Multiple precision integer arithmetic and public key encryption

Multiple precision integer arithmetic and public key encryption

Mathias Engan

2009-10-13

© 2009, M. Engan. All rights reserved.

First edition 2009-10-13

ISBN 978-1-4452-1122-0

To Axel, our son. And to Katarina. Thank you for your patience.

Contents

Contents vii

List of Figures ix

List of Tables xi

1 Preface 1

2 Mathematical background 3
- 2.1 Prerequisites . . . 3
- 2.2 Integers . . . 5
- 2.3 Euclid's theorems . . . 9
- 2.4 The fundamental theorem of arithmetic . . . 11
- 2.5 Modular arithmetic . . . 12
- 2.6 Fermat's little theorem . . . 15
- 2.7 Euler's theorem . . . 18

3 Computational Complexity 23
- 3.1 Prerequisites . . . 23
- 3.2 Turing machine . . . 24
- 3.3 Complexity . . . 28
- 3.4 Classes . . . 29

4 Elementary arithmetic 33
- 4.1 Representation . . . 33
- 4.2 Implementation . . . 37
- 4.3 Multiple precision arithmetic . . . 39

4.4 Addition 40
4.5 Subtraction 47
4.6 Multiplication 53
4.7 Division 63

5 Other operations **89**
5.1 Comparison 89
5.2 Even and odd 92
5.3 Length 92
5.4 Shift 93
5.5 Bit operations 97
5.6 Signed addition and subtraction 101
5.7 Greatest common divisor 105
5.8 Modular exponentiation 117
5.9 Scratch space 120
5.10 Error handling 123
5.11 Initialization 125

6 Random numbers **127**
6.1 Randomness 127
6.2 Entropy 133
6.3 Harvesting entropy 136
6.4 Random number generator 140
6.5 The Blum-Blum-Shub generator 143

7 Finding prime numbers **149**
7.1 Introduction 149
7.2 Prime number theorem 155
7.3 Probabalistic primality tests 157
7.4 Generating large prime numbers 163

8 RSA **167**
8.1 RSA 167
8.2 RSA Implementation 172

Bibliography **181**

Index **185**

List of Figures

4.1 Representation 39
4.2 Representation, size and length 39
4.3 Result z_i of primitive addition $(x_i)_b + (y_i)_b = (cz_i)_b$ in base $b = 10$ 40
4.4 Long addition $(Z = X + Y)$. 41
4.5 Addition registers. 43
4.6 Sum (S) and terms $(X$ and $Y)$. 44
4.7 Result z_i of primitive subtraction $(x_i)_b - (y_i)_b = (cz_i)_b$ in base $b = 10$ 47
4.8 Long subtraction $(Z = X - Y)$. 48
4.9 Subtraction registers. 49
4.10 Result z_i of primitive multiplication $x_i \cdot y_i = (cz_i)_b$ in base $b = 10$ 54
4.11 Long multiplication $(Z = X \cdot y_0)$. 54
4.12 Long multiplication $(Z = X \cdot Y)$. 55
4.13 Primitive multiplication base $b = 2^{\frac{w}{2}}$. 58
4.14 Multiplication registers. m_i and m'_j represent w bit memory cells and $\frac{w}{2}$ bit cells respectively. 59
4.15 Quotient q_i of primitive division $\lfloor x_i/y_i \rfloor = q_i$ in base $b = 10$. 64
4.16 Remainder r_i of primitive division $x_i - \lfloor x_i/y_i \rfloor = r_i$ in base $b = 10$ 64
4.17 Long division $(X = QY + R)$. 65
4.18 Long division example. 70
4.19 Division registers. 71
4.20 Addn. 76
4.21 Mulsubn. 79

4.22 Evaluating estimated quotient. 86
4.23 First comparison. 87

List of Tables

3.1 Compute [0 1 0 1 0 1 ...] 26

5.1 Multiple precision comparison. 89
5.2 Comparison and equality operators. 91
5.3 Signed addition 101
5.4 Signed subtraction 103
5.5 Extended gcd algorithm. 111

6.1 Exclusive-or. 139

Chapter 1

Preface

This started because I wanted to know how public key encryption works. A safe way to learn and to understand is to implement. I immediately realized the importance of multiple precision integer arithmetic. Fortunately, I did not realize the complexity of division.

I started reading about public key encryption, and more specifically, RSA, in early december 2008. Early in january 2009 I happily wrote the first lines of code, oblivious to the size of my pet project. When the code was basically done, I started documenting it. It grew into a book. This book.

I have spent hours of my precious spare time. During my parental leave I used the time when our son was napping to work on this project. And the late nights when the rest of the family slept. Writing code and seeing it run gives me satisfaction. That positive feedback kept me going. I hope more good comes out of this work.

The main part of this book is chapter 4, about the elementary arithmetic operations addition, subtraction, multiplication and divison. I present a clean and portable implementation of multiple precision arithmetic. In chapter 5 I discuss other multiple precision operations, such as comparison, bit operations, greatest common divisor, signed addition and subtraction and exponentiation. All necessary for implementation of RSA.

In chapters 6 and 7 I discuss random number generation and prime number generation. We need them both. Random numbers are a foundation of modern cryptography and large prime numbers are a cornerstone

in many modern cryptosystems. The RSA cryptosystem is discussed in chapter 8 and constitutes the final chapter of the book.

The two first chapters, chapters 2 and 3 is an attempt to make this book completley self contained. My goal has been to include all that is necessary, but not much else, to understand and implement public key cryptography.

A goal of mine has been to make the implementation self contained, with no, or at least very few, dependencies on external libraries. The routines presented in this text are sufficient for a working implementation of RSA. With one exception. Entropy. A real world implementation must provide, the inherently non-portable, entropy harvesting.

Chapter 2

Mathematical background

This is not a mathematics textbook. It is a book about multiple precision arithmetic and public key encryption. However, the goal is to make this text self contained. Thus we present the definitions and theorems which we will make use of in the following chapters.

2.1 Prerequisites

We begin with one of the most basic concepts. The set. Using the set as a foundation we can derive nearly all of mathematics. A set is a collection of distinct objects. Georg Cantor defined a set as

> By a *set* we mean any collection M into a whole of definite, distinct objects m (which are called the *elements* of M) of our perception or of our thought.

The elements of a set can be anything, however, every element of a set must be unique, i.e. no two members can be the same. Here we provide no formal definition of the set, but list some of the properties of a set.

Definition 1. Set.

These are some important characteristics of a set:

1. A set S is made up of *elements*. If a is an element of S we write $a \in S$.

2. There is exactly one set with no elements, the *empty set* which we denote by $\emptyset$.
3. We can describe a set either by giving a characterizing property of the elements or by listing the elements. We usually describe a set either by listing the elements within curly braces, such as $\{a, b, c\}$ or by providing a property, such as $\{x|P(x)\}$ which is read "the set of all x such that $P(x)$ is true.
4. A set is well defined, so that if S is a set and a is some object then a is either in S, $a \in S$ or a is not in S, $a \notin S$.

We continue with the definition of a *subset* and a *partition*. A subset is a set where all members are also members of a (possibly) larger set. A partitioning relation forms distinct subsets, *cells*, of a set, such that an element is a member of one cell.

Definition 2. Subset.

> A set B is a *subset* of A if every element of B is in A which we write $B \subseteq A$. We write $B \subset A$ if $B \subseteq A$ but $B \neq A$.

Definition 3. Partition of a set.

> A *partition* of a set is a decomposition of the set into subsets such that every element of the set is in one and only one of the subsets which we call cells.

Theorem 1. Partitioning relation.

> S is a nonempty set and $\sim$ is a relation between elements of S satisfying:
>
> 1. $a \sim a$ (reflexive).
> 2. if $a \sim b$ then $b \sim a$ (symmetric).
> 3. if $a \sim b$ and $b \sim c$ then $a \sim c$ (transitive).
>
> The relation $\sim$ partitions the set S such that $\bar{a} = \{x \in S | x \sim a\}$ is the cell containing a, $\forall a \in S$.

Proof. $a \in S$ and then $a \in \bar{a}$ (reflexive). So a is in at least one cell. Now, assume a is also in $\bar{b}$ and let $x \in \bar{a}$, then $x \sim a$, but $a \in \bar{b}$ so $a \sim b$, then by transitivity $x \sim b$ so $x \in \bar{b}$ thus $\bar{a} \subseteq \bar{b}$. Let $y \in \bar{b}$ then $y \sim b$, but $a \in \bar{b}$ so $a \sim b$ and by symmetry $b \sim a$ and transitivity $y \sim a$ so $y \in \bar{a}$ and thus $\bar{b} \subseteq \bar{a}$. We now have $\bar{a} \subseteq \bar{b}$ and $\bar{b} \subseteq \bar{a}$ so $\bar{a} = \bar{b}$.

Definition 4. Equivalence relation.

> A relation $\sim$ on a set S satisfying the reflexive, symmetric and transitive properties is an *equivalence relation* on S. Each cell $\bar{a}$ in the partition given by an equivalence relation is an *equivalence class*.

We will see later that the modulo operation partitions the integers into cells and is an equivalence relation.

2.2 Integers

In this text we will almost exclusively be concerned with integers. In fact, most of our work will be focused on positive integers. We assume the reader is familiar with the basic arithmetic operations; addition, subtraction, multiplication and division. In any case we present some important definitions and theorems regarding division and divisibility that will be refered to later.

Definition 5. The set of integers.

> The set $\mathbb{Z}$ is the set of integers:
>
> $$\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$$

Definition 6. Well ordering principle

> In every nonempty set of positive integers there is a smallest element.

Definition 7. Divisor.

> For $a, b \in \mathbb{Z}$, a *divides* b, a is a *divisor* of b or a is a *factor* of b if $\exists c \in \mathbb{Z}$ such that $b = ac$. If a divides b we write $a|b$.

Theorem 2. Properties of divisibility.

For $\forall a, b, c \in \mathbb{Z}$ we have:

1. $a|a$.
2. If $a|b$ and $b|a$ then $a = \pm b$.
3. If $a|b$ and $b|c$ then $a|c$.
4. If $a|b$ and $b|c$ then $\forall x, y \in \mathbb{Z}$, $a|(bx + cy)$.
5. If $ab|c$ then $a|c$ and $b|c$.

Proof. We show each property:

1. If $a|a$ then $\exists i \in \mathbb{Z}$ such that $ia = a$. The only possibility for i is 1 so a divides a.
2. Since $a|b$ $\exists i \in \mathbb{Z}$ such that $ia = b$ and since $b|a$ $\exists j \in \mathbb{Z}$ such that $jb = a$. Substituting gives $ijb = b$ and ij must be 1. Substituting for b gives $jia = a$ and ji must be 1. The only possibilities for i and j are 1 or -1, thus $a = \pm b$.
3. If $a|b$ then $\exists i \in \mathbb{Z}$ such that $ia = b$ and if $b|c$ then $\exists j \in \mathbb{Z}$ such that $jb = c$, since $ia = b$ we have $j(ia) = c$ or $(ji)a = c$ which implies $a|c$.
4. From 3 we know that if $a|b$ and $b|c$ then $a|c$. Since $a|b$ $\exists i \in \mathbb{Z}$ such that $ia = b$ and since $b|c$ $\exists j \in \mathbb{Z}$ such that $jb = c$. Now we can write $bx + cy = iax + jby = iax + jiaxy = a(ix + jixy)$ which implies $a|(bx + cy)$.
5. If $ab|c$ the $\exists k \in \mathbb{Z}$ such that $iab = c$, then $(ib)a = c$ and $a|c$, also, $(ia)b = c$ and $b|c$.

Definition 8. Greatest common divisor.

$d \in \mathbb{Z}$, $d > 0$ is the *greatest common divisor* of $a, b \in \mathbb{Z}$ if:

1. d is a common divisor of a and b such that $d|a$ and $d|b$.
2. whenever $c|a$ and $c|b$ then $c|d$.

We write $d = \gcd(a, b)$ and $\gcd(0, 0) = 0$.

Theorem 3. Division algorithm for integers.

> For $a, b, d \in \mathbb{Z}$ where $d \neq 0$ $\exists q, r \in \mathbb{Z}$ where q and r are unique such that $a = qd + r$ and $0 \leq r \leq |d|$.

Proof. We show existence and uniqueness.

1. To show existence we consider the set $R = \{a - nd | n \in \mathbb{Z}\}$. R has at least one nonnegative member.

    a) If $d > 0$ then $\exists n \in \mathbb{Z}$, $n \geq 0$ such that $dn \geq -a$, that is $a - d(-n) = a + dn \geq 0$.

    b) If $d < 0$ then $-d > 0$ and $\exists n \in \mathbb{Z}$, $n \geq 0$ such that $(-d)n \geq -a$, that is $a - dn \geq 0$.

    So R contains a nonnegative integer. By the well ordering principle R contains a least nonnegative integer r, let $q = (a - r)/d$ then $q, r \in \mathbb{Z}$ and $a = qd + r$.

    $r > 0$ which we know from the definition of R, suppose that $r \geq |d|$, then since $d \neq 0$, $r > 0$ and $d > 0$ or $d < 0$ we have:

    a) If $d > 0$ then $r \geq d$ implies $a - qd \geq -d$ thus $a - qd - d \geq 0$ and $a - (q + 1)d \geq 0$. Now $a - (q + 1)d$ is in R and since $a - (q+1)d = r - d$ and $d > 0$ we must have $(q+1)d < r$ which contradicts the assumption that r is the least nonnegative number of R.

    b) If $d < 0$ then $r \geq -d$ implies $a - qd \geq -d$ thus $a - qd + d \geq 0$ and $a - (q-1)d \geq 0$. Now $a - (q-1)d$ is in R and since $a - (q-1)d = r(-d)$ and $d < 0$ we must have $a - (q - 1)d < r$ which contradicts the assumption that r is the least nonnegative number of R.

    We have that $r > 0$ was not the least nonnegative number of R which is a contradiction, so we must have $r < |d|$. Now we know that q and r exists and $0 \geq r < |d|$.

2. To show uniqueness suppose $\exists q, q', r, r' \in \mathbb{Z}$ with $0 \leq r$, $r' < |d|$ such that $a = dq + r$ and $a = dq' + r'$. Assume $q < q'$. Subtracting gives $0 = dq - dq' + r - r'$ or $d(q' - q) = r - r'$.

If $d > 0$ then $r' \leq r$ and $r < d < d + r'$ so $r - r' < d$. Similarly, if $d < 0$ then $r \leq r'$ and $r' < -d \leq -d + r$ so $-(r - r') < -d$. Combining gives $|r - r'| < |d|$.

Since $d(q' - q) = r - r'$ we must have that $|d|$ divides $|r - r'|$ and we have either $|d| \leq |r - r'|$ or $|r - r'| = 0$. And we know that $|r - r'| < |d|$ so we conclude that $|d|$ can not be less than or equal to $|r - r'|$ and we have $r = r'$. Substituting gives $dq = dq'$. Since $d \neq 0$ we also have $q = q'$.

Theorem 4. Larger divisor yields smaller quotient.

If $y' > y$ then $q' \leq q$ for q in $x = qy + r$ and q' in $x = q'y' + r'$ for $y > 0$ and $y, y', q, q', r, r', x \in \mathbb{Z}$.

If $y' > y$ then $\exists n \in \mathbb{Z}$ such that $y' = y + n$. Consider q' in $x = q'y' + r'$ and q in $q = xy + r$. The we have

$$q' = \frac{x - r'}{y'} = \frac{x - r'}{y + n} \leq \frac{x}{y + n}$$

and

$$q = \frac{x - r}{y} \leq \frac{x}{y}$$

Assume $q' \leq q$ then we have $\frac{x}{y+n} \leq \frac{x}{y}$ which gives $xy \leq xy + xn$ and $0 \leq xn$ which is true, so $q' \leq q$.

Étienne Bézout was a French matematician. He was born in Nemours, Seine-et-Marne, France in 1730, and died 1783 in Basses-Loges, France. He was elected adjoint in mechanics at the French Academy of Sciences in 1758. He published numerous minor works and wrote the Théorie générale des équation algébriques, published 1779. Étienne Bézout proved his identity for polynomials, the same statement for integers can be found in the work of the French mathematician Claude Gaspard Bachet de Méziriac.

Bézouts identity (or Bézouts lemma) states that if a and b are nonzero integers with a greatest common divisor d then there exists numbers x and y, called Bézout numbers or Bézout coefficients, such that $ax + by = d$.

Theorem 5. Bezout's identity.

> If $a, b, d \in \mathbb{Z}$, $a, b \neq 0$ and d is the greatest common divisor of a and b then $\exists x, y \in \mathbb{Z}$ such that $ax + by = d$

Proof. Assume that $a, b > 0$, consider the set $S = \{\forall k | k = am + bn\}$ where $m, n \in \mathbb{Z}$. S is nonempty so by the well ordering principle there is at least one member $d = ax + by$. By the division algorithm $\exists q, r \in \mathbb{Z}$ such that $a = qd + r$, $0 \geq r < d$. Since $r = a - qd = a - q(ax + by) = a(1 - qx) + b(-qy)$ it follows that r must be 0 otherwise $r \in S$ thus contradicting that d is the least element of S. So $a = qd$, i.e. $d|a$. Similarly $d|b$ and d is a common divisor of a and b. By definition d is the greatest common divisor of a and b.

Two integers are *relatively prime* if they have no common factors. One integer is *prime* if it divisible only by itself and one, i.e. it has only one factor, itself. The opposite of a prime is a *composite*, an integer that is the product of two or more prime factors.

Definition 9. Relatively prime (coprime).

> $a, b \in \mathbb{Z}$ are *relatively prime* if $\gcd(a, b) = 1$.

Definition 10. Prime.

> $p \in \mathbb{Z}$, $p \geq 2$ is *prime* if its only positive divisors are 1 and p. If p is not prime p is said to be *composite*.

2.3 Euclid's theorems

> "Give him a coin, since he must profit by what he learns."
>
> – Euclid of Alexandria.

Euclid was a Greek mathematician (300 BC) often referred to as the father of geometry. Euclid was active in Alexandria under Ptolemy I's rule (323 BC – 283 BC). Euclid is the author of Elements, the most successful textbook in the history of mathemathics. In Elements Euclid deduced the principles of what is now known as Euclidean geometry from a small set of axioms. Euclid's works include topics such as perspective, conic sections, spherical geometry and number theory.

