsamer Speicherzugriff möglich war.

Auf eine eigentlich naheliegende Implementierung eines Protokolls zur Verteilung einzelner Intervalle über ein Netz nach dem „Farmer/Worker"-Prinzip wurde bewußt verzichtet, da diese Vorgehensweise bei Ausfall des Netzes oder des Rechners mit dem „Farmer"-Prozeß sämtliche Rechnungen blockiert hätte.

Im folgenden Abschnitt wird eine Software vorgestellt, die eine dezentrale Steuerung für verteilte Rechnungen implementiert und dabei den erwähnten Anforderungen genügt.

7.3 Das Steuerprogramm-Paket `dcnth`

`dcnth` (für `d`*istributed* `c`*omputational* `n`*umber* `t`*heory*) ist eine Zusammenfassung verschiedener Teilprogramme zur Steuerung und Überwachung einer verteilten Rechnung.

`dcnth` ist ein Unix-basiertes, portables System, daß aus einer Sammlung von (Korn-) Shellprogrammen besteht, die auf jeder an einer Rechnung beteiligten Maschine zur Verfügung gestellt wird. Das Verfahren selbst ist im Grunde betriebssystemunabhängig. Zum Zeitpunkt der Erstellung dieser Arbeit war jedoch das Betriebssystem Unix mit seiner Stabilität und seinen Fähigkeiten konkurrenzlos, so daß im Grunde keine Alternative bestand. Die Implementierung erfolgte daher Unix-basiert.

Abbildung 38 zeigt eine von `dcnth` unterstützte Beispielkonfiguration.

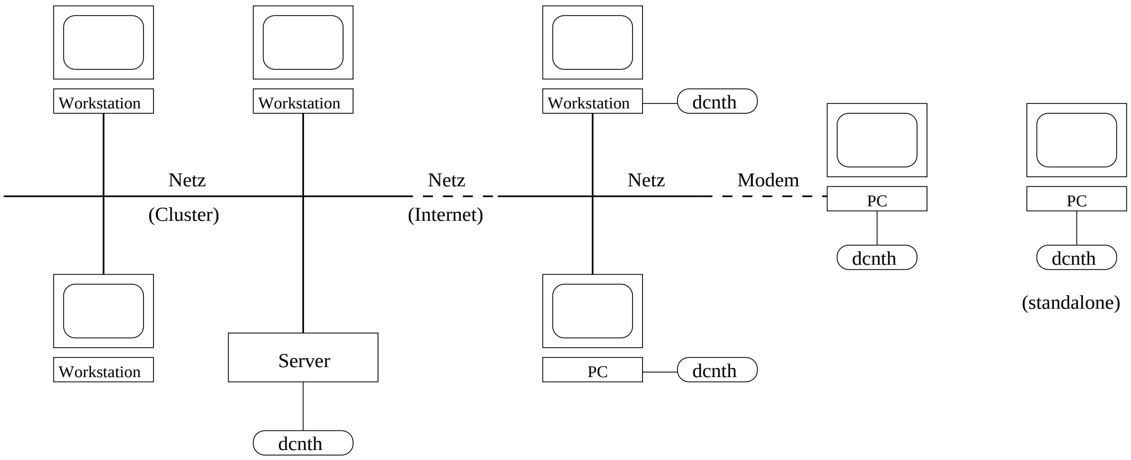


Abbildung 38: Beispielkonfiguration `dcnth`

7.3.1 Grundprinzip

Alle Schritte zur Konfiguration und Steuerung von `dcnth` können über eine einfache Konsole (`dcnthc.ksh`) von jeder Maschine für alle im Netz vorhandenen Rechner vorgenommen werden. Rechner, die sich außerhalb eines Netzes befinden, müssen natürlich separat konfiguriert und gesteuert werden. `dcnthc.ksh` wird in Abschnitt 7.3.9 genauer beschrieben.

Die Verteilung von Teilrechnungen der mit `dcnthd` bezeichneten Rechenprozesse und deren Kontrolle wird in `dcnth` über ein Überprüfungs-Shellskript `dcnthchk.ksh` vorgenommen. Dieses wird vom „Terminplaner" des Betriebssystems (`cron`) zu festzulegenden Zeitpunkten auf jedem Rechner gestartet.

Der Ablauf ist Abbildung 39 zu entnehmen.