Little is know about Euclid except his writings. He was active at the Library of Alexandria and may have studied at Plato's academy in Greece. His date of birth and date of death is unknown.

Many of the results in Euclid's Elements are from earlier mathematicians, Euclid's presented them in a logic and coherent way. He provided a system of writing rigorous mathematical proofs that still is the basis of mathematics.

Elements is best known for it's geometry but also includes number theory. It discusses perfect numbers and what we now know as Mersenne numbers, the infinitude of primes (Euclid's second theorem) and Euclid's lemma (Euclid's first theorem) on factorization and also the Euclidean algorithm for finding the greatest common divisor.

Theorem 6. Euclid's lemma (Euclid's first theorem).

If $p|ab$ then $p|a$ or $p|b$.

Proof. Assume $p|ab$ and that p is relatively prime to a. Then $gcd(a, p) = 1$ and $\exists x, y \in \mathbb{Z}$ such that $xp + ya = 1$. Multiply by b to get $xpb + yab = b$. Now $p|xpb$ and since $p|ab$ we also have $p|yab$, thus b is a multiple of p and we have $p|b$. So p either divides a or divides b.

Theorem 7. The infinitude of primes (Euclid's second theorem).

There is an infinite number of prime numbers.

Proof. Assume the prime numbers are finite. We can then create a finite set $P = \{p_1, p_2, \ldots, p_{n-1}, p_n\}$ of all the prime numbers where p_n is the largest prime number and $p_{n-1} < p_n$.

Now, consider the number $M = 1 + p_1 p_2 \ldots p_{n-1} p_n$. We have:

1. M is not a prime number because p_n is the largest prime number and $M > p_n$. Thus there must be some prime that divides M.

2. M is not divisible by any of the p_i, $1 \leq i \leq n$ and since M is not prime the prime that divides M must be larger than any p_i, $1 \leq i \leq n$.

Thus, for a finite set of prime numbers we can find a number that implies the existence of a larger prime number than already in the set. We conclude that there are an infinite number of prime numbers.

2.4 The fundamental theorem of arithmetic

The fundamental theorem of arithmetic was almost proven by Euclid in his Elements. The first full and correct proof is in Disquisitiones Arithmeticae by Carl Friedrich Gauss. The fundamental theorem of arithmetic is also called the unique factorization theorem and is a corollary of Euclid's lemma.

Theorem 8. The fundamental theorem of arithmetic.

$n \in \mathbb{Z}$, $n \geq 2$ can be written as a product of prime powers:

$$n = p_1^{e_1} p_2^{e_2} \dots p_{k-1}^{e_{k-1}} p_k^{e_k}$$

The factorization is unique up to the rearrangement of factors.

Proof. We show existence and uniqueness:

1. To show existence suppose there is a positive integer that can not be written as a product of prime numbers, then, by the well ordering principle, there must be a smallest such number, n. n can not be prime, by our initial assumption, so n must be a composite number and we have $n = ab$, $a, b \in \mathbb{Z}$, $a, b < n$. Now, n is the smallest number that has no prime factorization so a and b can be written as a product of prime numbers. Since $n = ab$, n must also be a product of prime numbers, contradicting our initial assumption. Thus all $x \in \mathbb{Z}$ can be written as a product of prime numbers.

2. To show uniqueness suppose s is the smallest positive integer that can be written as at least two different products of prime numbers. Thus $s = p_1 p_2 \cdots p_{m-1} p_m$ and also $s = q_1 q_2 \cdots q_{n-1} q_n$. By Euclid's lemma (theorem 6) $p_1 | q_1$ or $p_1 | q_2 q_3 \cdots q_{n-1} q_n$. Since q_1 and $q_2 q_3 \cdots q_{n-1} q_n$ are smaller than s both must have unique prime factorizations which implies $p_1 = q_i$, $1 \leq i \leq n$. We remove p_1 and q_i from $p_1 p_2 \cdots p_{m-1} p_m$ and $q_1 q_2 \cdots q_{n-1} q_n$ respectively. Now we have an integer s' where $s' < s$ which contradicts our initial assumption. Thus there can be no such s and all natural numbers have an unique prime factorization.

So, by 1 and 2 there is a unique prime factorization $\forall n \in \mathbb{Z}$.

2.5 Modular arithmetic

> "If others would but reflect on mathematical truths as deeply and as continuously as I have, they would make my discoveries."
>
> – Carl Friedrich Gauss.

Modular arithmetic was introduced by Carl Friedrich Gauss in his book Disquisitones Arithmeticae in 1801. Johann Carl Friedrich Gauss was a German mathematician and scientist, born 1777 in Braunschweig and died 1855 in Göttingen. Gauss is considered the Prince of mathematicians and the greatest mathematician since antiquity, he contributed to many areas, such as: number theory, statistics, analysis, differential geometry, geodesy, electrostatics, astronomy and optics.

He was a boy genious, one story tells that his primary school teacher occupied his pupils by making them adding a list of integers. To the teacher's astonishment Gauss produced the right answer within seconds. Gauss presumably realized that pairwise addition from both ends of a list of numbers from say 1 to 100 yields identical sums. The total sum thus being $(100 + 1)\frac{100}{2}$.

His father did not support Gauss' studies in mathematics and instead wanted him to follow him and become a mason. His mother on the other hand supported him, so did the Duke of Braunschweig who awarded him a fellowship to study at the Collegium Carolinum (Technische universität Braunschweig) from where he continued to the University of Göttingen in 1795. At the university Gauss rediscovered several important theorems and proved that any regular polygon with a Fermat prime number of sides can be constructed by a compass and a straightedge. This was an important result with implications for construction problems. Gauss was pleased with his result, so much that he requested that his tombstone be inscribed with a heptadecagon. The stonemason protested claiming that it would look to much like a circle.

In 1796 Gauss invented modular arithmetic, thus greatly simplifying manipulations in number theory, he also conjectured the prime number theorem and proved the quadratic reciprocity law. In 1799 Gauss proved the fundamental theorem of algebra. In his 1801 book Disquisitiones arithmeticae he cleanly presented the modular arithmetic and provided a proof of the quadratic reciprocity law.

Gauss also claimed to have discovered non-Euclidean geometry, but never published his findings. This was to be a paradigm shift in math-

ematics, it challenged the Euclid axioms as the only way to make geometry consistent. Research on non-Euclidean geometries later led to Einstein's theory of general relativity. non-Euclidean geometry was later discovered and published by János Bolyai in 1832. Gauss wrote to Farkas Bolyai, János father and a friend of Gauss

> To praise it would amount to praising myself. For the entire content of the work... coincides almost exactly with my own meditations which have occupied my mind for the past thirty-five years.

Gauss had indeed, before 1829, discussed the problem of parallel lines in letters and had discovered non-Euclidean geometry long before it was published by János but did not publish any of it because of fear of controversy.

Later he collaborated with physics professor Wilhelm Weber which led to new knowledge in magnetism and the discovery of Kirchoff's circuit laws of electricity.

Gauss was a perfectionist and worked hard. It is said that Gauss was once interrupted while working on a problem with the message that his wife was dying. He responded "Tell her to wait a moment till I'm done". He was not a very productive writer, choosing not to publish results he did not think was complete and above criticism. It is said that he attended only one scientific conference. Even though he disliked teaching he did take on a few students, several of them became successful mathematicians; Richard Dedekind, Bernhard Riemann and Friedrich Bessel.

Definition 11. Congruence modulo m.

> a is *congruent* to b modulo n, written $a \equiv b \pmod{n}$ if $\exists k \in \mathbb{Z} : a - b = kn$ for $a, b, n \in \mathbb{Z}$.
>
> If $a \equiv b \pmod{n}$ then $\exists k \in \mathbb{Z} : a - b = kn$.

Theorem 9. Addition and subtraction modulo m.

> If $a \equiv b \pmod{n}$ then $a+j \equiv b+j \pmod{n}$ and $a-j \equiv b-j \pmod{n}$ for $j, a, b, n \in \mathbb{Z}$.

Proof. By definition 11, $a \equiv b \pmod{n}$ implies $\exists k \in \mathbb{Z} : a - b = kn$, and then $a + j \equiv b + j \pmod{n}$ implies $(a + j) - (b + j) = a - b = kn$. Also $a - j \equiv b - j \pmod{n}$ implies $(a - j) - (b - j) = a - b = kn$.

Theorem 10. Multiplication modulo n.

1. If $a \equiv b \pmod{n}$ then $aj \equiv bj \pmod{n}$ for $j, a, b, n \in \mathbb{Z}$.
2. If $j, k \in \mathbb{Z}$, $k > 0$ and $a \equiv b \pmod{n}$ then $a \cdot j^k \equiv b \cdot j^k \pmod{n}$.
3. If $k \in \mathbb{Z}$, $k > 0$ and $a \equiv b \pmod{n}$ then $a^k \equiv b^k \pmod{n}$.

Proof.

1. By definition 11, $a \equiv b \pmod{n}$ implies $\exists k \in \mathbb{Z} : a - b = kn$, and then $aj \equiv bj \pmod{n}$ implies $aj - bj = j(a - b) = jkn$, where we can rewrite jk as k.
2. Apply property 1 k times.
3. First, $a - b$ is always a factor of $a^k - b^k$ so $\exists i \in \mathbb{Z}$ such that $a^k - b^k = (a - b) \cdot i = i \cdot (jn) = (ij) \cdot n$ which implies $a^k \equiv b^k \pmod{n}$.

Theorem 11. Modular arithmetic is an equivalence relation.

Modular arithmetic is reflexive, symmetric and transitive, which means it is an equivalence relation:

1. $a \equiv a \pmod{n}$ (reflexive).
2. if $a \equiv b \pmod{n}$ then $b \equiv a \pmod{n}$ (symmetric).
3. if $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$ then $a \equiv c \pmod{n}$ (transitive).

Proof. We show each property:

1. $a \equiv a \pmod{n}$ implies $a - a = kn$ which is satisfied by $k = 0$.
2. $a \equiv b \pmod{n} \Leftrightarrow a - b = kn$ and $b \equiv a \pmod{n} \Leftrightarrow b - a = -kn$ both satisifying defintion 11.
3. $\exists k, j \in \mathbb{Z} : a - b = kn$ and $b - c = jn$. If $a \equiv c \pmod{n}$ then $a - c = in$ by definition 11. Now $a - c = (a - b) + (b - c) = kn + jn = (k + j)n$ and $i = k + j$ which implies $a \equiv c \pmod{n}$.

Theorem 12. Modular addition and subtraction.

> If $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$ then $a + c \equiv b + d \pmod{n}$ and $a - c \equiv b - d \pmod{n}$.

Proof. By definition 11, $a \equiv b \pmod{n} \Leftrightarrow a - b = kn$ and $c \equiv d \pmod{n} \Leftrightarrow c-d = jn$ so $(a+c)-(b+d) = (a-b)+(c-d) = kn+jn = (k+j)n$ and similarly $(a-c)-(b-d) = (a-b)-(c-d) = kn-jn = (k-j)n$.

Theorem 13. Multiplying moduli.

> If $a \equiv b \pmod{n}$ and $a \equiv b \pmod{m}$ and n and m are relatively prime then $a \equiv b \pmod{nm}$.

Proof. $\exists k, j \in \mathbb{Z} : a-b = km$ and $a-b = jn$. Since m divides $(a-b)$, m must also divide jn, but n and m are relatively prime so m must divide j. Then $\exists l \in \mathbb{Z} : j = ml$ then $a - b = kn = mln = lmn \Rightarrow a \equiv b \pmod{nm}$.

Theorem 14. Cancellation modulo n.

> Given $ac \equiv bc \pmod{n}$ where c and n are relatively prime then $a \equiv b \pmod{n}$.

Proof. $\exists k \in \mathbb{Z} : ac - bc = c(a-b) = kn$. n divides kn so it must also divide $c(a-b)$ but c and n are relatively prime so n must divide $a-b$ and then $\exists j \in \mathbb{Z} : a - b = jn$ thus $a \equiv b \pmod{n}$.

2.6 Fermat's little theorem

> "Cuius rei demonstrationem mirabilem sane detexi hanc marginis exiguitas non caperet (I have discovered a truly remarkable proof of this theorem which this marigin is too small to contain)."
>
> – Pierre de Fermat.

Pierre de Fermat (1601 – 1665) was a French lawyer at the Parlement of Toulouse and also one of the two leading mathematicians of the early 17th century (together with René Descartes). He was a born at Beaumont-de-Lomagne, northwest of Toulouse, his father was a leather merchant and the second consul of Beaumont-de-Lomagne. He went to

the University of Toulouse and moved to Bordeaux in the late 1620s. In Bordeaux he achieved his first serious mathematical results. Later he moved to Orléans to study law. By 1631 he was a lawyer and a government official in Toulouse and was entiteled to change his name from Perre Fermat to Pierre de Fermat.

He stayed in Toulouse for the rest of his life, in 1638 he was promoted from the lower chamber of the parliament to a higher chamber, in 1652 he was promoted again to the highest level at the criminal court. He was preoccupied with mathematics, it was said about Fermat in a report to Colbert that

> Fermat, a man of great erudition, has contact with men of learning everywhere. But he is rather preoccupied, he does not report cases well and is confused.

Fermat shared his mathematical interest with Carcavi, who he met while both were councillors in Touluse. Through Carcavi Fermat became acquainted to Mersenne. Fermat was never interested in publishing his work and considered himself more of an amateur (he is now regarded as the "King of amateurs"). He communicated his results in letters to his friends, and provided little or no proof of his theorems.

Fermat pursued research in analytic geometry, probability, optics, the theory of numbers and is credited for early developments in calculus. His method of finding minima and maxima of curves led to what we now know as differential calculus.

In number theory Fermat studied, among other subjects, diophantine equations, Fermat numbers[1], perfect numbers[2] and amicable numbers[3]. It was when he was studying perfect numbers that he discovered Fermat's little etheorem.

Fermat's famous last theorem was discovered by his son in the margin on his father's copy of an edition of Diophantus. Pierre de Fermat had written in the margin that he had discovered a proof of the theorem but that the margin was too small to include the proof. He had not even bothered to tell Mersenne about the proof. Fermat's last theorem was unproven until 1994. It was proven with techniques unknown to Fermat.

---

[1] A Fermat number is a positive integer of the form $F_n = 2^{2^n} + 1$.

[2] A perfect number is a positive integer which is the sum of its proper positive divisors.

[3] Amicable numbers are two positive integers where one is the sum of the proper divisors of the other.

Perre de Fermat first stated his little theorem in a letter to Frénicle de Bessy as: "p divides $a^{p-1}-1$ whenever p is prime and a is coprime to p". Fermat did not prove his statement, he only said "And this proposition is generally true for all progressions and for all prime numbers; the proof of which I would send to you, if I were not afraid to be too long". Euler published a proof in 1736 and Leibniz provided an equivalent proof in an unpublished manuscript from before 1683. The name "Fermat's little theorem" was coined by Kurt Hensel in his 1913 book Zahlenteorie.

Theorem 15. Fermat's little theorem.

If p is prime and p is not a factor of a then $a^{p-1} \equiv 1 \pmod{p}$.

Proof. consider the set $A = \{a1, a2, \ldots, a(p-2), a(p-1)\}$.

1. $ak \not\equiv 0 \pmod{p}$ for $1 \leq k \leq p-1$. Assume the opposite, if $ak \equiv 0 \pmod{p}$ then $ak - 0 = np$ for some n, since p and a are relatively prime $ak = np$ implies that p divides k but that is impossible since $1 \leq k \leq p$. We conclude that none of the integers in the set A is congruent to $0 \pmod{p}$.

2. $aj \not\equiv ak \pmod{p}$ for $1 \leq k < j \leq p-1$. Assume the opposite, if $aj \equiv ak \pmod{p}$ we have $aj - ak = a(j-k) = np$ for some n. a and p are relatively prime so p must divide $j-k$ but $1 \leq j-k \leq p-1$ so it is impossible, thus $aj \not\equiv ak \pmod{p}$.

3. Consider the simpler set $R = \{1, 2, \ldots p-2, p-1\}$ of representatives of all the equivalence classes of the integers modulo p. Write the set R as $\{r_1, r_2, \ldots r_{p-2}, r_{p-1}\}$ in a possibly different order. Now, $\forall i, 1 \leq i \leq p-1$ we have:

$$
\begin{aligned}
a1 &\equiv r_1 \pmod{p} \\
a2 &\equiv r_2 \pmod{p} \\
&\vdots \\
a(p-2) &\equiv r_{p-2} \pmod{p} \\
a(p-1) &\equiv r_{p-1} \pmod{p}
\end{aligned}
$$

Multiply the left hand sides and the right hand sides to get:

$$\begin{aligned}(a1)(a2)\ldots(a(p-2))(a(p-1)) &= r_1 r_2 \ldots r_{p-2} r_{p-1} \pmod p \\ a^{p-1}(1\cdot 2\ldots(p-2)\cdot(p-1) &= \prod_{i=1}^{p-1} r_i \pmod p \\ a^{p-1}(p-1)! &= (p-1)! \pmod p\end{aligned}$$

Since all factors of $(p-1)!$ are relatively prime to p we can use the cancellation law (theorem 14) to get:

$$a^{p-1} \equiv 1 \pmod p \qquad (2.1)$$

2.7 Euler's theorem

> "Mathematicians have tried in vain to this day to discover some order in the sequence of prime numbers, and we have reason to believe that it is a mystery into which the human mind will never penetrate."
>
> – Leonhard Paul Euler.

Leonhard Paul Euler was born in Basel, Switzerland 1707 and died in S:t Petersburg, Russia in 1783. Euler is considered the first mathematician of the 18th century and one of the greatest mathematicians of all time. Euler made important discoveries in fields as diverse as calculus and graph theory, mechanics, fluid dunamics, optics and astronomy. Much of our modern mathematical notation and terminology was introduced by Euler, particulary in analysis.

Euler's father was a pastor and his mother a pastor's daughter. Euler spent most of his childhood in Riehen, Euler's father was a friend of the Bernoulli family. Johann Bernoulli, then regarded as the foremost mathemathician in Europe, influenced Euler. At the age of thirteen he was enrolled at the university of Basel and graduated in 1723. Euler was studying theology, Greek and Hebrew in order to become a pastor. On Saturday afternoons he received lessons in mathematics by Johann

Bernoulli who discovered Euler's incredible talent. Johann convinced Euler's father that Euler was destined for mathematics. In 1726 Euler received his Ph. D.

Bernoulli's sons were at the Imperial Russian Academy of Sciences in S:t Petersburg. In the summer of 1726 one of the sons died of appendicitis, the other son assumed his position and recommended Euler for his own, now empty, position in physiology. Euler was soon promoted from the medical departement to a position in the mathematics departement. Euler's career flourished and he was appointed professor of physics in 1731 and two years later became head of the mathematics departement. He married in 1734 and moved to Berlin in 1741, where he took a post at the Berlin Academy offered by Frederick the Great of Prussia. Euler stayed in Berlin for 25 years where he wrote over 380 articles and his two most important works were published; the Introductio in analysin infinitorium in 1748 and the Institutiones calculi differentialis in 1748.

Eventually Euler was forced to leave Berlin and returned to Russia after receiving an invitation from the S:t Petersburg Academy, where he spent the rest of his life.

Euler's achievements are too numerous to list and deserve an entire book. He worked in almost all areas of mathematics: geometry, calculus, trigonometry, algebra, number theory, etc. He also studied continuum physics, lunar theory and other fields. He introduced or popularized much of the notation and terminology we use today, most notably the concept of a function $f(x)$ and the trigonometric functions as well as the letter e for the base for the natural logarithm, the greek letter $\sum$ for summation and i, the imaginary unit and the letter π for the ration of a cirle's circumference to its diameter. A notable example from Euler's work on analysis is the Euler identity $e^{i\pi} + 1 = 0$, voted the most beautiful mathematical formula ever by the readers of the Mathematical Intelligencer in 1988.

Euler laid the foundations to graph theory by solving the problem of the seven bridges of Königsberg. Königsberg in Prussia (now Kaliningrad, Russia) was divided by the Pregel river and included two large islands connected by seven bridges.

The problem of the seven bridges of Königsberg was to find a walk through the city that crossed each bridge once, and only once. Euler proved that there is no solution to the problem. He noted that the choice of route on the mainland or on the islands is irrelevant, the only feature of a route that matters is the sequence of bridges crossed. Building on this realization he formulated the problem in the abstract, thus laying

the foundations to graph theory by defining nodes and edges and forming a graph. He found that a walk that traverses each edge once exists if and only if the graph is connected and if there are exactly zero or two nodes of an odd degree, where degree is the number of edges of a node. Such a walk is called an Eulerian path or Euler walk. The city of Königsberg had four nodes of odd degree and cannot have an Eulerian path.

Euler's interest in number theory came from his friend Christian Goldbach at the S:t Petersburg Academy. Much of Euler's early work on number theory was based on the works of Pierre de Fermat. Euler proved Fermat's little theorem and invented the totient function $\Phi(n)$ which is the number of positive integers less than n that is coprime to n. Using the totient function he generalized Fermat's little theorem to Euler's theorem. He also made progress toward the prime number theorem and contributed to the theory of perfect numbers.

Definition 12. The Euler totient function $\Phi(n)$.

> $\Phi(n), n \in \mathbb{Z}, n > 0$ is the number of integers not exceeding n that are relatively prime to n. By definition $\Phi(1) = 1$.

Theorem 16. Totient evaluation 1.

> If p is prime then $\Phi(p) = p - 1$.

Proof. Since p is prime every integer less than p is relatively prime to p.

Theorem 17. Totient evaluation 2.

> If p is prime then $\Phi(p^k) = p^k - p^{k-1} = p^k(1 - \frac{1}{p}) = p^{k-1}(p - 1) = p^{k-1}\Phi(p)$.

Proof. Consider the set of integers $\{1p, 2p, \ldots, (p^{k-1} - 1)p, p^{k-1}p\}$. There are p^{k-1} members, each is a multiple of p and we have $\Phi(p^k) = p^k - p^{k-1} = p^{k-1}(p - 1) = p^{k-1}\Phi(p)$.

Theorem 18. Totient evaluation 3.

> If p and q are two primes, $p \neq q$ then
>
> $$\Phi(p^k q^l) = \Phi(p^k)\Phi(q^l) = p^k q^l (1 - \frac{1}{p})(1 - \frac{1}{q})$$

Proof. Let $n = pq$, p and q are the only prime factors of n. So, all integers less than n, except the multiples of p and the multiples of q, are relatively prime to n. The number of multiples of p (and q) that are less than n are $\frac{n}{p}$ (and $\frac{n}{q}$). And $\frac{n}{pq}$ are the number of multiples of pq that are less than n. Some of the multiples of p include all the multiples of pq and some of the multiples of q include all the multiples of pq. So, we compute $\Phi(n)$ as

$$\Phi(n) = n - \frac{n}{p} - \frac{n}{q} + \frac{n}{pq}$$

now, since we subtract the multiples of pq twice, and add them once, after rewriting, we arrive at

$$\Phi(p^k q^l) = p^k q^l (1 - \frac{1}{p})(1 - \frac{1}{q})$$

which, by theorem 17, is

$$\Phi(p^k q^l) = \Phi(p^k)\Phi(q^l)$$

Theorem 19. Euler's theorem.

If a and n are relatively prime then

$$a^{\Phi(n)} \equiv 1 \pmod{n}$$

Proof. Consider the set R of integers less than n that are relatively prime to n. There are $\Phi(n)$ members of the set R. We write

$$R = \{r_1, r_2, \ldots, r_{\Phi(n)-1}, r_{\Phi(n)}\}$$

Now consider the set $A = \{ar_1, ar_2, \ldots ar_{\Phi(n)-1}, ar_{\Phi(n)}\}$. We claim that no two members of this set are congruent to each other modulo n. If $j \neq k$ but $ar_j \equiv ar_k \pmod{n}$ then $ar_j - ar_k = bn$ for some $b \in \mathbb{Z}$ and $a(r_j - r_k) = bn$ and since a and n are relatively prime we must have $r_j - r_k = cn$ for some $c \in \mathbb{Z}$. This implies that $r_j \equiv r_k \pmod{n}$ which is impossible since all r_i represent distinct equivalence classes.

Now, since r_i as well as a are relatively prime to n, so are the $\Phi(n)$ members of the set A and we can replace the set A with the set R. We write $S = \{s_1, s_2, \ldots, s_{\Phi(n)-1}, s_{\Phi(n)}\}$ as a reordering of the set R where:

$$\begin{aligned}
ar_1 &\equiv s_1 \pmod{n} \\
ar_2 &\equiv s_2 \pmod{n} \\
&\vdots \\
ar_{\Phi(n)-1} &\equiv s_{\Phi(n)-1} \pmod{n} \\
ar_{\Phi(n)} &\equiv s_{\Phi(n)} \pmod{n}
\end{aligned}$$

Multiplying the right hand sides with the left hand sides, we get (note that $\prod_{i=1}^{\Phi(n)} r_i = \prod_{j=1}^{\Phi(n)} s_j$) :

$$\begin{aligned}
(ar_1)(ar_2)\dots(ar_{\Phi(n)}) &\equiv s_1 s_2 \dots s_{\Phi(n)} \pmod{n} \\
a^{\Phi(n)}(r_1 r_2 \dots r_{\Phi(n)}) &\equiv \prod_{i=1}^{\Phi(n)} s_i \pmod{n} \\
a^{\Phi(n)} \prod_{j=1}^{\Phi(n)} r_i &\equiv \prod_{i=1}^{\Phi(n)} s_i \pmod{n}
\end{aligned}$$

Since all of the r_j:s and s_j:s are relatively prime to n we get, by the cancellation law (theorem 14):

$$a^{\Phi(n)} \equiv 1 \pmod{n} \tag{2.2}$$

Chapter 3

Computational Complexity

3.1 Prerequisites

Definition 13. Alphabet.

An *alphabet* is a nonempty finite set (Σ) such that every string formed by elements of the alphabet can be decomposed uniquely into elements of the alphabet.

If we have the alphabet Σ and $n \in \mathbb{Z}^+$ we say

- $\Sigma^0 = \lambda$ where λ is the empty string.
- $\Sigma^n = \{ \ xy \mid x \in \Sigma, y \in \Sigma^{n-1} \}$

Thus Σ^n is the set of all strings formed from Σ of length n.

Definition 14. Kleene star.

The *Kleene star* of an alphabet is the set

$$\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$$

where $n \in \mathbb{Z}+$. We also define Σ^+ as

$$\Sigma^+ = \bigcup_{n=1}^{\infty} \Sigma^n$$

Definition 15. Language.

> A *language* over an alphabet Σ is a subset of Σ^*, i.e. a set of strings formed by the elements of the alphabet Σ.

Definition 16. Grammar.

> A grammar is a tuple (Σ, N, P, σ) where
>
> 1. (Σ, P) is a rewriting system, a tuple where Σ is an alphabet and P is binary relation (nonempty, finite) on Σ^*. The elements of P are *rewrite rules* or *productions*. Instead of writing $(x, y) \in P$ we write $x \to y$.
> 2. N is the set of *non-terminals*, a subset of Σ and $T = \Sigma - N$ is the set of *terminals*.
> 3. The element $\sigma \in N$ is the *starting symbol*.
>
> We say that a word v over Σ is *immediately derivable* from the word u over Σ if there is a production $x \to y$ such that $u = rxs$ and $v = rys$.

Given a grammar G, the language generated by G is the set $L(G) = \{w \in T^* \mid \sigma \to w\}$.

3.2 Turing machine

A *Turing machine* is a mathematical model of a machine. The machine has a finite set of primitive instructions and reads and writes symbols on a tape according to the instructions. A turing machine consists of

1. A *tape*, divided into cells, one after the other. The cells contain symbols selected from an alphabet. The cells can be read and written (owerwritten). The alphabet includes a blank symbol and unwritten cells of the tape are initially filled with blank symbols. The tape can be infinitely extended in both ends.

2. A *head* that can read and write symbols from the alphabet on the tape as well as move the tape left and write one cell at a time.

3. A *table* of instructions. The instructions are tuples

$$(q_i, a_j, q'_j, a'_j, d_k)$$

which are interpreted as

> Given that the machine is in *state* q_i and has read the symbol a_j then do the following (in sequence):
> a) Erase or write a symbol. Write a'_j instead of a_j.
> b) Move the head. If $d_k = \text{R}$ move the tape one step to the right, if $d_k = \text{L}$ move the tape one step to the left, if $d_k = \text{N}$ do not move the tape.
> c) Change state. Change the state to q'_i.

4. A *state register* that records the current state of the table. There is one *start state* which is the initial state of the machine.

The Turing machine can be formally defined, see definition 17 below.

Definition 17. Turing machine.

A *Turing machine* is a tuple,

$$M = (Q, \Gamma, b, \Sigma, \delta, q_0, F)$$

where

1. Q is a finite set of *states*.
2. Γ is a finite set of *symbols*, the *alphabet*.
3. $b \in \Gamma$ is the *blank symbol*.
4. $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of *input symbols*.
5. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, N\}$ is the *transition function*, a partial function (not defined for all $Q \times \Gamma$).
6. $q_0 \in Q$ is the *initial state*.
7. $F \subseteq Q$ is the set of *final states* (or *accepting states*).

The rows of the table of a Turing machine are a representation of the transition function and contains five tuples of the form:

$$(q_i, a_j, q'_i, a'_j, d_k)$$

where $q_i, q_i' \in Q$ where $Q = \{q_k \mid 0 \leq k \leq n\} \cup F$, and F is the set of final states, q_0 is the initial or start state. Also $a_j, a_j' \in \Gamma$ and $\Gamma = \{a_j \mid 0 \leq j \leq m\} \cup \{\mathbf{E}, \mathbf{N}\}$ where **E** means erase the cell (write a blank, b) and **N** means do nothing (write a_j). Finally, $d_k \in \{\mathbf{R}, \mathbf{L}, \mathbf{N}\}$ where **R** means move the tape to the right, **L** move the tape to the left and **N** means do not move the tape.

The example in table 3.1 (Turing's first example, 1936) computes the sequence [0 1 0 1 0 1 ...] i.e. 0, <blank>, 1, <blank>, etc.

q_i	a_j	q_i'	a_j'	d_k
b	b	c	0	**R**
c	b	e	**N**	**R**
e	b	f	1	**R**
f	b	b	**N**	**R**

Table 3.1: Compute [0 1 0 1 0 1 ...]

The Turing machine models *computation* rather than it models a computer. A real machine (computer) is a deterministic finite automaton and is restricted to a finite number of configurations. The Turing machine has infinite memory and is equivalent to a computer with unlimited memory. It can compute anything a real computer can compute. In a finite amount of time it can manipulate only a finite amount of data like a real computer.

In 1937, at Princeton, Alan Turing set out to realize the logical design of the Turing machine. He manufactured his own relays and built a digital multiplier. At the same time Konrad Zuse, in Germany (1938), and Howard Aiken and George Stiblitz (1937) built similar machines. This was the birth of computer science.

> "A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine."
>
> – Alan Mathison Turing

Alan Mathison Turing (1912 – 1964) was a Brithish mathematician. Time magazine has named Turing as one of the 100 most influential people of the 20th century for his contributions. He is considered the father to computer science.

Alan Turing was conceived in India. His parents wanted Alan to be raised in England and moved back moved back to England where he

was born. His talent was reconized early and he loved solving advanced problems. Turing became an atheist as a teenager when his first love, an older fellow student died in bovine tuberculosis after drinking infected cow's milk.

He failed to win a scholarhip at Trinity College in Cambridge because of his lack of interest in classical studies. Instead Alan went to King's College (Cambridge), his second choice, where he graduated in 1934. In 1936 he published his paper "On Computable Numbers, with an Application to the Entscheidungsproblem" where he introduced the Turing machine and proved that such a machine is capable of solving any conceivable mathematical problem if it is representable as an algorithm.

From 1936 to 1938 he spent most of his time at Princeton's Institute for Advanced Study, working under Alonzo Church, and received his Ph. D. in June 1938. He then returned to Cambridge.

During the Second World War he was one of the main participants at Bletchley Park where German ciphers was broken. At Bletchley Park he designed electromechanical machines, *Bombes*, that were one of the main tools used to break the German ciphers.

From 1945 to 1947 Alan was at the National Physical Laboratory working on the design of the Automatic Computing Engine (ACE). In 1946 he published a paper with the first detailed design of a stored program computer. He became disillusioned by the secrecy around the work at Bletchley park which delayed the construction of the ACE and returned to Cambridge.

In 1949 he became the deputy director of the computing laboratory at the University of Manchester where he wrote programs for one of the first true computers, the Manchester Mark 1 and also devised the Turing test. If a machine could fool an interrogator in a separate room that the he had a conversation with a human then the machine could be considered intelligent. The Turing test thus decides if a machine is intelligent.

In 1952, Arnold Murray, an acquaintance of Turing helped an accomplice breaking into Turing's house. Alan reported the crime to the police and in the investigation Turing acknowledged a sexual relation with Murray. Homosexuality was illegal in the United Kingdom so Turing and Murray were charged. Turing had to choose between imprisonment and probation after chemical castration. Turing accepted to avoid jail and was given estrogen injections to reduce libido. His security clearance was revoked as a consequence of the conviction. He could no longer continue as a cryptographic consultant to the military intelligence. Turing com-

mitted suicide by cyanide poisoning in 1954. A half-eaten apple laced with cyanide was next to his bed, where he was found dead.

The computing world's equivalent to the Nobel Prize, the Turing Award, has been awarded annually by the Association for Computing Machinery since 1966.

3.3 Complexity

A *decision problem* is a question, posed in a formal system, with a yes or no answer. For example "given $x, y \in \mathbf{N}$ does x evenly divide y?"

We can describe how to answer the question of a decision problem in an *algorithm*, or a *decision procedure*. One such decision procedure for our divisibility question is long division. If the remainder of the division of y by x is zero then the answer is yes, if not it is no. The decision problem takes a string as input, in our example $x\ y$.

We introduce two terms *time complexity* and *space complexity*. The time complexity of a problem (decision problem) is a measure of the number of steps executed while solving an instance of a problem as a function of the size of the input and space complexity is a measure of the space required to solve an instance of a problem as a function of the size of the input.

Big O notation or asymptotic notation is used to describe the behavior of a function when the argument tends to infinity (sometimes toward a specific value). Big O notation allows us to specify the general behavior of a function, i.e. using a simpler function.

Definition 18. Big O notation.

> We say
>
> $$f(x) = O(g(x)) \text{ as } x \to \infty$$
>
> if and only if there exists a real number $c > 0$ and a real number x_0 such that
>
> $$\forall x > x_0\ |f(x)| \leq c \cdot |g(x)|$$
>
> Thus $f(x)$ is at most a constant times $g(x)$, for sufficiently large x.

We also write

$$\begin{aligned} f(x) = \Theta(g(x)) \quad \text{if} \quad & \exists c_0,\ c_1 > 0,\ x_0\ \forall x > x_0 \\ & c_0 \cdot |g(x)| \leq f(x) \leq c_1 \cdot |g(x)| \\ f(x) = \Omega(g(x)) \quad \text{if} \quad & \exists c > 0,\ x_0\ \forall x > x_0 \\ & |f(x)| \geq c \cdot g(x) \end{aligned}$$

For example, say we have $f(x) = ax^3 + bx^2 + cx + d$. Using big O notation we can describe the growth rate of $f(x)$ as x approaches infinity as $f(x) \in O(n^3)$.

We will abuse the notation and simply write $f(x) = O(g(x))$, thus implicitly mean as $x \to \infty$.

3.4 Classes

If there is an algorithm (a Turing machine, or a computer program) that can decide an input string of length n in at most cn^k steps, i.e. in $O(n^k)$ operations (see definition 18), we say that the problem can be solved in *polynomial time*.

A *deterministic Turing machine* has at most one action to perform for any given situation. A *non-deterministic Turing machine* may have a set of rules that specify more than one action for a given situation.