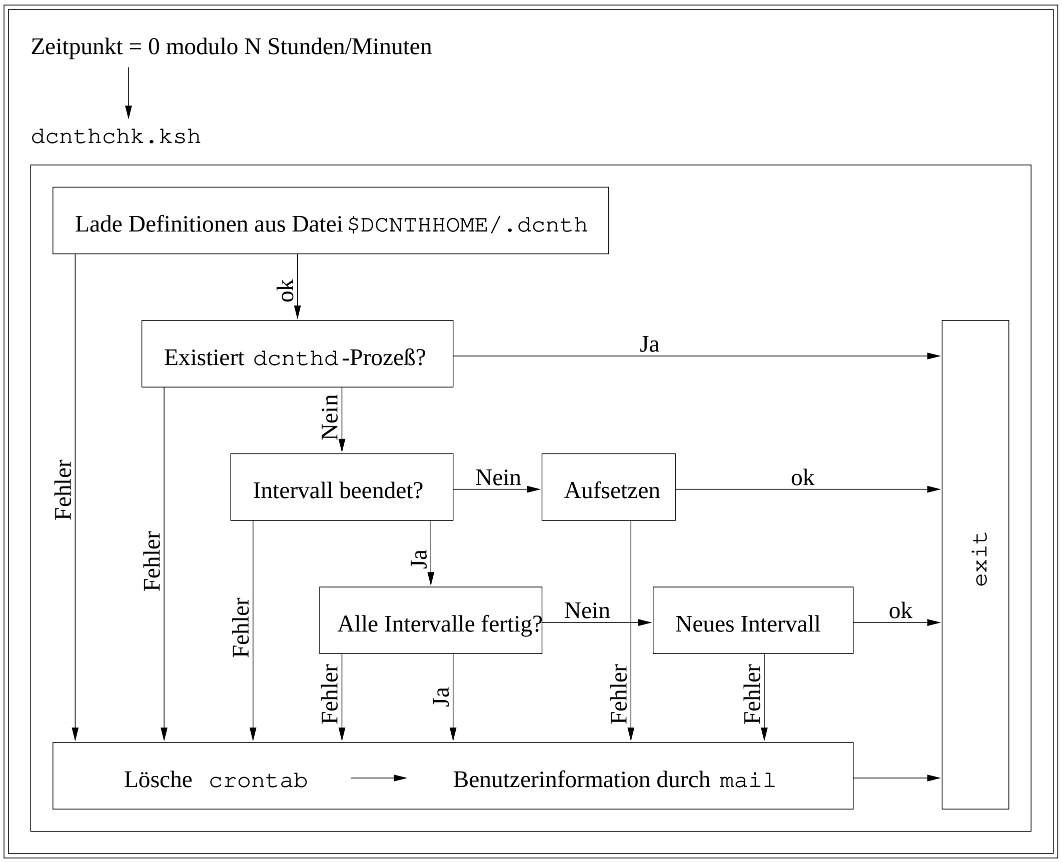


Abbildung 39: Ablauf `dcnth`

7.3.2 Installation

Zur Installationsvorbereitung muß auf jedem Rechner ein Benutzeraccount angelegt sein. Eine Umgebungsvariable `DCNTHHOME` wird gesetzt, die das Installationsverzeichnis referenziert. Die gesamte Software ist als gepackte und komprimierte Datei etwa 20kB groß. Die Installation erfordert ausschließlich das Entpacken und Dekomprimieren dieser Datei. Dies muß jedoch auf jedem zu verwendenden Rechner stattfinden. Eine Ausnahme bilden Cluster mit gemeinsamen Speicherzugriff, wo nur eine einzige Installation erforderlich ist. Der entstehende Verzeichnisbaum ist in Abbildung 40 dargestellt. Dabei sind bereits einige Beispielrechner und -architekturen mit eingetragen.