The current state and current symbol on the tape does not uniquely specify the behavior of a non-deterministic Turing machine. Instead many different transitions may apply for a combination of a state and a symbol. A non-deterministic Turing machine will always select one such transition that will eventually lead to an accepting state (halt). It is a very lucky guesser.

A deterministic Turing machine has a single path of computation whereas the non-deterministic Turing machine has a branching path of computation, a computation tree. The deterministic Turing machine is a special case of the non-deterministic Turing machine. We can view the non-deterministic variant as spawning deterministic Turing machines at each branching point.

We define the set **P** as the set of all languages that can be decided by some deterministic polynomial-time Turing machine (M).

$$\mathbf{P} = \{L \mid L = L(M)\} \tag{3.1}$$

where $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.

The *complexity class* **P** is one of the fundamental complexity classes. It contains all those decision problems which can be solved by a deterministic Turing machine in polynomial time. Cobham's thesis propose that

> A computational problem can be feasibly computed on some computational device only if they can be computed in polynomial time, i.e. if they belong to **P**

We know that some problems not known to be in **P** have feasible solutions and that some problems known to be in **P** do not. However, Cobham's thesis is a useful rule of thumb; problems in **P** are usually tractable.

We define the important complexity class **NP**, the set of all decision problems that can be decided in polynomial time by some *non-deterministic* Turing machine M_n, i.e.

$$\mathbf{NP} = \{L \mid L = L(M_n)\} \tag{3.2}$$

where $L(M_n) = \{w \in \Sigma^* \mid M_n \text{ accepts } w\}$.

Another way to define **NP** is to consider a *verifier*. If the verifier is given a solution to a problem, a *certificate*, the verifier verify if indeed the certificate is a solution to the problem. Then a problem is in **NP** if and only if there exists a verifier for the problem such that it executes in polynomial time. Note that the verifier need only be able to answer "Yes" in polynomial time. The complexity classs **co-NP** is the set of problems whose certificates can be verified with a "No" answer in polynomial time.

We define yet another complexity class, the class of **NP**-complete problems, **NPC**. A problem p is in **NPC** if

1. p can be verified by a deterministic Turing machine in polynomial time.

2. All problems can be transformed, or *reduced* to p in polynomial time.

So, by 1, **NPC** is a subset of **NP**. And, by 2, we have that if we can solve *any* problem in **NPC** in polynomial time then we can solve all problems in **NPC**.

A *reduction* of a problem is a transformation of one problem into another problem. A problem p_1 is reducible to another problem p_2 if there exists solutions to p_2 and they provide solutions to p_1 whenever there are solutions to p_1. Or, formally for the sets $A \subset N$ and $B \subset N$ and a set of functions $F = \{f : N \times N\}$ we have $\exists f \in F \ \forall x \in N \ x \in A \Leftrightarrow f(x) \in B$.

Definition 19. Reducible.

> A language L' over the alphabet Σ is polynomial time *reducible* to L if and only if
>
> 1. $\exists f : \Sigma^* \to \Sigma^*$ such that $\forall w \in \Sigma^* \ w \in L'$ implies $f(w) \in L$.
> 2. There exists a polynomial time Turing machine which halts with $f(w)$ on its tape on any input w.
>
> If a language L' is polynomial time reducible to L we write $L' \leq_p L$.

Definition 20. NP-complete.

> The language L is **NP**-complete if and only if
>
> 1. $L \in \mathbf{NP}$.
> 2. $\forall L' \in \mathbf{NP} \ L \leq_p L'$.
>
> That is, $\mathbf{NPC} = \{L \mid \forall L' \in \mathbf{NP} \ L \leq_p L'\}$.

The class of **NP**-complete problems can be verified in polynomial time but there is no known efficient way of finding a solution. Often, if a solution to a problem can be verified efficiently (**NP**) there is often an efficient algorithm to solve the same problem (**P**). However for the **NP**-complete problems there are no known efficient solution. It has not been proven that no such solution exists nor that it does. If a solution is found to any one of the **NP**-complete problems then all **NP**-complete problems have a solution and we would have that $\mathbf{P} = \mathbf{NP}$. This is one of the most important remaining problems in the field of theoretical computer science.

Chapter 4

Elementary arithmetic

4.1 Representation

A *number* is a mathematical object. A *numeral* is a notational symbol used to denote a number. By combining numerals in a positional numeral system we can express a large number of numbers.

A positional numeral system is a scheme to represent numbers with numerals. A positional numeral system has a base b and numerals for all numbers $0 \ldots b-1$. We write a number X as $(x_n x_{n-1} \ldots x_1 x_0)_b$ which denotes:

$$X = (x_n x_{n-1} \ldots x_1 x_0)_b = \sum_{i=0}^{n} x_i b^i$$

Where every x_i is less than the base b, i.e. $0 \leq x_i < b$.

We (humans) commonly use the base 10, which we will use below to demonstrate some properties of addition, subtraction, multiplication and division of two single digits, which we call *primitive* operations.

Below, we prove two important consequences of our representation. First we show that the expansion of a number X to its base b representation exists and is unique.

Theorem 20. Unique representation of integers.

> There exists a unique representation (expansion) of the number $X \in \mathbb{Z}$.

Proof. Assume $b > 1$ and $X \geq 0$. We prove existence and uniqueness.

1. To show existence we consider the representation or expansion of $X \geq 0$ in the base b:

$$X = (x_k x_{k-1} \ldots x_1 x_0)_b = \sum_{i=0}^{k} x_i b^i \quad 0 \leq x_i < b$$

   $X = 0$ obviously has the expansion $(0)_b$. We use induction to prove the existence of an expansion for $X > 0$ by assuming there exists an expansion for $X - 1$. Then $X - 1$ has the expansion

$$X - 1 = (x_k x_{k-1} \ldots x_1 x_0)_b = \sum_{i=0}^{k} x_i b_i$$

   where $0 \leq x_i < b$ for $x_i \in \mathbb{Z}$ and $x_{k+1} = 0$. Now, add 1 to both sides of the equation, to get

$$X = x_0 + 1 + \sum_{i=1}^{k} x_i b^i$$

   If $x_0 < b - 1$ then we have a representation for X. If not, then $x_0 + 1 = b$ and we write

$$X = 0 + (x_1 + 1)b + \sum_{i=2}^{k} x_i b^i$$

   If $x_1 < b - 1$ then we have a representation for X. If not we repeat the process, continuing until an i for which $x_i < b - 1$ is reached. We know that $x_{k+1} = 0$ so termination is guaranteed. When the process terminates we have an expansion for X in base b.

2. To show uniqueness we again consider the integer X

$$X = (x_k x_{k-1} \ldots x_1 x_0)_b = \sum_{i=0}^{k} x_i b^i, \quad 0 \leq x_i < b$$

We have

$$
\begin{aligned}
x_k b^k &\leq \sum_{i=0}^{k} x_i b^i \leq x_k b^k + \sum_{i=0}^{k-1} (b-1) b^i \\
x_k b^k &\leq \sum_{i=0}^{k} x_i b^i \leq x_k b^k + (b^k - 1) \\
x_k b^k &\leq \sum_{i=0}^{k} x_i b^i < (x_k + 1) b^k
\end{aligned}
$$

which define the intervals $[x_k b^k, (x_k + 1)b^k)$. These intervals are disjoint for every value of x_k and thus no different values of x_k belong in the same interval.

So, X uniquely defines which interval x_k is in. We consider

$$X - x_k b^k = \sum_{i=0}^{k-1} x_i b^i$$

and repeat the previous reasoning with $k - 1$ instead of k. We find that x_{k-1} is also uniquely determined. Then we repeat the process for $X - x_k b^k - x_{k-1} b^{k-1}$ and so on to find that all the x_i are uniquely determined and thus the expansion of X is unique.

We have shown that an expansion exists $\forall X \in \mathbb{Z}$ where $X > 0$ and that it is unique.

The expansion of a negative number, by definition is the negative of the expansion of its absolute value. We know $-X = -|X|$ and by the reasoning above the we also know that a unique expansion exists for $|X|$ since $|X| \geq 0$.

Thus a unique expansion exists $\forall X \in \mathbb{Z}$.

Then we show a useful property of our representation. If we have a number represented in base b we can, simply by grouping its digits, switch bases, if the destination base is a multiple of the source base, i.e. we can go from base b to base b^k.

Theorem 21. From base b to base b^k.

If we have X in an $n+1$ digit base b representation, i.e. $X = (x_n x_{n-1} \ldots x_1 x_0)_b$ then for $k \in \mathbb{Z}$, $k > 0$ we have the base b^k representation of X, $X = (x'_m x'_{m-1} \ldots x'_1 x'_0)_{b^k}$, where $m = \lceil \frac{n+2}{k} \rceil - 1$ and

$$x'_i = \sum_{j=0}^{k-1} x_{ik+j} b^j, \quad 0 \le i \le m-1$$

$$x'_m = \sum_{j=0}^{n-lk} x_{lk+j} b^j$$

Proof. Consider the number X, X is $(x_n x_{n-1} \ldots x_1 x_0)_b$ in base b and X is $(x'_m x'_{m-1} \ldots x'_1 x'_0)_{b^k}$ in base b^k. Thus, we have

$$x'_i = X - q'_i b^{ki} - \sum_{j=0}^{i-1} x'_j b^{kj}$$

where q'_i is given by $X = q'_i b^{ki} + r'_i$. Now, $0 \le x'_i < b^k$ and we can express x'_i in the base b with k digits as $x'_i = (x''_{k-1} x''_{k-2} \ldots x''_1 x''_0)_b$ where x''_u is given by

$$x''_u = x'_i - q''_u b^u - \sum_{v=0}^{u-1} x''_v b^v, \quad 0 \le u \le k$$

and we see that $(x''_{k-1} x''_{k-2} \ldots x''_1 x''_0)_b$ is $(x_{k-1} x_{k-2} \ldots x_1 x_0)_b$. So a number $Y = (y'_{m-1} y'_{m-2} \ldots y'_1 y'_0)_{b^k}$ in base b^k can be represented as $Y = (y_{n-1} y_{n-2} \ldots y_1 y_0)_b$ in base b with $n = mk$ base b digits. Conversely a number Y with n base b digits can be represented as a base b^k number with $m = \lceil \frac{n+1}{k} \rceil + 1$ digits.

Finally, we state a useful consequence of our representation. If we have X as

$$X = (x_n x_{n-1} \ldots x_1 x_0)_b = \sum_{i=0}^{n} x_i b^i$$

Then

$$(x_n x_{n-1} \ldots x_k \; 0 \ldots 0)_b = \sum_{j=k}^{n} x_j b^j \le X \tag{4.1}$$

and

$$(x_n x_{n-1} \dots (x_k + 1)\ 0 \dots 0)_b = \sum_{j=k}^{n} x_j b^j > X \tag{4.2}$$

For example, we have, $x_n b^n \leq X < (x_n + 1)b^n$ and $x_n b^n + x_{n-1} b^{n-1} \leq X < x_n b^n + (x_{n-1} + 1)b^{n-1}$

4.2 Implementation

In this section we discuss the implementation of the multiple precision integer arithmetic operations.

We use a computing machine, a computer. The computer is a binary machine and all computations are done in the binary base. The computer represents an unsigned number as an array of binary digits, *bits*, each with a value of 0 or 1. The *wordsize* is the number of elements, or digits, in the widest such bitarray the computer can use for primitive operations. We are mainly concerned with positive integers, thus we use only unsigned arithmetic.

The computer has some limitiations, however, we make only few assumptions about the actual computing machine used for our implementation. One such assumption is that the computer can represent unsigned numbers between 0 and $2^w - 1$ where w is the wordsize. So $2^w - 1$ is the largest number we can represent using only the basic, primitive, datatypes available on our computer.

We are also assuming that our computer is capable of performing primitive operations such as addition, subtraction, multiplication and division using primitive datatypes, where we are mainly interested in unsigned operations on the unsigned integer datatype.

We further assume that our computer performs these arithmetic operations modulo 2^w. One consequence of this assumption is that if there is an overflow, or underflow, as a result of one of the primitive operations the result will *wrap* and we have *wraparound*.

Elaborating further on the topic of wraparound, overflow and underflow, assume we have two numbers in the largest primitive unsigned integer datatype available, say x and y. Say that we, for example, add x and y to compute the result s, then we have that if $x + y > 2^w - 1$ then the result s of $x + y$ will be $x + y - 2^w$, when s is a w bit bit array. We have seen this in sections 4.4, 4.5, 4.6 and 4.5 when we analysed the

primitive operations addition, subtraction, multiplication and division respectively.

Now, looking at multiplication, the product of two w wide bit arrays (words) requires at most a bit array that can hold the number $(2^w - 1)(2^w - 1)$, which is $2^{2w} - 2^{w+1} + 1$. For example, say we have $w = 8$, then the maximum product of two 8 bit words is $2^{16} - 2^9 + 1$. We will need at least a 16 element bit array to represent the maximum product of two 8 bit numbers.

We have, essentially, two options when it comes to selecting the maximum digit given a computer with a wordsize of w. If we select the maximum digit (base) as $2^{\frac{w}{2}}$, then we can use the computers primitive multiplication without the risk of loosing parts of our result to wraparound. However, if we do, we still have to handle the additions of the products of a multiple precision integer with another multiple precision integer. That is, when we multiply our first factor with a digit from the second factor the result will be an array of digits with a maximum value of $2^w - 1$. This array are then to be added to the rest of the arrays resulting from each of the single digit products. To finally return to our base $2^{\frac{w}{2}}$ representation we need to perform a number of bit manipulations and additions. A consequence of this selection of the base is that the word arrays of the multiple precision integers are only used to half their storage capacity.

If we instead select our base as the maximum wordsize w we will again perform each of the primitive multiplications in the base $2^{\frac{w}{2}}$. And the same bit manipulation will have to be done, but we can utilize the full storage capacity of the word arrays.

Our conclusion is that, given a wordsize of w, we will perform primitive multiplication of digits up to $2^{\frac{w}{2}} - 1$ without the risk of loosing parts of the result because of truncation or wraparound. As we will see later we will implement our primitive multiplication (and division) using *halfwords*, i.e. we will do primitive multiplication and division in the base $2^{\frac{w}{2}}$. And we will represent our multiple precision integers as an array of digits with a maximum value of $2^w - 1$, i.e. in the base 2^w.

We define two functions $l(x)$ and $h(x)$, they are the base $2^{\frac{w}{2}}$ digits of x where $l(x)$ is the least significant part and $h(x)$ the most significant part, i.e. $(x)_{2^w} = (h(x)l(x))_{2^{\frac{w}{2}}}$ and $x = h(x)b^{2^{\frac{w}{2}}} + l(x)$.

A new notation is introduced, we write

$$x = [x_n x_{n-1} \dots x_1 x_0]_w$$

instead of

$$x = (x_n x_{n-1} \dots x_1 x_0)_{2^w}$$

to represent an array of n w bit words.

4.3 Multiple precision arithmetic

We store our multiple precision integers in an array of words with the computer's maximum available wordsize for unsigned arithmetic (w).

So, if we have the number X as

$$X = (x_n x_{n-1} \dots x_1 x_0)_b = \sum_{i=0}^{n} x_i b^i$$

where $x_n \neq 0$ we will store it as shown in figure 4.1.

	0	1	2		$n-1$	n
X:	n	x_n	x_{n-1}	$\cdots$	x_1	x_0

Figure 4.1: Representation

The first element of the array holds the number of available positions for digits in the array, this we will call the *size* of the array. The least significant digit is stored in the rightmost position. We define the *length* of a multiple precision integer as the number of positions (digits), counted from the left, to the last nonzero digit. So, for example, our number X above can be stored as in figure 4.2, where the size of X is $n+3$ and the length of X is n.

	0	1	2	3	4	5		$2+n$	$3+n$
X:	$n+3$	0	0	0	x_n	x_{n-1}	$\cdots$	x_1	x_0

Figure 4.2: Representation, size and length

As indicated we consider the arrays to start at index 0. We will mainly use *pointers* to address the individual elements of the array in our implementation. A pointer is an index into the computers memory, the memory cell used to hold a pointer thus contains an *address* into the

computers memory. We *dereference* the pointer, that is, we access the memory cell it points to by saying `*p` where `p` is a pointer.

We could have chosen not to let our datastructure hold the size of the array. However, storing the size in the zeroth position allows us to cleanly call arithmetic functions without bothering the user (programmer) to manage the size of multiple precision integers and also we can conveniently write `X[*X]` for the size of a multiple precision integer.

4.4 Addition

Addition computes the sum of two terms, $z = x + y$. We examine primitive addition in the decimal base ($b = 10$). That is, let x_i and y_i be two numbers where $0 \leq x_i, y_i < b$. The primitive addition operation computes z_i in $(cz_i)_b = (x_i)_b + (y_i)_b$.

$x_i \backslash y_i$	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

Figure 4.3: Result z_i of primitive addition $(x_i)_b + (y_i)_b = (cz_i)_b$ in base $b = 10$

.

The maximum value of a digit is $b - 1$ so the maximum result of primitive addition is $(b-1) + (b-1) = 2b - 2 = 1b^1 + (b-2)b^0$, in base 10 we have $(x_i)_{10} + (y_i)_{10} \leq 18$.

In figure 4.3 we see that if $(x_i)_b + (y_i)_b \geq b$ then $(z_i)_b$ is $(x_i)_b + (y_i)_b - b$ (shaded cells) and we need two digits to represent the actual result. Also, it is clear that if one of the terms is zero the sum is less than b and can be represented in a single digit. We also see that whenever $x_i + y_i \geq b$

the resulting z_i is less than x_i and also less than y_i, i.e. if $x_i + y_i \geq b$ then $z_i < x_i$ and $z_i < y_i$.

Long addition

Long addition is the algorithm we recognize as the standard pen and paper method for addition of large numbers. It consists of tabulating the terms, one above the other (figure 4.4), and then performing primitive additions with carry over into the next primitive addition.

$$\begin{array}{cccccccc} & & x_n & x_{n-1} & \cdots & x_1 & x_0 \\ + & & y_n & y_{n-1} & \cdots & y_1 & y_0 \\ \hline & z_{n+1} & z_n & z_{i-1} & \cdots & z_1 & z_0 \end{array}$$

Figure 4.4: Long addition ($Z = X + Y$).

We examine, more closely, the addition of two numbers in a base b. Consider two numbers X and Y:

$$X = (x_n x_{n-1} \ldots x_1 x_0)_b = \sum_{i=0}^{n} x_i b^i$$

$$Y = (y_n y_{n-1} \ldots y_1 y_0)_b = \sum_{i=0}^{n} y_i b^i$$

Adding X and Y to get the sum $Z = X + Y$:

$$Z = (z_{n+1} z_n \ldots z_1 z_0)_b = \sum_{i=0}^{n+1} z_i b^i$$

consists of calculating the primitive additions of each x_i and y_i. Whenever $x_i + y_i \geq b$, which would give $z_i \geq b$, violating our representation, we must also *carry* the *overflow* into the next digit z_{i+1}. Since we know that $(x_i)_b + (y_i)_b \leq 2b - 2$ we know that the overflow which we must carry over from digit i to digit $i + 1$ will be at most b which translates into adding 1 to digit z_{i+1}.

So, for addition base b let

$$c_i = \begin{cases} 0 & \text{if } x_i + y_i < b \\ 1 & \text{if } x_i + y_i \geq b \end{cases}$$

then $(c_i z_i)_b = (x_i)_b + (y_i)_b + (c_{i-1})_b$ and $c_0 = 0$.

We can make use of the fact that if $(cz_i)_b = (x_i)_b + (y_i)_b$ and $x_i + y_i \geq b$ then $z_i < x_i$ and $z_i < y_i$ to detect such an overflow.

Algorithm 1. Addition.

The addition algorithm adds two positive integers to produce the sum.

In: Integers $A = (a_n a_{n-1} \dots a_1 a_0)_b$ and $B = (b_n, b_{n-1} \dots b_1 b_0)_b$ where A and B are greater than zero and each have $n + 1$ digits in the base b.

Out: The sum, $S = A + B = (s_{n+1} s_n \dots s_1 s_0)_b$.

1: $c = 0$
2: **for** $i = 0 \dots n$ **do**
3: $s_i = (a_i + y_i + c) \bmod b$
4: **if** $s_i < b$ **then**
5: $c = 0$
6: **else**
7: $c = 1$
8: **end if**
9: **end for**
10: **return** $S = (s_{n+1} s_n \dots s_1 s_0)_b$

The addition algorithm (algorithm 1) executes n primitive additions to compute its result. The computational complexity of the addition algorithm is linear, i.e. addition belongs to $O(n)$.

Primitive addition

The primitive addition is the simplest of our primitive operations. We know that the sum of two base b words is at most $2b - 2$, our words are w bits long and our base is 2^w. We can represent the sum of two w bit words with $w + 1$ bits since we need one extra bit for the carry.

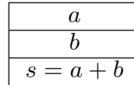

Figure 4.5: Addition registers.

The sum s is the sum of x, y and the carry from the previous iteration, c. If there was an overflow in the addition $x + y$ then we set the carry for the next iteration (line 6 in listing 4.1).

We have $[x]_w + [y]_w + [c]_1 = [s]_w$, where $[c]_1$ is the one bit carry from the previous operation. If, after the addition, $[s]_w < [x]_w$, $[s]_w < [y]_w$ or if $[s]_w < [c]_1$ then we set the new carry to 1, else to 0.

```
void
mp_add_p(mp_limb_t *s, mp_limb_t *c,
        mp_limb_t x, mp_limb_t y)

        *s = *x + *y + c;
        *c = (*s < *x) || (*s < *y) ||
                (*s < c) ? 1 : 0;
}
```

Listing 4.1: Primitive addition

Multiple precision addition

Addition consist of repeatedly performing primitive addition of the digits of the same positional value of the terms and adding in the carry from the previous operation. The size of the sum must be at least the length of the largest of the terms. Thus the sum will be large enough to hold the sum of the terms, with the exeption of the carry from the final primitive addition which is returned. Since the terms may be of different lengths the implementation must handle the different cases, i.e. $len(X) \leq len(Y)$ or $len(X) > len(Y)$. We also want the addition to be in-place safe, that is, we expect the statement `X = X + Y` to provide a meaningful result.

We start by adding the digits of the terms up til the length of the smallest term. Continuing we only have to add in the carry from the first part, and follow it through to the end of the longest term. If the

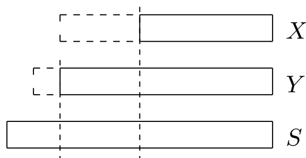

Figure 4.6: Sum (S) and terms (X and Y).

sum is larger than the longest of the terms we must also zero out the remaining digits of the sum.

Looking at listing 4.2, we start by finding the lengths of X and Y and the minimum length of X and Y on lines 8 to 11. If the size of S is less than the length of X or the length of Y then we can not compute the sum and an error is generated (line 13). On lines 16 to 18 pointers to the digits of S, X and Y are initialized, the pointers s, x and y initially point to the least significant digit of the sum and the terms. Also the initial carry is set to zero at line 20. Now the initializations are done and we continue with the main task of adding the terms together.

```
mp_limb_t
mp_add(mp_t S, mp_t X, mp_t Y)
{
        mp_limb_t c, t;
5       mp_size_t i, k, n, m, AL, BL;
        mp_limb_t *s, *x, *y, *p;

        XL = mp_len(A);
        YL = mp_len(B);
10
        n = XL < YL ? XL : YL;

        if (*S < XL || *S < YL)
                MP_ERR(MP_ERR_RESSIZEOPLEN);
15
        s = S + *S;
        x = X + *X;
        y = Y + *Y;
```

```
20          c = 0;

            for (i = 0; i < n; i++) {

                    t = *x + c;
25                  c = ((t < *x) || (t < c)) ? 1 : 0;
                    t += *y;
                    c += t < *y ? 1 : 0;
                    *s = t;
                    s--;
30                  x--;
                    y--;
            }

            if (XL > YL) {
35                  p = x;
                    m = XL;
            } else {
                    p = y;
                    m = YL;
40          }

            for (i = n; i < m; i++) {

                    t = *p + c;
45                  c = ((t < *p) || (t < c)) ? 1 : 0;
                    *s = t;
                    s--;
                    p--;
            }
50
            if (*S > m) {

                    *s = c;
                    s--;
55                  m++;

                    for (i = m; i < *S; i++) {
```

```
                          *s = 0;
60                        s--;
                   }
          }

          return c;
65 }
```

Listing 4.2: Addition

We are going to add all the digits of the shortest term to the corresponding digits of the longest term to form the n first digits of the sum, thus we iterate from 0 (least significant digit) to digit $n-1$. The digit sum is computed in a temporary register t. First (line 24) the carry is added to the current digit of the term X. At the beginning of the loop c holds the carry from the previous iteration. If an overflow occurs, i.e. if x contained $b-1$ and c is 1, the carry is set to 1 else it is cleared (set to 0), now c represents the current carry instead (line 25). Next, a digit from the other term (Y, line 26) is added and again we check for overflow on line 27. Finally, the digit sum is stored as the resulting digit of S (line 28) and the pointers s, x and y are decreased so that they point to the next more significant digit of S, X and Y respectively (lines 29 to 31), and we are ready for the next iteration.

After line 32 the major work of the addition is complete. What remains is to add the carry to the remaining digits of the longest term and repeatedly check for overflow until we are done with all the digits of the longest term.

First we find the length of the longest term (lines 34 to 40) and let the pointer p point to the first remaining digit of the longest term, i.e. digit n. We save the length of the longest term in m. So, we iterate from n to $m-1$, i.e. $n-m$ times (line 42).

Adding in the carry follows the same patterns as the main addition. The previous carry is added to the term and saved in t (line 44), we check for overflow (line 45) and either set or clear the carry which now represents the carry from this iteration. The resulting digit of the sum is stored (line 46) and the pointers are decreased (lines 47 to 48).

Now, the last remaining task is to take care of the final carry and possibly clear (zero) remaining digits of the sum. First we check if we have to do anything, i.e. if S is longer than m (line 51). If so we save the final carry in digit s, step the pointer and increase m (lines 53, 55 and 54 respectively). We now continue to clear the rest of the digits in

S, thus we start at m and go from m to $*S - 1$, setting each digit to zero and stepping the pointer as we iterate, lines 57 to 60).

Finally, we leave the result in S and return the last carry (line 64). The return value will only ever be set if S is the same size as the length of the longest term. If S is longer the carry will be stored in the most significant word of the sum and the return value will be zero.

4.5 Subtraction

Subtraction computes the difference of two terms $z = x - y$. We examine primitive subtraction in the decimal base ($b = 10$). Let x_i and y_i be two numbers $0 \leq x_i, y_i < b$. The primitive subtraction operation computes z_i in $(cz_i)_b = (x_i)_b - (y_i)_b$.

$x_i \backslash y_i$	0	1	2	3	4	5	6	7	8	9
0	0	9	8	7	6	5	4	3	2	1
1	1	0	9	8	7	6	5	4	3	2
2	2	1	0	9	8	7	6	5	4	3
3	3	2	1	0	9	8	7	6	5	4
4	4	3	2	1	0	9	8	7	6	5
5	5	4	3	2	1	0	9	8	7	6
6	6	5	4	3	2	1	0	9	8	7
7	7	6	5	4	3	2	1	0	9	8
8	8	7	6	5	4	3	2	1	0	9
9	9	8	7	6	5	4	3	2	1	0

Figure 4.7: Result z_i of primitive subtraction $(x_i)_b - (y_i)_b = (cz_i)_b$ in base $b = 10$

.

The minimum value of a digit is 0 and and the maximum value is $b-1$ so the minimum result of a primitive subtraction is $0 - (b - 1) = 1 - b$. In base 10 we have $(x_i)_{10} - (y_i)_{10} \geq -9$.

In figure 4.7 we see that if $(x_i)_b < (y_i)_b$ then $(z_i)_b$ is $(x_i)_b - (y_i)_b + b$ (shaded cells) and X must have at least one additional digit more significant than x_i for the result to be positive. Also, it is clear that if y_i is zero the difference is equal to x_i and if the terms are equal the result is zero. We also see that whenever $x_i < y_i$ the resulting z_i is greater than x_i, i.e. if $x_i < y_i$ then $z_i > x_i$.

Long subtraction

Long subtraction is the algorithm we recognize as the standard pen and paper method for subtracting a large number from another large number. It, just like the long addition, consists of tabulating the terms one above the other (figure 4.8), and then performing primitive subtractions with borrowing from the next digit (next primitive subtraction).

$$\begin{array}{cccccc} & x_n & x_{n-1} & \cdots & x_1 & x_0 \\ - & y_n & y_{n-1} & \cdots & y_1 & y_0 \\ \hline & z_n & z_{n-1} & \cdots & z_1 & z_0 \end{array}$$

Figure 4.8: Long subtraction ($Z = X - Y$).

We examine, more closely, the subtraction of a number from another number in a base b. Consider two numbers X and Y, where $X \geq Y$:

$$X = (x_n x_{n-1} \ldots x_1 x_0)_b = \sum_{i=0}^{n} x_i b^i$$

$$Y = (y_n y_{n-1} \ldots y_1 y_0)_b = \sum_{i=0}^{n} y_i b^i$$

Subtracting Y from X to get the difference $Z = X - Y$:

$$Z = (z_n z_{n-1} \ldots z_1 z_0)_b = \sum_{i=0}^{n} z_i b^i$$

consists of calculating the primitive subtractions of each y_i from x_i. Whenever $x_i < y_i$, we have an *underflow* which would give $z_i < 0$, violating our representation, we must *borrow* from the next digit z_{i+1}.

Since we know that $(x_i)_b - (y_i)_b \geq -(b-1)$ we know that the underflow which we must borrow from the next digit (digit $i + 1$) is at most $b - 1$. This translates into subtracting one from digit z_{i+1}. So, for subtraction base b let

$$c_i = \begin{cases} 0 & \text{if } x_i \geq y_i \\ -1 & \text{if } x_i < y_i \end{cases}$$

We make use of the fact that if $(cz_i)_b = (x_i)_b - (y_i)_b$ and $x_i < y_i$ then $z_i > x_i$ to detect such an underflow.

Algorithm 2. Subtraction.

The subtraction algorithm computes the difference between two positive integers.

In: Integers $A = (a_n a_{n-1} \ldots a_1 a_0)_b$ and $B = (b_n, b_{n-1} \ldots b_1 b_0)_b$ where A and B are greater than zero, $A \geq B$ and each with $n+1$ digits in the base b.

Out: The difference, $D = A - B = (d_n d_{n-1} \ldots d_1 d_0)_b$.

1: $c = 0$
2: **for** $i = 0 \ldots n$ **do**
3: $d_i = (a_i - y_i + c) \bmod b$
4: **if** $d_i > 0$ **then**
5: $c = 0$
6: **else**
7: $c = -1$
8: **end if**
9: **end for**
10: **return** $d = (d_n d_{n-1} \ldots d_1 d_0)_b$

The subtraction algorithm (algorithm 2) executes n primitive subtractions to compute its result. The computational complexity of the subtraction algorithm is linear, i.e. subtraction belongs to $O(n)$.

Primitive subtraction

Primitive subtraction is very similar to primitive addition. We know that the difference between two base b words it no larger than $b-1$ so we know we can represent the difference of two w bit words in $w+1$ bits, i.e. the difference and a carry.

a
b
$d = a - b$

Figure 4.9: Subtraction registers.

The difference d is the difference of x and y with the carry (borrow) from the previous iteration, so we have $d = x - y - c$. If there is an

underflow in the subtraction we set the carry for the next iteration (line 6 in listing 4.3).

We have $[x]_w - [y]_w - [c]_1 = [d]_w$, where $[c]_1$ is the one bit carry from the previous operation. If, after the subtraction, $[s]_w > [x]_w$, $[s]_w > [y]_w$ or if $[s]_w > [c]_1$ then we set the new carry to 1, else to 0.

```
void
mp_sub_p(mp_limb_t *d, mp_limb_t *c,
        mp_limb_t x, mp_limb_t y)

        *d = *x - *y -c;
        *c = (*d > *x) || (*d > *y) ||
                (*d > c)? 1 : 0;
}
```

Listing 4.3: Primitive subtraction

Multiple precision subtraction

Subtraction, like addition, consists of repeatedly performing the primitive operation on the digits of the same positional value of the terms and handling the carry from the previous operation. The size of the difference must be at least the length of the first operand (*minuend*), and the minuend must be at least as large as the *subtrahend*. The implementation must be able to handle the case where the subtrahend is shorter than the minuend. We also want the subtraction to be in-place safe, thus we expect the statement `X = X - Y` to provide a meaningful result.

We start by subtracting the digits of the subtrahend from the digits of the minuend up until the length of the subtrahend. Continuing we only have to subtract the carry from the first part, and follow it through to the end of the minuend. If the size of the difference is even larger we must also continue to zero out the remaining digits of the difference.

Looking at listing 4.4, we start by finding the lengths of X and Y and the minimum length of X and Y, on lines 8 to 11. If the size of D is less than the length of X or the length of Y then we can not compute the difference and an error is generated (line 13). Furthermore, if $X < Y$ then the difference would be negative and we generate an error (line 17). On lines 21 to 23 the pointers to the digits of D, X and Y are initialized. The pointers d, x and y initially point to the least significant digit of

the difference, the minuend and the subtrahend. The initial carry is set to zero on line 25. Now the initializations are complete and we continue with the main task of subtracting.

```
void
mp_sub(mp_t D, mp_t X, mp_t Y)
{
        mp_size_t i, n, XL, YL;
        mp_limb_t c0, c1, t;
        mp_limb_t *d, *x, *y;

        XL = mp_len(X);
        YL = mp_len(Y);

        n = XL < YL ? XL : YL;

        if (*D < XL || *D < YL) {
                MP_ERR(MP_ERR_RESSIZEOPLEN);
        }

        if (mp_lt(X, Y)) {
                MP_ERR(MP_ERR_NEG);
        }

        d = D + *D;
        x = X + *X;
        y = Y + *Y;

        c0 = c1 = 0;

        for (i = 0; i < n; i++) {

                t = *x;
                c1 = t < c0 ? 1 : 0;
                t -= c0;
                c1 += t < *y ? 1 : 0;
                t -= *y;
                *d = t;
                d--;
                x--;
```

```
                y--;
                c0 = c1;
        }
40
        for (i = n; i < XL; i++) {

                c1 = *x < c0 ? 1 : 0;
                *d = *x - c0;
45              c0 = c1;
                d--;
                x--;
        }

50      if (*D > XL) {
                for (i = XL; i < *D; i++) {

                        *d = 0;
                        d--;
55              }
        }
}
```

Listing 4.4: Subtraction

We are going to subtract all the digits of the subtrahend from the corresponding digits of the minuend to form the first n digits of the difference, we start from 0, the least significant digit, and iterate n times, arriving at $n - 1$. The digit differences are computed in a temporary register t. At the beginning of the loop the carry from the previous iteration is held in c_0. First (line 29) the digit of the minuend is stored in t, then, on line 30, we check if the result of subtracting the carry from the current minuend digit will result in an underflow. If so, then the new carry is set to one, else it is set to zero. Then (line 31) the carry is subtracted from the current digit. Next we check if the result of subtracting the subtrahend digit from t will result in an underflow, if so, we set the carry. Note that the carry will only be set once, either on line 30 or on line 32. We subtract the digit from the subtrahend from t (line 33) and store the result in the current digit of the difference on line 34. Finally we decrease the pointers d, x and y, set the carry so that c_0 contains the carry from this iteration, and we are ready for the next iteration.

After line 39 the major work of the subtraction is complete. What remains is to subtract the carry from the remaining digits of the minuend, repeatedly checking for underflow, until we are done with all the digits of the minuend.

We iterate from n to X_L, where X_L is the length of the minuend, line 41. Subtracting the carry follows the same pattern as the main subtraction. We check if the subtraction of the previous carry will result in an underflow (line 43), if it does we set the next carry, if not we clear the next carry. Then we subtract the previous carry from the current digit of the minuend (line 44), set c_0 to c_1 so that the carry from this iteration will be the carry from the previous iteration in the next iteration (line 45), finally we decrease the pointers and we are ready for the next iteration (lines 46 and 47).