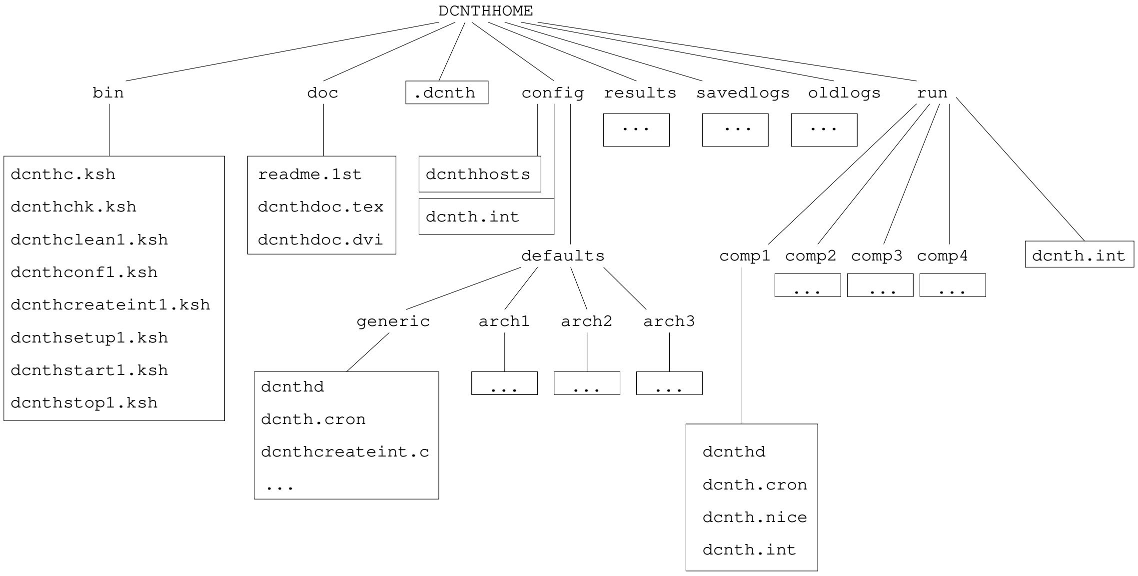


Abbildung 40: Verzeichnisbaum `dcnth`

7.3.3 Vorbereitung einer verteilten Rechnung

Um nach der Installation von `dcnth` eine verteilte Rechnung zu beginnen, müssen die folgenden Schritte vollzogen werden:

1. Festlegung der Rechner-Konfiguration
2. Festlegung der Rechnungs-Parameter
3. Erstellung des eigentlichen Programms für jede Architektur
4. Erstellung der Teilintervalle
5. Aufteilung der Teilintervalle

Zunächst müssen für jeden Cluster sowohl die beteiligten Rechnerarchitekturen als auch die darauf basierenden Rechner eingetragen werden (`dcnthc.ksh`→`addarch/addhost`). Die Konfiguration erfolgt mit `dcnthc.ksh`→`conf`. Die eigentlichen Rechenprogramme müssen erstellt und in die im ersten Schritt entstandenen Verzeichnisse unter dem Namen `dcnthd` kopiert werden. Die Zielverzeichnisse der Logdatei-Sicherungen (Verzeichnis `oldlogs`) und ihrer Zweitkopien (`savedlogs`) sollten möglichst über Links auf unterschiedliche Platten verweisen.

Danach werden die relevanten Dateien aus den für die jeweiligen Rechner zutreffenden Architektur-Verzeichnissen (in `config/defaults`) in das Rechnerverzeichnis im Unterverzeichnis `run` kopiert (`dcnthc.ksh`→`setup`).

Nachdem die Aufspaltung der Teilintervalle vollzogen ist, erfolgt die Erzeugung der entsprechenden Dateien, wobei an dieser Stelle natürlich der Vorteil des gemeinsamen Speicherzugriffs zum Tragen kommt. Insbesondere sind auch Rechner ohne Netzzugriff zu versorgen.

Danach kann die Rechnung gestartet werden.

7.3.4 Start

Der Start der Rechnung kann für alle netzverbundenen Rechner mit den `dcnthc.ksh`-Kommandos `set all` und `start` vollzogen werden. Die Folge ist die Installation der für die jeweiligen Rechner gültigen Terminkalender `crontab`. Zum dort festgelegten Zeitpunkt wird `dcnthchk.ksh` aufgerufen und der eigentliche Rechenprozeß `dcnthd` gestartet (siehe auch 7.3.7).

7.3.5 Zwischenkontrolle

Beispielsweise jede halbe Stunde startet `dcnthchk.ksh` erneut und prüft, ob der Hintergrundprozeß `dcnthd` noch läuft.

7.3.6 Reaktionen bei Fehlern

Nach temporärem Rechnerausfall wird der Prozeß `dcnthd` von `dcnthchk.ksh` automatisch wieder gestartet. Dies erfolgt durch Lesen der Datei *logfile*.`last`, in der sich der letzte `dcnthd`-Aufruf mit Kommandozeilenparametern befindet (siehe auch 7.3.7). Falls dies aus irgendwelchen Gründen nicht möglich war, wird die entsprechende `crontab` gelöscht und der Benutzer durch `mail` vom Störfall benachrichtigt.

7.3.7 Intervallverteilung

Nach normaler Beendigung eines Intervalls werden zunächst die Logdateien doppelt gesichert. Dann wird aus der durch die Variable `$DCNTHINTFILE` festgelegten Datei `dcnth.int` eine Zeile destruktiv eingelesen und die Berechnung mit diesem Intervall gestartet. Dabei wird die zusätzliche Datei *logfile*`.last` angelegt, in die die Kommandozeile des Aufrufs geschrieben wird.

7.3.8 Ende einer verteilten Rechnung

Das Ende einer Rechnung tritt ein, wenn sämtliche Intervalldateien leer sind, also alle Teilintervalle abgearbeitet sind.

7.3.9 Die Steuerkonsole `dcnthc.ksh`

Die Konfiguration und Steuerung von `dcnth` erfolgt mit Hilfe eines kleinen Konsolenprogramms namens `dcnthc.ksh`. Dabei ist es möglich, alle durch ein Netz erreichbare Rechner in einer einzigen `dcnthc.ksh`-Sitzung auf einfache Art zu manipulieren.

Es sind folgende Punkte verfügbar:

Kommando	Bedeutung
`addarch`	Hinzufügen einer Architektur
`addhost`	Hinzufügen eines Rechners
`clean`	Aufräumen nach Tests
`conf`	Konfiguration von Rechnern
`droparch`	Löschen einer Architektur
`drophost`	Löschen eines Rechners
`env`	Anzeige der `dcnth`-Umgebungsvariablen
`exit/quit`	Verlassen von `dcnthc.ksh`
`help`	Anzeige des Menüs
`int`	Erstellen einer Intervalldatei
`log`	Anzeige der globalen Logdatei
`ps`	Anzeige des lokalen `dcnthd`-Prozesses
`restart`	Stoppen und Starten von `dcnthd`
`set`	Einstellung der Rechnerumgebung
`setup`	Initialisierung der Rechner
`start`	Starten der Berechnung
`stop`	Stoppen der Berechnung
`top`	Kontrolle konkurrierender Prozesse

Funktionsweise Nach dem Start ist `dcnthc.ksh` zunächst auf den lokalen Rechner eingestellt, d.h. alle auszuführenden Aktionen sind auf diesen Rechner beschränkt. Durch `set` *Rechnername(n)* kann jede beliebige Teilmenge aller verfügbaren Rechner eingestellt werden. `set all` setzt dabei alle verfügbaren Rechner. Nach Setzen der Rechnerumgebung werden alle Aktionen auf den hinter `set` angegebenen Maschinen durchgeführt. Dabei ist eine Netzwerkerreichbarkeit vorausgesetzt, d.h. auf standalone-Rechnern wird das Kommando `set` nie ausgeführt.

Die einzelnen Kommandos führen zu einer Remote-Ausführung der zugehörigen, auf den Rechnern vorhandenen Shellskripte via `telnet/rlogin` bzw. der sichereren Version `ssh`.

Zu den einzelnen Kommandos von `dcnthc.ksh`:

`addarch` fügt eine Architektur (z.B. sun4) hinzu. Folge: Im Verzeichnis `config/defaults` entsteht ein Unterverzeichnis, in das zunächst die Dateien aus `config/defaults/generic` kopiert werden. `droparch` löscht dieses Verzeichnis wieder.

`addhost` fügt einen Rechner hinzu. Folge: Ein Unterverzeichnis für diesen Rechner wird im Verzeichnis `run` angelegt und alle Dateien aus dem für diesen Rechner gültigen Architekturverzeichnis kopiert.

`clean` ist für Testzwecke vorhanden und löscht sämtliche Dateien, beläßt aber die Grundkonfiguration. `clean` ist während einer laufenden Rechnung gesperrt.

`conf` dient der Konfiguration von Rechnern. Dabei werden Laufzeiten (z.B. nur nachts und am Wochenende) gesetzt, die Priorität des Rechenprozesses definiert und die Häufigkeit des Aufrufs des Kontrollprozesses festgelegt. Eine Rekonfiguration findet ebenso mit `conf` statt.

`int` erzeugt ein Teilintervall, daß je nach Definition in der Datei `dcnthhosts` (siehe unten) entweder im globalen Verzeichnis `run` oder im lokalen `run/`*Rechnername* liegt.

Der Zweck von `setup` ist die Neuverteilung der `dcnthd`-Prozesse aus den Architekturverzeichnissen auf die einzelnen Rechner nach Veränderungen der Quellen. `setup` ist während einer laufenden Rechnung nicht möglich.

`start/stop` startet/stoppt die Berechnung auf allen durch `set` eingestellten Rechnern durch Installation bzw. Deinstallation der `crontab`s.

`top` führt das Unix-Kommando `top` auf allen Rechnern aus, um eine Übersicht über konkurrierende Prozesse auf den Maschinen zu bekommen. Hängengebliebene Prozesse anderer Benutzer können so leicht aufgespürt und beendet werden.