Now, the last remaining task is to clear (zero) the remaining digits of the difference. First we check if we have to do anything, i.e. if D is longer than X_L (line 50). If so, then we iterate from X_L to the size of D ($*D$, line 51), setting each remaining digit of the difference to zero and decreasing the pointer as we iterate, lines 53 and 54.

Finally, we leave the result in D. There will never be a remaining carry since we require that $X \geq Y$.

4.6 Multiplication

Multiplication computes the product of two factors. For two natural numbers a and b multiplication is the repeated addition of a b times: $a + a + \cdots + a = a \cdot b$.

We examine primitive multiplication in the decimal base ($b = 10$). Let x_i and y_i be two numbers $0 \leq x_i, y_i < b$. The primitive multiplication operation computes z_i in $(cz_i)_b = (x_i)_b \cdot (y_i)_b$.

Since the maximum value of a digit is $b - 1$ the maximum value of the product of two single digit factors (primitive multiplication) is $(b-1)\cdot(b-1) = b^2 - 2b + 1$. To represent $b^2 - 2b + 1 = (b-2)b^1 + 1b^0$ we need two digits. It is also obvious (figure 4.10) that if one of the factors is zero then the product is also zero and if one of the factors is one the product is equal to the other factor.

$x_i \backslash y_i$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	0	2	4	6	8
3	0	3	6	9	2	5	8	1	4	7
4	0	4	8	2	6	0	4	8	2	6
5	0	5	0	5	0	5	0	5	0	5
6	0	6	2	8	4	0	6	2	8	4
7	0	7	4	1	8	5	2	9	6	3
8	0	8	6	4	2	0	8	6	4	2
9	0	9	8	7	6	5	4	3	2	1

Figure 4.10: Result z_i of primitive multiplication $x_i \cdot y_i = (cz_i)_b$ in base $b = 10$

.

Long multiplication

Long multiplication is the algorithm we recognize as the standard pen an paper method of multiplying two large numbers with each other. It consists of tabulating the two numbers, one above the other (figure 4.11 and figure 4.12), and then repeatedly performing primitive multiplication to produce the intermediate products which are finally added together.

$$\begin{array}{rcccc} & x_n & \cdots & x_1 & x_0 \\ \cdot & & & & y_0 \\ \hline & z_n & \cdots & z_1 & z_0 \end{array}$$

Figure 4.11: Long multiplication ($Z = X \cdot y_0$).

We start by examining multiplication of an $n + 1$ digit number with a single digit number (figure 4.11). Consider the two numbers

$$X = (x_n x_{n-1} \ldots x_1 x_0)_b = \sum_{i=0}^{n} x_i b^i$$

and $Y = (y_0)_b$. Multiplying X and Y to get the product $Z = X \cdot Y$:

$$Z = \sum_{i=0}^{n} x_i y_0 b^i = (z_{n+1} z_n \ldots z_1 z_0)_b = \sum_{i=0}^{n+1} z_i b^i$$

consists of calculating the primitive multiplications of each x_i with y. Whenever $x_i \cdot y \geq b$, which would give $z_i \geq b$, violating our representation, we must also carry the overflow into the next digit.

Since we know that $(x_i)_b \cdot (y)_b \leq b^2 - 2b + 1 = (b-2)b + 1$ we also know that the overflow which we must carry over from digit i to digit $i+1$ will be at most $(b-2)b^1$ which translates into adding c, where c satisfies $x_i \cdot y_i = cb + r$, to digit $i+1$. So, for multiplication base b, let

$$c_i = \lfloor \frac{x_i \cdot y_i + c_{i-1}}{b} \rfloor$$

then $(z_i)_b = (x_i)_b \cdot (y_i)_b + (c_{i-1})_b$ and $c_0 = 0$.

$$\begin{array}{cccccccc}
 & & & x_n & \cdots & x_1 & x_0 \\
 & & \cdot & y_m & \cdots & y_1 & y_0 \\
\hline
 & & & z_{(0,n)} & \cdots & z_{(0,1)} & z_{(0,0)} \\
 & & z_{(1,n)} & \cdots & z_{(1,1)} & z_{(1,0)} & \\
 & & & & & & \\
+ & z_{(m,n)} & \cdots & z_{(m,1)} & z_{(m,0)} & & \\
\hline
 & z_{m+n+1} & \cdots & & \cdots & z_1 & z_0
\end{array}$$

Figure 4.12: Long multiplication ($Z = X \cdot Y$).

Now, for multiplication of two numbers X and Y, where

$$X = (x_n x_{n-1} \cdots x_1 x_0)_b = \sum_{i=0}^{n} x_i b^i$$

$$Y = (y_m y_{m-1} \cdots y_1 y_0)_b = \sum_{j=0}^{m} y_j b^j$$

we repeatedly, for each y_j, apply the technique we used above to multiply an $n+1$ digit number with a single digit number (see figure 4.12) to get

$m + 1$ numbers Z_j where

$$Z_j = (z_{(j,n+1)} z_{(j,n)} \dots z_{(j,1)} z_{(j,0)})_b = \sum_{i=0}^{n+1} z_{(j,i)} b^i$$

and finally summing them multiplied by their respective base multiplier to get the $m + n + 2$ digit number:

$$Z = \sum_{k=0}^{m} Z_k b^k = (z_{m+n+1} z_{m+n} \dots z_1 z_0)_b$$

Examining primitive multiplication in base b^k we find that $xy \leq (b^k - 1)(b^k - 1)$ which we can write $b^{2k} - 2b^k + 1$. Now, if we have X and Y in base b^k as

$$X = (x_n x_{n-1} \dots x_1 x_0)_{b^k} = \sum_{i=0}^{n} x_i (b^k)^i$$

$$Y = (y_m y_{m-1} \dots y_1 y_0)_{b^k} = \sum_{j=0}^{n} y_j (b^k)^i$$

then we can write, by theorem 21:

$$x_i = (x'_1 x'_0)_{b^{k-1}}$$

$$y_j = (y'_1 y'_0)_{b^{k-1}}$$

So, multiplying $x_i \cdot y_i$ then consists of computing $(x'_1 x'_0)_{b^{k-1}} \cdot (y'_1 y'_0)_{b^{k-1}}$ which is

$$\sum_{i=0}^{1} \sum_{j=0}^{1} x'_i y'_j (b^{k-1})^{i+j} = x'_1 y'_1 (b^{k-1})^2 + (x'_1 y'_0 + x'_0 y'_1) b^{k-1} + x'_0 y'_0$$

We note that $(y)_{b^{k-1}} \cdot (x)_{b^{k-1}} = cb^{k-1} + z$ which we can represent as $(cz)_{b^k} = (x \cdot y)_{b^k}$. So we have $(x'_i)_{b^{k-1}} \cdot (y'_j)_{b^{k-1}} = (x'_i \cdot y'_j)_{b^k}$ for $0 \leq i \leq 1$ and $0 \leq j \leq 1$. Now taking the sum

$$\sum_{i=0}^{1} \sum_{j=0}^{1} (x'_i y'_j)_{b^k} (b^k)^{i+j}$$

we get

$$(x'_1 \cdot y'_1)_{b^k}(b^k)^2 + (x'_1 \cdot y'_0 + x'_0 \cdot y'_1)_{b^k} b^k + (x'_0 \cdot y'_0)_{b^k}$$

So we have $(x)_{b^k} \cdot (y)_{b^k} = (z_2 z_1 z_0)_{b^k}$, where:

$$\begin{aligned}
(z_0)_{b^k} &= (x'_0)_{b^{k-1}} \cdot (y'_0)_{b^{k-1}} \\
cb^k + (z_1)_{b^k} &= (x'_1)_{b^{k-1}} \cdot (y'_0)_{b^{k-1}} + (x'_0)_{b^{k-1}} \cdot (y'_1)_{b^{k-1}} \\
(z_2)_{b^k} &= (x'_1)_{b^{k-1}} \cdot (y'_1)_{b^{k-1}} + c
\end{aligned}$$

We can now compute $(x)_{b^k} \cdot (y)_{b^k}$ by computing $(x'_1 x'_0)_{b^{k-1}} \cdot (y'_1 y'_0)_{b^{k-1}}$. This will be useful.

Algorithm 3. Multiplication.

The multiplication algorithm computes the product of two positive integers.

In: Two integers, A and B, with $n+1$ and $m+1$ base b digits respectively, where $A = (a_n a_{n-1} \dots a_1 a_0)_b$ and $B = (b_m, b_{m-1} \dots b_1 b_0)_b$ such that $A, B \geq 0$, $A \geq B$,

Out: The product, $P = A \cdot B = (p_{n+m+1} p_{n+m} \dots p_1 p_0)_b$.

1: **for** $i = 0 \dots n + m + 1$ **do**
2: $\quad p_i = 0$
3: **end for**
4: **for** $i = 0 \dots m$ **do**
5: $\quad c = 0$
6: $\quad$ **for** $j = 0 \dots n$ **do**
7: $\quad\quad (uv)_b = p_{i+j} + a_j \cdot b_j + c$
8: $\quad\quad p_{i+j} = v$
9: $\quad\quad c = u$
10: $\quad$ **end for**
11: $\quad p_{i+j+1} = u$
12: **end for**
13: **return** $P = (p_{n+m+1} p_{n+m} \dots p_1 p_0$

The multiplication algorithm (algorithm 3) executes $(n + 1)(m + 1)$ primitive multiplications to compute its result. The computational complexity of the algorithm is quadratic, i.e. our multiplication algorithm, algorithm 3, belongs to $O(n^2)$. There are better algorithms.

		x_1	x_0
	$\cdot$	y_1	y_0
		$x_1 \cdot y_0$	$x_0 \cdot y_0$
$+$	$x_1 \cdot y_1$	$x_0 \cdot y_1$	
	$x_1 \cdot y_1$	$x_1 \cdot y_0 +$ $x_0 \cdot y_1$	$x_0 \cdot y_0$

Figure 4.13: Primitive multiplication base $b = 2^{\frac{w}{2}}$.

Primitive multiplication

Remember that we in section 4.6, calculated the product of two base b^k numbers by representing them as four base b^{k-1} numbers and computed their product. Also, recall that the maximum product of two base b numbers is $b^2 - 2b + 1$ (section 4.6), thus we know that it is always possible to store the result of the product of two w bit words in two w bit words.

The goal is to compute $p = [p_1 p_0]_w$, the two w bit words that is the result of the multiplication of x and y. We have that

$$p_1 = x_1 \cdot y_1 + h(x_1 \cdot y_0 + x_0 \cdot y_1)$$
$$p_0 = l(x_1 \cdot y_0 + x_0 \cdot y_1) + x_0 \cdot y_0$$

The two factors x and y are in base 2^w. We divide them into four base $2^{\frac{w}{2}}$ numbers, $x_1 = h(x)$, $x_0 = l(x)$, $y_1 = h(y)$ and $y_0 = l(y)$ (lines 9 to 13 in listing 4.5). The multiplication of the two base $2^{\frac{w}{2}}$ numbers $(x_1 x_0)_{2^{\frac{w}{2}}}$ and $(y_1 y_0)_{2^{\frac{w}{2}}}$ is performed according to our multiple precision multiplication as in figure 4.10.

First the intermediate values $t = x_1 \cdot y_0$, $u = x_0 \cdot y_1$ and $v = t + u$ are computed, where t and u are base 2^w numbers in w bit words. It is v, the sum of t and u, that we are primarily interested in. We divide v into two base $2^{\frac{w}{2}}$ numbers v_1 and v_0 so that $h(v_1) = 0$, $l(v_1) = h(v)$ and $h(v_0) = l(v)$, $l(v_0) = 0$ (lines 19 to 20), we then have $v_1 = [0\ h(v)]_{\frac{w}{2}}$ and $v_0 = [l(v)\ 0]_{\frac{w}{2}}$, see figure 4.14.

If $t+u$ resulted in an overflow we add $2^{\frac{w}{2}}$ to v_1 as a carry (lines 22 and 23). Since v_1 is a $\frac{w}{2}$ bit integer in a w bit word there is room for the carry. Now v_1 holds $[c\ h(x_1 \cdot y_0 + x_0 \cdot y_1)]_{\frac{w}{2}}$ and v_0 holds $[l(x_1 \cdot y_0 + x_0 \cdot y_1)\ 0]_{\frac{w}{2}}$. We compute p_0 as $y_0 \cdot x_0 + v_0$ and p_1 as $x_1 \cdot y_1 + v_1$ (lines 25 to 27).

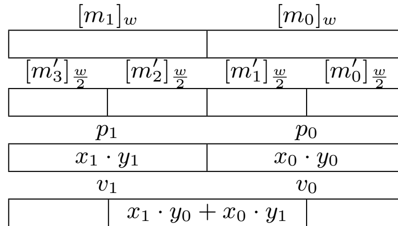

Figure 4.14: Multiplication registers. m_i and m'_j represent w bit memory cells and $\frac{w}{2}$ bit cells respectively.

If there was an overflow in the computation of p_0 we add 1 to p_1. And now p_1 and p_0 contains the product of x and y.

```
void
mp_mul_p(mp_limb_t *p1, mp_limb_t *p0,
        mp_limb_t x, mp_limb_t y)
{
        mp_limb_t x1, x0;
        mp_limb_t y1, y0;
        mp_limb_t t, u, v, v1, v0, c;

        x0 = x & ((1 << w/2) - 1);
        x1 = x >> w/2;

        y0 = y & ((1 << w/2) - 1);
        y1 = y >> w/2;

        t = y0 * x1;
        u = y1 * x0;
        v = t + u;

        v0 = v << w/2;
        v1 = v >> w/2;

        if (v < t || v < u)
                v1 += 1 << w/2;
```

```
        *p0 = y0 * x0 + v0;
        c = p0 < v0 ? 1 : 0;
        *p1 = c + x1 * y1 + v1;
28  }
```

Listing 4.5: Primitive multiplication

Multiple precision multiplication

We have the factors X and Y, where $X = (x_n\ x_{n-1}\ ...x_1\ x_0)b$ and $Y = (y_m\ y_{m-1}\ \ldots y_1\ y_0)_b$. Multiplication consists of two phases; first each digit from one of the factors is multiplied by the other factor, i.e. $y_j \cdot X = P_j$. The second phase sums the intermediate results multiplied by their respective positional factor, to form the final product. That is, we have $P = \sum_{i=0}^{m} P_i b^i$. In our implementation the two phases are interleaved, the separate intermediate products are never formed individually, instead they are summed to the final product during each iteration.

The product P of the $n+1$ digit number X and the $m+1$ digit number Y will have (at most) $n+m+2$ digits. Thus the size of P must be $n+m+2$ w bit words. We want to avoid dynamic allocation of temporary space so we do not require the multiplication to be in-place safe, i.e. we do not allow statements like $X = X \cdot Y$. For convenience and readability we use the array notation instead of the pointer notation, so we write `P[i]` instead of `*(p+i)`.

Now to multiply X with Y to form P we simply iterate over the $m+1$ digits of the factor Y. During each of the $m+1$ iterations we iterate over the $n+1$ digits of the factor X. It is in this inner loop the main work of the multiplication is performed. Each digit from the factor Y is multiplied with all the digits from the factor X and then added, with carry, to the product P. The outer loop advances to the next digit of the factor Y and the primitive multiplications are repeated for this digit. Thus, the final product accumulates in P.

Looking at listing 4.6 we start by detecting if in-place multiplication is mistakenly attempted, if so, an error is generated (line 8). Then we find the lengths of the factors (X and Y) as X_L and Y_L.

Next, we detect special cases. If either X or Y is zero then the product will also be zero. If X_L is zero then there was no significant digits in X and thus X must be zero. At line 18 we check the factors and set P to zero if any of the factors are zero. If X or Y is one then the product

will be Y, or X, respectively. This we detect on lines 24 and 30. Note that the comparison operation will be discussed later, see section 5.1. If $P = 1 \cdot Y$ we set P to Y, on line 26. If $P = X \cdot 1$ we set P to X, on line 32. If no special cases were detected P is initialized to zero prior to the twin loops of the multiplication (line 36). Now the special cases have been handled and P is initialized. We can continue with the main task of multiplying X with Y.

```
void
mp_mul(mp_t P, mp_t X, mp_t Y)
{

5        mp_size_t XL, YL, i, j;
         mp_limb_t c, p1, p0, x;

         if (P == X)
                 MP_ERR(MP_ERR_INPLACE);
10
         XL = mp_len(X);
         YL = mp_len(Y);

         if (*P < (XL + YL)) {
15               MP_ERR(MP_ERR_RESSIZEOPLEN);
         }

         if (XL == 0 || YL == 0) {

20               mp_zero(P);
                 return;
         }

         if (mp_eq(X, mp_One)) {
25
                 mp_set(P, Y);
                 return;
         }

30       if (mp_eq(Y, mp_One)) {

                 mp_set(P, X);
```

```
                return;
        }

        for (i = 1; i <= *P; i++) {
                P[i] = 0;
        }

        for (i = 0; i < YL; i++) {

                c = 0;

                for (j = 0; j < XL; j++) {

                        mp_mul_p(&p1, &p0, X[*X - j],
                                Y[*Y - i]);

                        p0 += c;
                        if (p0 < c)
                                p1++;
                        x = P[*P - i - j];
                        p0 += x;
                        if (p0 < x)
                               p1++;
                        P[*P - i - j] = p0;
                        c = p1;
                }

                P[*P - i - XL] = p1;
        }
}
```

Listing 4.6: Multiplication

We are going to multiply each of the digits of Y, y_i, $0 \leq i < Y_L$ with each of the digits of X, x_j, $0 \leq j < X_L$. The outer loop, lines 36 to 61, computes $P = \sum_{i=0}^{Y_L} P_i b^i$. The inner loop, over j, lines 44 to 58, computes $P_i b^i$ in P.

Looking at the inner loop, we iterate j from 0 to X_L (line 44) and perform the primitive multiplication of digit j of X, i.e. x_j with digit i of Y (y_i). We use our primitive multiplication operation from section

4.6, `mp_mul_p`, to multiply x_j with y_i, the result is returned in p_1 and p_0. We now have $[y_i]_w \cdot [x_j]_w$ in $[p_1 p_0]_w$. We add the carry from the previous iteration, c, to p_0 and handle an eventual overflow, lines 49 to 51.