7.3.10 Die Dateien `.dcnth` und `dcnthhosts`

`dcnth` enthält zwei wesentliche Konfigurationsdateien:

`$DCNTHHOME/.dcnth`: Die Datei `.dcnth` wird von jedem einzelnen `dcnth`-Shellskript als erstes ausgeführt. In `.dcnth` werden alle wichtigen Verzeichnis- und Dateinamen ermittelt, einige Shellfunktionen definiert sowie Betriebssystem(versions)spezifische Kommandos beschrieben.

`dcnthhosts`: Die Datei `dcnthhosts` im Verzeichnis `config` enthält sämtliche Rechner, die im entsprechenden Teilnetz an einer Rechnung beteiligt sind. Der Aufbau einer Zeile sieht wie folgt aus:

Rechner(alias-)name:Architektur:Default-Priorität:Rechner-IP-Adresse:Intervalldatei-Flag

Das Intervalldatei-Flag gibt dabei an, ob die entsprechende Maschine eine eigene Intervalldatei besitzt oder eine Datei mit anderen Rechnern teilt.

In den folgenden Abschnitten werden die Vor- und Nachteile der verschiedenen Konfigurationen zusammengefaßt.

7.3.11 `dcnth` im Netz mit NFS

Der Vorteil von Rechnern mit gemeinsamem Plattenzugriff liegt vor allem darin begründet, daß Intervalldateien gemeinsam genutzt werden können. Ein Lock-Mechanismus ermöglicht dabei konkurrenten Zugriff. Darüber hinaus muß `dcnth` nur einmal gespeichert und konfiguriert werden. Ein abschließendes Sammeln entfällt.

Der Nachteil gemeinsamer Daten ist die Abhängigkeit aller Rechner vom (NFS-) Serverrechner. Ein Ausfall bedeutet hier die Blockade aller Rechnungen.

7.3.12 Vernetzte Systeme ohne gemeinsamen Speicher

Rechner, auf die über das Netz zugegriffen werden kann, können entsprechend von einer einzigen Stelle aus konfiguriert und gesteuert werden. Die abschließende Sammlung von Logbuchdateien kann recht einfach über Programme wie `ftp` erfolgen.

Allerdings muß mangels gemeinsamen Speichers jeder Rechner eigene Intervall-Dateien besitzen. Die Erstellung dieser Dateien kann wiederum von jedem Rechner im Netz remote erfolgen.

7.3.13 `dcnth` auf Standalone-Rechnern

Standalone-Rechner sind netzunabhängig und daher völlig autark. Allerdings muß jede beteiligte Standalone-Maschine separat konfiguriert und gesteuert werden. Ein noch größerer Nachteil liegt dabei in der Schwierigkeit der Sammlung von Resultaten, wenn diese aus relativ großen Datenmengen bestehen. Dieses Problem ergab sich z.B. bei der Berechnung der Goldbach-Partitionen, wo pro Intervall jeweils eine Datei der Größe $\approx$ 2MB entstand.

7.4 Ausblick/Probleme

`dcnth` hat sich in der Praxis als sehr bequemes Hilfsmittel zur Steuerung verteilter Rechnungen erwiesen. Allerdings gibt es einige Verbesserungsmöglichkeiten. Dazu zählen vor allem:

- Erstellung einer graphischen Oberfläche
- Vereinfachung/Automatisierung der Initialisierung vor Beginn
- Betriebssystemunabhängige Implementierung
- Erweiterte Kontrollmöglichkeiten
- Automatische Erholung von dauerhaften Rechnerausfällen
- Automatische Neuverteilung von Teilintervallen
- (Teil-)Automatisierung der Sammlung von Daten nach Ende
- Anzeige des Rechnungsstatus

Danksagungen

An dieser Stelle möchte ich mich herzlich bei allen bedanken, die durch ihre Unterstützung, Hilfe und Geduld zur vorliegenden Arbeit beigetragen und sie dadurch erst ermöglicht haben.

Besonderer Dank gilt meinem Betreuer Herrn Prof. Dr. Henner Kröger, der mir nicht nur durch wertvolle Diskussionen, kritische Anmerkungen und Ratschläge half, sondern mir auch den nötigen Freiraum gab, den die Erstellung dieser Arbeit erforderte.

Für die Übernahme des Zweitgutachtens und einige wichtige Hinweise möchte ich Herrn Prof. Dr. Sigbert Jaenisch herzlich danken.

Bei Herrn Dr. Martin Kutrib möchte ich mich für die zahlreichen Diskussionen, seine geduldige Unterstützung in allen Bereichen und nicht zuletzt für seine Mühe, die Arbeit Korrektur zu lesen, sehr bedanken.