Continuing, we add our new result, p_0 to the accumulated result in P. First we get P_{i+j} in x (line 52), then we add p_0 and x, checking for overflow, line 53 to 55. If there is an overflow adding the accumulated product digit with the new partial result we increase p_1 by one. Finally we update the result, setting P_{i+j} to the new partial result (line 56) and we also set the new carry to p_1 (line 57). Returning to the beginning of the loop we now see that the carry c that is added to p_0 is the high part of the result of the primitive multiplication, i.e. p_1 from the previous iteration.

After the inner loop is completed we must handle the last remaining digit, i.e. the high part from the final primitive multiplication which is to be added to digit $i + X_L$. Since digit $i + X_L$ will always be zero when we reach line 60 we can simply set P_{i+X_L} to p_1.

The outer loop initializes the carry (c, line 42), executes the inner loop and takes care of the last remaining digit. The product is returned in P.

4.7 Division

Division computes the quotient and remainder of the dividend when divided by the divisor. For two numbers X and Y we have $X = Q \cdot Y + R$.

We examine primitive division in the decimal base ($b = 10$). Let x_i and y_i be two numbers, where $0 \leq x_i < b$ and $0 < y_i < b$. The primitive division operation computes q_i and r_i in $(x_i)_b = (q_i)_b \cdot (y_i)_b + (r_i)_b$.

From figure 4.15 (shaded areas) we see that the quotient is undefined if the divisor is zero ($y_i = 0$) and that the quotient is zero if the dividend is zero, i.e. if $x_i = 0$ then $q_i = 0$. We also see that if the divisor is one the quotient is equal to the divisor ($y_i = 1$ implies $q_i = y_i$). Further, if $y_i > x_i$ then the quotient is zero. More important is that if $y_i \geq \frac{b}{2}$ the quotient is either zero or one, i.e. $q_i \in \{0, 1\}$.

Examining figure 4.16 we see that the remainder is undefined if the divisor is zero ($y_i = 0$). If the dividend is zero then the remainder is zero, i.e. if $x_i = 0$ then $r_i = 0$ and if the divisor is one the remainder is also zero, i.e. if $y_i = 1$ then $r_i = 0$, since all numbers are evenly divisible with one. We also see that if the divisor is equal to the dividend then

$x_i \backslash y_i$	0	1	2	3	4	5	6	7	8	9
0	-	0	0	0	0	0	0	0	0	0
1	-	1	0	0	0	0	0	0	0	0
2	-	2	1	0	0	0	0	0	0	0
3	-	3	1	1	0	0	0	0	0	0
4	-	4	2	1	1	0	0	0	0	0
5	-	5	2	1	1	1	0	0	0	0
6	-	6	3	2	1	1	1	0	0	0
7	-	7	3	2	1	1	1	1	0	0
8	-	8	4	2	2	1	1	1	1	0
9	-	9	4	3	2	1	1	1	1	1

Figure 4.15: Quotient q_i of primitive division $\lfloor x_i/y_i \rfloor = q_i$ in base $b = 10$

.

$x_i \backslash y_i$	0	1	2	3	4	5	6	7	8	9
0	-	0	0	0	0	0	0	0	0	0
1	-	0	1	1	1	1	1	1	1	1
2	-	0	0	2	2	2	2	2	2	2
3	-	0	1	0	3	3	3	3	3	3
4	-	0	0	1	0	4	4	4	4	4
5	-	0	1	2	1	0	5	5	5	5
6	-	0	0	0	2	1	0	6	6	6
7	-	0	1	1	3	2	1	0	7	7
8	-	0	0	2	0	3	2	1	0	8
9	-	0	1	0	1	4	3	2	1	0

Figure 4.16: Remainder r_i of primitive division $x_i - \lfloor x_i/y_i \rfloor = r_i$ in base $b = 10$

.

the remainder is zero, i.e. if $y_i = x_i$ then $r_i = 0$, since any number is divisible by itself. Finally, if the divisor is greater than the dividend the remainder is equal to the dividend, i.e. if $y_i > x_i$ then $r_i = y_i$, since no number is divisible by a number greater than itself.

Long division

Long division is the algorithm we recognize as the standard pen and paper method of dividing a long number (dividend) with another long number (divisor). It consists of tabulating the two numbers (figure 4.17) and then repeatedly computing the quotient digits and a remainder until all the quotient digits and the final remainder are calculated.

$$
\begin{array}{r|llllllll}
 & q_{n-m} & q_{n-m-1} & \cdots & q_1 & q_0 & & & \\
\hline
y_m y_{m-1} \cdots y_1 y_0 & x_{n+1} & x_n & \cdots & x_1 & x_0 & & & \\
 & -z_{m+1} & z_m & \cdots & z_1 & z_0 & & & \\
\cline{2-6}
 & & r_n & r_{n-1} & \cdots & r_{n-m} & x_{n-m-1} & & \\
 & - & z_{m+1} & z_m & \cdots & z_1 & z_0 & & \\
\cline{3-7}
 & & & r_{n-1} & r_{n-2} & \cdots & r_{n-m-1} & x_{n-m-2} & \\
 & & & z_{m+1} & z_m & \cdots & z_1 & z_0 & \\
 & & & & & & & & \\
 & & & & r_{m+1} & r_m & \cdots & r_1 & x_0 \\
 & & & - & z_{m+1} & z_m & \cdots & z_1 & z_0 \\
\cline{4-9}
 & & & & & r_m & \cdots & r_1 & r_0
\end{array}
$$

Figure 4.17: Long division ($X = QY + R$).

We examine division by considering division of an $n+1$ digit number X with an $m+1$ digit number Y. We have

$$X = (x_n x_{n-1} \ldots x_1 x_0)_b = \sum_{i=0}^{n} x_i b^i$$

$$Y = (y_m y_{m-1} \ldots y_1 y_0)_b = \sum_{j=0}^{n} y_j b^j$$

and wish to calculate the $m-n$ digit quotient and the m digit remainder

$$Q = (q_{n-m} q_{n-m-1} \ldots q_1 q_0)_b = \sum_{k=0}^{n-m} q_k b^k$$

$$R = (r_m r_{m-1} \ldots r_1 r_0)_b = \sum_{l=0}^{m} r_l b^l$$

We determine the quotient digits iteratively, by computing each q_k where $n - m \leq k \leq 0$ in

$$R_{k+1} = q_k Y b^k + R_k$$

starting with $k = n - m$ going to 0 and $R_{n-m+1} = X$. Then we have

$$q_k = \left\lfloor \frac{R_{k+1}}{Y b^k} \right\rfloor$$

and

$$R_k = R_{k+1} - q_k Y b^k$$

Determining each quotient digit consists of finding the largest factor (q_k) that when multiplied with Yb^k divides R_{k+1}. The product $q_k Y$ is at least m digits and at most $m + 1$ digits. We (humans) have little problem calculating (or guessing) the quotient digits, however, we are interested in a method to estimate, and then correct the estimate to find the the quotient digits, which require only a few number of primitive operations.

We would like to establish that $\hat{q}$ in

$$(x_n x_{n-1})_b = \hat{q} y_m + \hat{r}$$

is a good estimate of the true quotient digit q in

$$X = qYb^{m-n-1} + r$$

We assume that $x_n < y_m$, this will ensure that the quotient digit q is less than b, i.e. $q \leq b - 1$. We are also assuming that the divisor is *normalized*, i.e. that $y_m \geq \lfloor \frac{b}{2} \rfloor$, see section 4.7.

We know by theorem 4 that $q' \leq q$ if $y' > y$ in $x = qy + r$ and $x = q'y' + r'$ and also, by equation 4.1, that if $X = (x_n\ x_{n-1} \ldots x_1\ x_0)_b$ then $(x_n\ x_{n-1})_b b^{n-1} \leq X$ and by equation 4.2 that $(x_n\ (x_{n-1}{+}1))_b b^{n-1} > X$. Thus if we have $Y = (y_m\ y_{m-1} \ldots y_1\ y_0)_b$ and

$$\begin{aligned} X &= qYb^{n-m-1} + R, \quad 0 \leq R < Yb^{n-m-1} \\ (x_n b + x_{n-1})b^{n-1} &= \hat{q} y_m b^{n-1} + \hat{r}, \quad 0 \leq \hat{r} < y_m \end{aligned}$$

then we see that

$$\begin{aligned} qy_m b^{n-1} &\leq qYb^{n-m-1} \\ &\leq X \\ &< (x_n b + x_{n-1} + 1)b^{n-1} \\ &\leq (x_n b + x_{n-1} + 1)b^{n-1} - 1 \\ &= (x_n b + x_{n-1})b^{n-1} + b^{n-1} - 1 \\ &< (\hat{q} + 1)y_m b^{n-1} + b^{n-1} - 1 \\ &\leq ((\hat{q} + 1)y_m - 1)b^{n-1} + b^{n-1} - 1 \\ &= (\hat{q} + 1)y_m b^{n-1} - 1 \end{aligned}$$

and we have that $qy_m b^{n-1} \leq y_m(\hat{q} + 1)b^{n-1} - 1$ or $qy_m b^{n-1} < y_m(\hat{q} + 1)b^{n-1}$ thus $q < \hat{q} + 1$ and then we can say that

$$\hat{q} \leq q \tag{4.3}$$

Continuing, we see that

$$\begin{aligned} \hat{q}y_m b^{n-1} &\leq (x_n b + x_{n-1})b^{n-1} \\ &\leq X \\ &< (q + 1)Yb^{n-m-1} \\ &\leq (q + 1)Yb^{n-m-1} - 1 \\ &< (q + 1)(y_m + 1)b^{n-1} - 1 \\ &\leq (q + 1)(y_m b^{n-1} + b^{n-1} - 1) - 1 \\ &= (q + 1)(y_m + 1)b^{n-1} - (q + 1) - 1 \\ &= (q + 1)(y_m + 1)b^{n-1} - (q + 2) \\ &< (q + 1)(y_m + 1)b^{n-1} \end{aligned}$$

So we have $qy_m < (q + 1)(y_m + 1)$ which gives $qy_m < y_m(q + 1) + q + 1$ and

$$qy_m \leq (q + 1)y_m + q \tag{4.4}$$

Now, since $x_n < y_m$ we must have that $\hat{q} \leq b - 1$. We now have two possibilities, first if $q \geq b - 1$ then we trivially have

$$\hat{q} \leq q \tag{4.5}$$

Second, if instead $q < b-1$ then, since y_m is normalized, i.e. $y_m \geq \lfloor \frac{b}{2} \rfloor$ or $2y_m \geq b-1$, we have that $q < 2y_m$. So from equation 4.4 above we see

$$\hat{q}y_m \leq (q+1)y_m + q < (q+1)y_m + 2y_m < (q+3)y_m$$

or

$$\hat{q} \leq q+2 \tag{4.6}$$

So we have from equation 4.3 that $\hat{q} \geq q$ and from equation 4.5 that $\hat{q} \leq q$ if $q \geq b-1$ and finally $\hat{q} \leq q+2$ if $q > b-1$ from equation 4.6, we can now conclude that

$$q \leq \hat{q} \leq q+2 \tag{4.7}$$

This means our estimate ($\hat{q}$) is either correct or larger than the true quotient and if it is not correct (larger) then it is at most off by two.

Algorithm 4. Division.

The division algorithm computes the quotient and remainder when a positive integer is divided by another positive integer.

In: Two integers A and B with $n+1$ and $m+1$ base b digits respectively, i.e. $A = (a_n a_{n-1} \ldots a_1 a_0)_b$ and $B = (b_m, b_{m-1} \ldots b_1 b_0)_b$, such that $A \geq 0$, $B \geq 0$, $n \geq m \geq 1$ and $b_m \neq 0$.

Out: The quotient, $Q = (\, q_{n-m} q_{n-m-1} \ldots q_1 q_0 \;,)_b$ and the remainder $R = (\, r_m r_{m-1} \ldots r_1 r_0 \,)_b$ such that $A = QB + R$, where $0 \leq R < B$.

1: **for** $j = 0 \ldots n-m$ **do**
2: $\quad q_j = 0$
3: **end for**
4: **while** $A \geq Bb^{n-m}$ **do**
5: $\quad q_{n-m} = q_{n-m} + 1$
6: $\quad A = A - Bb^{n-m}$
7: **end while**
8: **for** $i = n \ldots m+1$ **do**
9: $\quad$ **if** $a_i = b_m$ **then**
10: $\quad\quad q_{i-m-1} = b-1$
11: $\quad$ **else**
12: $\quad\quad q_{i-m-1} = \lfloor \frac{a_i b + a_{i-1}}{b_m} \rfloor$
13: $\quad$ **end if**
14: $\quad$ **while** $q_{i-m-1}(b_m b + b_{m-1}) \geq a_i b^2 + a_{i-1} b + a_{i-2}$ **do**
15: $\quad\quad q_{i-m-1} = q_{i-m-1} - 1$

16: **end while**
17: $A = A - q_{i-m-1}Bb^{i-m-1}$
18: **if** $A < 0$ **then**
19: $A = A + Bb^{i-m-1}$
20: $q_{i-m-1} = q_{i-m-1} - 1$
21: **end if**
22: **end for**
23: $R = A$
24: **return** (Q, R)

For each iteration, lines 8 to 22, the division algorithm (algorithm 4) executes $1 + (m + 2)$ primitive multiplications. The first one on line 14. Since the divisor is normalized we assume it is extended with one digit, and we have $m + 2$ multiplications on line 17. For each iteration the algorithm executes one primitive division (line 12).

Before termination the algorithm will execute $n - m$ iterations. In total we will have $(n - m)(m + 4)$ primitive operations (multiplications and divisions).

If $k = n - m$ and $l = 2m - n$ we have $(n - m)(m + 4) = k(k + l + 4) = k^2 + kl + 4k$, thus our division algorithm, algorithm 4, belongs to $O(k^2)$. There are better algorithms.

Primitive division

Our primitive division computes q in $x = qy + r$ where x is a two digit base b^k number and y is a single digit base b^k number, i.e. x is stored in two w bit word unsigned integers and y in a one w bit word unsigned integer. We divide x into four base b^{k-1} numbers and y into two b^{k-1} numbers, i.e. x is represented as four $\frac{w}{2}$ bit unsigned integers and y as two $\frac{w}{2}$ bit unsigned integers.

$$x = [x_1 x_0]_w = [x'_3 x'_2 x'_1 x'_0]_{\frac{w}{2}}$$
$$y = [y]_w = [y_1 y_0]_{\frac{w}{2}}$$

The goal is to compute the two w bit words of $q = [q_1 q_0]_w$ and the w bit word of $[r]_w$. This is done by calculating $q = [q_2 q_1 q_0]_{\frac{w}{2}}$ and $r = [r_1 r_0]_{\frac{w}{2}}$ and ending with a change to a w bit representation. That is, we use the base change theorem (theorem 21) and we have that $b^k = B^2 = 2^w$ and $b^{k-1} = B = 2^{\frac{w}{2}}$.

So we want to divide $[x'_3x'_2x'_1x'_0]_{\frac{w}{2}}$ by $[y_1y_0]_{\frac{w}{2}}$ using the schoolbook long division, see an example in figure 4.18. We are going to use some techniques to facilitate the computation. The first trick is not to separate the w bit x_1 and x_0 into four distinct $\frac{w}{2}$ bit words, instead we will let them remain in the original two w bit words and handle the base change on the fly. We do this in r_1 and r_0.

q				0	5	4		$[q_2q_1q_0]_{\frac{w}{2}}$	
y x	6	3	/	3	4	5	6	$[y_1y_0]_{\frac{w}{2}}$	$[x'_3x'_2x'_1x'_0]_{\frac{w}{2}}$
$q_2 \cdot y$			-	0	0				
r_2				3	4	5		r_2	
$q_1 \cdot y$			-	3	1	5			
r_1				0	3	0	6	r_1	
$q_0 \cdot y$				-	2	5	2		
r_0					0	5	4	r_0	

$$
\begin{aligned}
r_2 &= \left([x'_3x'_2]_{\frac{w}{2}} - [q_2]_{\frac{w}{2}} \cdot [y_1y_0]_{\frac{w}{2}}\right)2^{\frac{w}{2}} + [x'_1]_{\frac{w}{2}} \\
r_1 &= \left(r_2 - [q_1]_{\frac{w}{2}} \cdot [y_1y_0]_{\frac{w}{2}}\right)2^{\frac{w}{2}} + [x'_0]_{\frac{w}{2}} \\
r_0 &= r_1 - [q_0]_{\frac{w}{2}}[y_1y_0]_{\frac{w}{2}}
\end{aligned}
$$

Figure 4.18: Long division example.

Next, we will assume that the divisor, y is *normalized*, i.e. that the first digit of the divisor y_m is greater than $\lfloor\frac{b}{2}\rfloor$ or $2y_m + 1 \geq b$. In our case greater than $\frac{2^{\frac{w}{2}}}{2}$. One effect of normalization is that the most significant bit in the binary representation of the digit will be set. If y_m is normalized then the value of $(x)_b/(y_m)_b$ will be either 1 or 0, see section 4.7. We will also require that the first digit of the divisor is greater than the first digit of the dividend, i.e. $[y_1]_{\frac{w}{2}} > [x'_3]_{\frac{w}{2}}$, this will ensure that we will only have to execute two iterations of the long division algorithm because $[q_2]_{\frac{w}{2}}$ will always be zero.

Finally, and possibly most important, computing the exact quotient digit immmediately requires multiple precision division which we do not have so we are going to first compute an estimate of the quotient digit then check it and adjust it if it is incorrect.

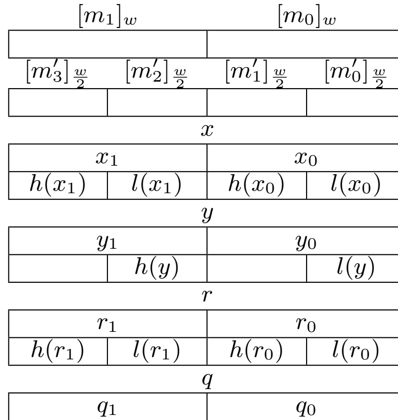

Figure 4.19: Division registers.