Herrn Dipl.-Math. Thomas Buchholz danke ich für seine uneingeschränkte Hilfsbereitschaft bei vielen Problemen, insbesondere im Zusammenhang mit der verwendeten Hard- und Software, ohne deren zuverlässige Wartung keine der hier durchgeführten Rechnungen so reibungslos hätte verlaufen können.

Im selben Zusammenhang möchte ich Herrn Dipl.-Math. Jan-Thomas Löwe danken, der mir durch seine sofortige Unterstützung bei technischen Problemen sehr geholfen hat.

Bei Herrn Dr. Gerrit Eichner möchte ich mich für einige interessante Diskussionen und sein unermüdliches Engagement in der akademischen Selbstverwaltung bedanken, von dem ich nicht nur direkt profitiert habe, sondern durch das ich selbst auch entlastet wurde.

Für ihre Mühe des Korrekturlesens und ihre Geduld und Rücksicht während der Fertigstellung dieser Arbeit danke ich sehr herzlich meiner Freundin Dr. Anne Katrin Hilbert.

Literatur

[Abe28] N. H. Abel. Aufgabe 28. *Journal für die reine und angewandte Mathematik*, 3:212, 1828.

[Adl78] L. Adleman, R.L. Rivest, A. Shamir. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[Adl83] R. Rumely, L. M. Adleman, C. Pomerance. On distinguishing prime numbers from composite numbers. *Annals of Mathematics*, 117:173–206, 1983.

[Aho74] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[Aho86] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, Tools.* Addison-Wesley, 1986.

[App77] W. Haken, K. Appel. Every planar map is four colorable. *Illinois Journal of Mathematics*, 21:429–490, 1977.

[Bac96] J. Shallit, E. Bach. *Algorithmic Number Theory.* MIT Press, 1996.

[Bay77] R. Hudson, C. Bays. The segmented sieve of eratosthenes and primes in arithmetic progressions. *BIT*, 17:121–127, 1977.

[Bee22] N. G. W. H. Beeger. On a new case of the congruence $2^{p-1} \equiv 1(mod\ p^2)$. *Messenger of Mathematics*, 51:149–150, 1922.

[Bor56] K. G. Borozkin. On the problem of I.M. Vinogradov's constant. *Proceedings Third Mathematical Congress*, 1, 1956.

[Bre73] R. P. Brent. The first occurrence of large gaps between successive primes. *Mathematics of Computation*, 27:959–963, 1973.

[Bri71] P. Weinberger, J. Brillhart, J. Tonascia. On the Fermat Quotient. In B. J. Birch A. O. L. Atkin, editor, *Computers in Number Theory*, pages 213–222. Academic Press, 1971.

[Bru15] V. Brun. Über das Goldbachsche Gesetz und die Anzahl der Primzahlpaare. *Archiv for Mathematik og Naturvidenskab*, 34:8–19, 1915.

[Cay78] A. Cayley. The theory of groups: Graphical representations. *American Journal of Mathematics*, 11:139–157, 1878.

[Che89] Y. Wang, J.-R. Chen. On the odd Goldbach problem. *Sci. Sinica*, 32:702–718, 1989.

[Coo73] C. Reckhow, S. Cook. Time bounded random access machines. *Journal of Computer and System Science*, 7:354–375, 1973.

[Cra97] C. Pomerance, R. Crandall, K. Dilcher. A search for Wieferich and Wilson primes. *Mathematics of Computation*, 66:433–449, 1997.

[Des93] W. Narkiewicz, C. Pomerance, J.-M. Deshoulliers, A. Granville. An upper bound in Goldbach's problem. *Mathematics of Computation*, 61:209–213, 1993.

[Des97] H.J.J te Riele, D. Zinoviev, J.-M. Deshoulliers, G. Effinger. A complete Vinogradov 3-primes theorem under the Riemann hypothesis. *Electronic Research Announcements of the AMS*, 3:99–104, 1997.

[Des98] Y. Saouter J.-M. Deshoulliers, H. J. J. te Riele. New experimental results concerning the Goldbach conjecture. *Proceedings Third International Symposium on Algorithmic Number Theory*, 61:204–215, 1998.

[Des99] H. J. J. te Riele, J.-M. Deshoulliers. On the probabilistic complexity of numerically checking the binary goldbach conjecture in certain intervals. Technical report, Centrum voor Wiskunde en Informatica, 1999.

[Doo26] L. C. Karpinski, M. L. D'ooge, F. E. Robbins. *Nicomachus of Gerasa*. Macmillan, 1926.

[Dun96] J. P. Sorenson, B. Dunten, J. Jones. A space-efficient fast prime number sieve. *Information Processing Letters*, 59:79–84, 1996.

[Elg64] A. Robinson, C. Elgot. Random access stored program machines. *Journal of the ACM*, 11:365–399, 1964.

[Ern97] T. Metsänkylä, R. Ernvall. On the p-divisibility of Fermat-quotients. *Manuskript, eingereicht zur Publikation*, 1997.

[Gla78] J. W. L. Glaisher. An enumeration of prime-pairs. *Messenger of Mathematics*, 8:28–33, 1878.

[Gra89] H. J. J. te Riele, A. Granville, J. van de Lune. Checking the Goldbach conjecture on a vector computer. In R. A. Mollin, editor, *Number Theory and Applications*, pages 423–433. Kluwer, 1989.

[Gra95] T. Granlund. Gnu multiple precision 2. Technical report, TMG Datakonsult, Göteborg, 1995.

[Gri78] J. Misra, D. Gries. A linear sieve algorithm for finding prime numbers. *Communications of the ACM*, 21:999–1003, 1978.

[Hal74] H.-E. Richert, H. Halberstam, editor. *Sieve Methods*. Academic Press, 1974.

[Har22] J. E. Littlewood, G. H. Hardy. Some problems of 'partitio numerorum'; III: On the expression of a number as a sum of primes. *Acta Math*, 44:1–70, 1922.

[Har79] E. M. Wright, G. H. Hardy. *Introduction to the Theory of Numbers, 5th Ed.* Oxford Science Publications, 1979.

[Hee69] H. Heesch. Untersuchungen zum Vierfarbenproblem. *Hochschulskriptum, Bibliographisches Institut, Mannheim*, 810/a/b, 1969.

[Jac28] C. G. J. Jacobi. Beantwortung der Aufgabe S. 212 dieses Bandes. *Journal für die reine und angewandte Mathematik*, 3:301–302, 1828.

[Jae93] G. Jaeschke. On strong pseudoprimes to several bases. *Mathematics of Computation*, 61:915–926, 1993.

[Kel97] W. Keller. Solutions of the Congruence $a^{p-1} \equiv 1(mod\ p^r)$. Private Kommunikation, 1997.

[Knu81a] D. E. Knuth. *The Art of Computer Programming 1: Fundamental Algorithms.* Addison-Wesley, 1981.

[Knu81b] D. E. Knuth. *The Art of Computer Programming 2: Seminumerical Algorithms.* Addison-Wesley, 1981.

[Kuc96] H. Kuchen. Skriptum zur Vorlesung Informatik III, Universität Gießen, 1996.

[Lan00] E. Landau. Über die zahlentheoretische Funktion $\varphi(n)$ und ihre Beziehung zum Goldbachschen Satz. *Göttinger Nachrichten*, pages 177–186, 1900.

[Lav98] Y. Saouter, D. Lavenier. The Goldbach Conjecture: Checking it and counting the Partitions. Eingereicht zur Publikation, 1998.

[Mai77] H. G. Mairson. Some new upper bounds on the generation of prime numbers. *Communications of the ACM*, 20:664–669, 1977.

[Mei13] W. Meissner. Über die Teilbarkeit von $2^p - 2$ durch das Quadrat der Primzahl 1093. *Sitzungsberichte der königlich preussischen Akademie der Wissenschaften*, 2:663–667, 1913.

[Mer97] F. Mertens. Über eine zahlentheoretische Funktion. *Sitzungsberichte der Akademie Wissenschaften Wien*, 106 IIa:761–830, 1897.

[Mon75] R. C. Vaughan, H. L. Montgomery. On the exceptional set in Goldbach's problem. *Acta Arith.*, 27:353–370, 1975.

[Odl85] H. J. J. te Riele, A. Odlyzko. Disproof of the Mertens conjecture. *Journal für die reine und angewandte Mathematik*, 357:138–160, 1985.

[Pau78] W. J. Paul. *Komplexitätstheorie.* Teubner, 1978.

[Pri81] P. Pritchard. A sublinear additive sieve for finding prime numbers. *Communications of the ACM*, 24:18–23, 1981.

[Pri83] P. Pritchard. Fast compact prime number sieves. *Journal of Algorithms*, 4:332–344, 1983.

[Rei90] K. R. Reischuk. *Einführung in die Komplexitätstheorie.* Teubner, 1990.

[Rib83] P. Ribenboim. 1093. *The Mathematical Intelligencer*, 5:28–33, 1983.

[Rib91] P. Ribenboim. *The Little Book of Big Primes.* Springer, 1991.

[Rib96] P. Ribenboim. *The New Book of Prime Number Records.* Springer, 1996.

[Rie85] H. J. J. te Riele. Some historical and other notes about Mertens conjecture and its recent disproof. *Nieuw Archiv voor Wiskunde*, 3:237–243, 1985.

[Rie94] H. Riesel. *Prime Numbers and Computer Methods for Factorization, 2nd Edition.* Birkhuser, 1994.

[Sao98] Y. Saouter. Checking the odd Goldbach conjecture up to 10^{20}. *Mathematics of Computation*, 67:863–866, 1998.

[Sao99] Y. Saouter. Computations of Goldbach partitions up to 128×10^6 with fft. *Zur Publikation eingereicht*, 1999.

[Sel42] E. S. Selmer. Eine neue hypothetische Formel für die Anzahl der Goldbachschen Spaltungen einer geraden Zahl, und eine numerische Kontrolle. *Archiv for Mathematik og Naturvidenskab*, 46, 1942.

[She64] M.-K. Shen. On checking the Goldbach conjecture. *BIT*, 4:243–245, 1964.

[Sin93] M. K. Sinisalo. Checking the Goldbach conjecture up to $4 \cdot 10^{11}$. *Mathematics of Computation*, 61:931–934, 1993.

[Sor98] J. P. Sorenson. Trading time for space in prime number sieves. In J. Buhler, editor, *Proceedings of the Third International Algorithmic Number Theory Symposium*, pages 179–195, Portland, Oregon, 1998. LNCS 1423.

[Ste65] P. R. Stein, M. L. Stein. New experimental results on the Goldbach conjecture. *Math Mag*, 38:72–80, 1965.

[Stä96] P. Stäckel. Über Goldbach's empirisches Theorem: Jede grade Zahl kann als Summe von zwei Primzahlen dargestellt werden. *Göttinger Nachrichten*, pages 292–299, 1896.

[Syl71] J. J. Sylvester. On the partition on an even number into two primes. *Proceedings of the London Mathematical Society*, 4:4–6, 1871.

[Vin37] I. M. Vinogradov. Representation on an odd number as a sum of three primes. *Doklady Akad. Nauk SSSR*, 16:169–172, 1937.

[Wag86] G. Wechsung, K. Wagner. *Computational Complexity*. VEB Deutscher Verlag der Wissenschaften, 1986.

[Wie09] A. Wieferich. Zum letzten Fermat'schen Theorem. *Journal für die reine und angewandte Mathematik*, 136:293–302, 1909.

[Wil95] A. Wiles. Modular elliptic curves and Fermat's Last Theorem. *Annals of Mathematics*, 141:443–551, 1995.

[Wol99] G. Woltham. Great internet mersenne prime search. http://www.mersenne.org, 1999.

[Woo61] T. C. Wood. Algorithm 35: sieve. *Communications of the ACM*, 4:151, 1961.

[Yua84] W. Yuan, editor. *Goldbach Conjecture*. World Scientific, 1984.

[Zim99a] P. Zimmermann. ECMNET. http://www.loria.fr/~zimmerma/records/ecmnet.html, 1999.

[Zim99b] P. Zimmermann. Wieferich status. http://www.loria.fr/~zimmerma/records/Wieferich.status, 1999.

[Zin97] D. Zinoviev. On Vinogradov's constant in Goldbach's ternary problem. *Journal of Number Theory*, 65:334–358, 1997.

A Einige zahlentheoretische Hilfsmittel

Die hier – ohne Beweis – angegebenen Sätze werden für die Analyse der vorgestellten Algorithmen benötigt. Beweise finden sich z.B. in [Har79], [Rib96], [Bac96]

Satz A.1

$$\sum_{p \leq x} \frac{1}{p} = \log\log x + C + O\left(\frac{1}{\log x}\right),$$

wobei $C \approx 0,2615\ldots$.

Satz A.2

$$\sum_{p \leq x} \log p \sim x$$

Satz A.3

$$\pi(x) > \frac{x}{\log x}\left(1 + \frac{1}{2\log x}\right)$$

Satz A.4 Für $x > 58$:

$$\pi(x) < \frac{x}{\log x}\left(1 + \frac{3}{2\log x}\right)$$

Satz A.5 Für $n \geq 2$:

$$p_n \geq n\log n + n\log\log n - 1,0072629n$$

Satz A.6

$$\sum_{p \leq x} p \sim \frac{x^2}{2\log x}.$$

Satz A.7

$$\pi(x) \sim \frac{x}{\log x}$$

Satz A.8

$$\prod_{p \leq x} \frac{p-1}{p} = O\left(\frac{1}{\log x}\right)$$