Looking at listing 4.7, we start by dividing y into two $\frac{w}{2}$ bit words, y_1 and y_0 (lines 10 and 11). We do not explicitly divide x_1 and x_0, instead we will shift the bits in $[x_1]_w$ and $[x_0]_w$ to align as required. We work with the remainder in $*r$, so we set it to x_1, the two most significant half-words of the dividend (line 13).

Our first attempt at the first quotient digit q_1 is

$$q_1 = \left\lfloor \frac{[x_1]_w}{[y_1]_{\frac{w}{2}}} \right\rfloor = \left\lfloor \frac{[x'_3 x'_2]_{\frac{w}{2}}}{[y_1]_{\frac{w}{2}}} \right\rfloor$$

since $[x_1]_w$ holds $[x'_3\ x'_2]_{\frac{w}{2}}$, see listing 4.7 on line 15. We know from theorem 4 and section 4.7 that our estimate of the quotient digit will be greater than or equal to the true quotient digit. Thus we must check and possibly correct our guess.

In the end the first remainder will have been computed as $[x_1]_w - ([q_1]_{\frac{w}{2}} \cdot [y_1\ y_0]_{\frac{w}{2}}$. However, the computation of the first remainder and the correction of the quotient digit guess is interleaved. First we set the remainder to $*r = [x_1]_w - [q_1]_{\frac{w}{2}} \cdot [y_1]_{\frac{w}{2}}$ and $v = [q_1]_{\frac{w}{2}} \cdot [y_0]_{\frac{w}{2}}$, which will be subtracted later. This will leave the high half of $[*r]_w$ empty. See the computation of r_1 in figure 4.18 for an illustration. Next

(line 18) we shift $*r$ so that the low part becomes the high part and we also shift in the high part of $[x_0]_w$, i.e. $[x'_1]_{\frac{w}{2}}$. $*r$ now contains $\big([x'_3\ x'_2]_{\frac{w}{2}} - [q_1]_{\frac{w}{2}} \cdot [y_1]_{\frac{w}{2}}\big)2^{\frac{w}{2}} + [x'_1]_{\frac{w}{2}}$.

Now we are ready to check and correct our guess. If $*r$ is greater than or equal to v, where $v = [q_1]_{\frac{w}{2}} \cdot [y_0]_{\frac{w}{2}}$, then we have the correct quotient digit and we are done with q_1. If not, that is, if $*r$ is less than v we decrease q_1 by one and add back $[y]_w$ to r_1 (lines 20 to 23). We check again (line 25). If $*r \geq y$ and $*r < v$ then q_1 is still off by one so we repeat the correction, we again decrease q_1 and subtract y from $*r$ (lines 23 to 24). We happen to know that the error in our quotient guess is never larger than two (equation 4.7), so we will never need to correct more than twice.

Finally we subtract v from $*r$ to finish the computation of the first qotient digit. Now $*r$ holds $[x'_3\ x'_2\ x'_1]_{\frac{w}{2}} - [q_1]_{\frac{w}{2}} \cdot [y_1 y_0]_{\frac{w}{2}}$.

We now continue with q_0. The computation of q_0 follows the same pattern as our previous computation of q_1. First, on line 29, q_0 is estimated as

$$q_0 = \lfloor \frac{[r_1]_w}{[y_1]_{\frac{w}{2}}} \rfloor$$

Then we compute $*r$. In the end we will have computed $*r$ as $*r - [q_0]_{\frac{w}{2}} \cdot [y_1\ y_0]_{\frac{w}{2}}$. But again the computation of $*r$ and the correction of q_0 is interleaved. We set the remainder to $*r = [*r]_w - [q_0]_{\frac{w}{2}} \cdot [y_1]_{\frac{w}{2}}$ and $v = [q_0]_{\frac{w}{2}} \cdot [y_0]_{\frac{w}{2}}$ (lines 30 to 31). This leaves the high half of $*r$ empty. We shift in x'_0, see line 32.

If $*r \geq v$ then q_0 is correct and we only have to subtract v from r_0. If not ($r_0 < v$) then we adjust q_0 by subtracting one and adding back y to $*r$ (lines 37 to 38). We check again, if $*r \geq y$ and $r_0 < v$ (line 39) then we need to adjust again, by subtracting one from q_0 and adding back y to $*r$ (lines 38 to 39). The final operation is to subtract v, which we now know we can do, from $*r$ (line 41). Now we have $*r = [r_1]_w - [q_0]_{\frac{w}{2}} \cdot [y_1 y_0]_{\frac{w}{2}}$.

Finally we return the result, quotient in $[q]_w$ as $[q]_w = [q_1 q_0]_{\frac{w}{2}}$ (line 43) and the remainder is already in $[*r]_w$.

```
void
mp_div_p(mp_limb_t *q, mp_limb_t *r,
        mp_limb_t x1, mp_limb_t x0, mp_limb_t y)
{
5       mp_limb_t y1, y0;
```

```
        mp_limb_t r1, r0;
        mp_limb_t q1, q0;
        mp_limb_t v;

10      y1 = y >> w/2;
        y0 = y & ((1 << w/2) - 1);

        *r = x1;

15      q1 = *r / y1;
        v = q1 * y0;
        *r -= q1 * y1;
        *r = (*r << w/2) | (x0 >> w/2);

20      if (*r < v) {

                do {
                        q1 -= 1;
                        *r += y;
25              } while (*r >= y && *r < v);
        }
        *r -= v;

        q0 = *r / y1;
30      v = q0 * y0;
        *r -= q0 * y1;
        *r = (*r << w/2) | (x0 & ((1 << w/2) - 1));

        if (r0 < v) {
35
                do {
                        q0 -= 1;
                        *r += y;
                } while (*r >= y && *r < v);
40      }
        r0 -= v;

        *q = (q1 << w/2) | q0;
```

```
}
```

Listing 4.7: Primitive division

Multiple precision division

Division is the most complex of the four elementary arithmetic operations. It consists of repeatedly calculating the quotient digits, subtracting the intermediate products, $q_i \cdot Y$, from the current position, thus iteratively forming the quotient Q and the remainder R.

To directly compute the correct current quotient digit we need multiple precision division. But we are currently implementing multiple precision division so we will have to find an alternative technique to determine the correct quotient digits. Our strategy is to estimate the quotient digit, check the estimate and correct it if it is incorrect.

To simplify our task we divide our implementation into a main division routine, `mp_div` which makes use of the four subroutines `mp_norm`, `mp_div_mulsubn`, `mp_div_addn` and the primitive division `mp_mul_p`.

`mp_norm` normalizes the provided operand and computes the shift offset. Normalization consists of shifting the operand so that it is larger than $\lfloor \frac{b}{2} \rfloor$. In this context normalization implies shifting the operand so that it's highest bit is set. As usual we have $X = (x_n\ x_{n-1} \ldots x_1\ x_0)_b$ and $Y = (y_m\ y_{m-1} \ldots y_1\ y_0)_b$. `mp_div_mulsubn` computes the product of a single digit and a multiple precision number and subtracts the product from a minuend, starting at position i going to $i + m$. In other words, `mp_div_mulsubn` computes $X - q \cdot Y \cdot b^i$. `mp_div_addn` computes $X + Y \cdot b^i$, i.e. it adds Y to X starting at position i going to $i + m$. We also make use of the primitive multiplication routine, `mp_mul_p`.

Normalization

To normalize X we multiply X by a constant so that $X \geq \lfloor \frac{b}{2} \rfloor$. Our base is $b = 2^w$ and thus, a normalized divisor is larger than or equal to 2^{w-1}. This implies that the highest bit of the divisor must be set.

Our computer works in the base 2 and we have primitive bit manipulation operations, such as shift, at our disposal. An efficient strategy to normalize the operand is to multiply it with 2^s, i.e. to shift it left s times so that the highest bit of the most significant word in the operand is set.

```
void
mp_norm(mp_limb_t *s, mp_t R, mp_t X)
{
        mp_size_t i, XL;
        mp_limb_t x;

        XL = mp_len(X);
        x = *(X + 1 + *X - XL);

        for (i = 0; x; i++)
                a >>= 1;
        *s = w - i;
        (void)mp_sl(R, X, *s);
}
```

Listing 4.8: Normalization

First we find the length of the operand, X_L (line 7), then we fetch the highest digit of X (line 8). The least significant digit (x_0) of X is in $X + *X$ (see section 4.3). We want to find x_n. The length of X is X_L, so $X + *X - X_L$ points to x_{n+1} and $X + *X - X_L + 1$ thus points to x_n.

We then shift x *right* until x is zero, lines 10 and 11. So in i we have the position of the highest set bit of the most significant digit of X. Then $w - i$ is the length of the zero prefix of the most significant digit of X which we store in s at line 12.

Finally, at line 13, we shift X *left* s bits and return the normalized operand in R. Note that s is also provided to the caller. The normalization routine makes us of `mp_sl`, the left shift routine. Our implementation of left shift is in place safe and so is normalization, thus we can write `mp_norm(&s, X, X)` for $X = \text{norm}(X)$.

Addition at digit position n

In our calculation of the quotient digits we first compute an estimate of the current quotient digit, check it and correct it until it the estimate is correct. We check the quotient digit by subtracting the product $\hat{q}Yb^i$ from the current remainder. If there is an underflow in the subtraction then clearly $\hat{q}$ was too large. We adjust $\hat{q}$ by decreasing it with one ,then we must also add back Yb^i to the current remainder. Adding back Yb^i is the task performed by `mp_div_addn`.

`mp_div_addn` is a specialized implementation of addition and follows the same pattern as the general addition described in section 4.4. However, multiplication by b^i will have the effect of adding digits of Y starting at digit position i of X, see figure 4.20.

$$\begin{array}{ccccccccc} & r_{i+m} & r_{i+m-1} & \cdots & r_{i+1} & r_i & \cdots & r_1 & r_0 \\ + & y_m & y_{m-1} & \cdots & y_1 & y_0 & & & \\ \hline \end{array}$$

Figure 4.20: Addn.

Looking at listing 4.9 we start by finding the lengths of X and Y in X_L and Y_L. We check that the inputs are valid, i.e. that the length of X and Y is at least 2 (lines 11 and 14), that the size of X is at least the size of Y (line 17) and that the size of R is equal to the size of X (line 20). Then the pointers r, x and y are initialized to the least significant digit of R, X and Y respectively (lines 23 to 25).

```
mp_limb_t
mp_div_addn(mp_t R, mp_t X, mp_t Y, mp_size_t n) {

        mp_size_t XL, YL;
        mp_size_t i;
        mp_limb_t *r, *x, *y, c, t;

        XL = mp_len(X);
        YL = mp_len(Y);

        if (YL < 2)
                MP_ERR(MP_ERR_OPSIZE);

        if (XL < 2)
                MP_ERR(MP_ERR_OPSIZE);

        if (*X < YL)
                MP_ERR(MP_ERR_OPSIZE);

        if (*X != *R)
                MP_ERR(MP_ERR_RESSIZEOPLEN);

        r = R + *R;
```

```
        x = X + *X;
25      y = Y + *Y;

        for (i = 0; i < n; i++) {

                *r = *x;
30              r--;
                x--;
        }

        c = 0;
35
        for (; i < YL; i++) {

                t = *x + c;
                c = (t < *x) || (t < c) ? 1 : 0;
40              t += y;
                c += (t < *y) ? 1 : 0;

                *r = t;
                r--;
45              x--;
                y--;
        }

        return c;
50 }
```

Listing 4.9: Addition, digit position n

Continuing, the low n digits of X will not be changed by the addition so they are simply copied to R, lines 27 to 31. Also the initial carry is set to zero, on line 34. We are now ready to proceed with the main task of adding Y to X starting at digit n.

We iterate from n to Y_L, i.e. we will process digit n to digit $n+m$ (line 36).

The result to compute is $x_i + y_i + c_{i-1}$ and the new carry c_i. We compute the result in a temporary variable t, by adding x_i and the previous carry on line 38 and then setting the new carry if an overflow occured (line 39). Then we add y_i to t (line 40) and once more check

and set the carry if an overflow occured on line 41. We set the result, $*r$, to t (line 43).

Finally the pointers are decreased to refer to the next digits, lines 44 to 46 and this iteration is complete.

When all digits have been processed we return c, so that an overflow in the addition is indicated to the caller.

Multiply and subtract

Our calculation of the quotient digits in division first estimates the quotient digit, then we check if it is correct, if not we repeatedly correct it until it is good. To check the quotient digit for correctness we must evaluate if the product of the quotient digit and the divisor is larger than the current remainder *at the current position.*

We have the dividend $X = (x_n\ x_{m-1} \ldots x_1\ x_0)_b$, the remainder $R = (r_k\ r_{k-1} \ldots r_1\ r_0)_b$ where $0 \leq k \leq n$ and the divisor $Y = (y_m\ y_{m-1} \ldots y_1 y_0)_b$. We need to calculate $R - q_i Y b^i$, where $i = k - m$.

Multiplying Y by b^i implies that the n lowest digits of X will remain the same after the operation. We start by setting the n lowest digits of R to the n lowest digits of X. Then we execute the main loop of the `mp_div_mulsubn` routine where we make use of the `mp_mul_p` routine to multiply and subtract $\hat{q}Yb^i$ from R. Finally we handle the last digit and return the last carry. That is, if $\hat{q}Yb^i > X$ we will return an overflow.

Looking at listing 4.10, at line 10 we get the lengths of X and Y in X_L and Y_L respectively. Then we check so that X is longer than two digits (line 13) and that the result of the operation will fit in R (line 16).

On lines 19 to 21 the pointers x, y and r are initialized to the least significant digit of x, y and r. Next the low digits of R, which will not be touched by the operation, are copied from X to R and p_0 and c_0 are initialized (lines 30 to 31).

We are now ready to enter the main loop where the computation of $X - \hat{q}Yb^i$ is performed. We iterate over the length of Y, i.e. from 0 to Y_L (line 33).

First we compute the product of $\hat{q}$ and the current divisor digit (y) to p_2 and t, high and low part respectively (line 35). Initially p_0 is zero, but on subsequent iterations p_0 will hold the higher part of the result from the previous round, i.e. the carry from the multiplication in the previous round, see figure 4.21. p_0 and t are added (line 37) and we check for overflow (line 38). If there is an overflow p_1 is increased with one (line 39). We now continue to subtract our result from the dividend

i					p_1	p_0	
0:					$h(\hat{q}y_0)$	$l(\hat{q}y_0)$	
1:				$l(\hat{q}y_1)$	$l(\hat{q}y_1)$		
$\vdots$			$\cdots$				
m:	$h(\hat{q}y_m)$	$l(\hat{q}y_m)$					
	r_{i+m+1}	r_{i+m}	$\cdots$	$\cdots$	r_{i+1}	r_i	$\cdots$

Figure 4.21: Mulsubn.

(X). First, we check for underflow (line 40), which we save in c_1. Then we subtract p_0 from x and also the carry, c_0 from the previous round (borrow), on lines 41 and 43. We also need to check for underflow once more, line 42.

```
mp_limb_t
mp_div_mulsubn(mp_t R, mp_t X, mp_t Y,
       mp_limb_t q, mp_size_t n) {

       mp_size_t RL, XL, YL, i;

       mp_limb_t p1, p0, t, c1, c0;
       mp_limb_t *x, *y, *r;

       YL = mp_len(Y);
       XL = mp_len(X);

       if (XL < 2)
              MP_ERR(MP_ERR_OPSIZE);

       if (*R <= YL)
              MP_ERR(MP_ERR_RESSIZEOPLEN);

       y = Y + *Y;
       x = X + *A;
       r = R + *R;

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

```
25              *r = *x;
                r--;
                x--;
        }

30      p0 = 0;
        c0 = 0;

        for (i = 0; i < YL; i++) {

35              mp_mul_p(&p1, &t, *y, q);

                p0 += t;
                if (p0 < t)
                        p1++;
40              c1 = (*x < p0) ? 1 : 0;
                *r = *x - p0;
                c1 += (*x < c0) ? 1 : 0;
                *r -= c0;
                p0 = p1;
45              c0 = c1;
                y--;
                x--;
                r--;
        }
50
        c1 = (*x < p1) ? 1 : 0;
        *r = *x - p1;
        c1 += (*x < c0) ? 1 : 0;
        *r -= c0;
55
        return c1;
}
```

Listing 4.10: Multiply and subtract, digit position n

Finally we update p_0 and c_0 for the next round on lines 44 and 45, and decrease the pointers so that they point to the next digits to be processed (lines 46 to 48).

To finish the computation we need to handle the last digit. We check for underflow and subtract p_1 from x and also subtract the previous

carry c_0 from x (lines 51 and 52 to 54). Note that we must check for underflow when subtracting c_0, on line 53. If there an underflow occured when subtracting either the last term or the last carry we return it. Also note that `mp_div_mulsubn` is in place safe, that is we can write

```
mp_div_mulsubn(X, X, Y, q, i)
```

and we will get the expected result in X.

Division routine

The division routine consists of two main parts. First, initialization and handling of special and error cases and then the main loop where the quotient digits are calculated. The initialization and handling of special and error cases is described in the next section. The main computational loop is described in the following section.

Initialization

The first part of the division routine, lines 10 to 89, consists of initialization and handling of special cases.

First we find the lengths of the dividend X in X_L and of the divisor Y in Y_L (lines 10 and 11). Then we check the special cases.

If the length of the dividend is zero then all digits of the divisor is zero and thus the divisor is zero. Division by zero is undefined and results in an error (line 13).

If the length of the dividend is one we check for division by one, by two or for single digit division (divisor and dividend are single digit numbers). If the divisor is one (line 20) we simply set the remainder to zero and the quotient to the dividend (lines 22 and 23) and return.

If the divisor is two the remainder is zero if the dividend is even and one if the dividend is odd. We use the `mp_even` function and simply set the remainder accordingly on lines 29 to 32. We shift the dividend left one bit, set the quotient to the result (line 33) and return.

If both the divisor and the dividend are single digit numbers (line 37) we use the native division and remainder operators to compute the result, lines 42 and 44 respectively.

If the length of the dividend (X) is less than the length of the divisor (Y) then the dividend is less than the divisor and the quotient is zero. We set the quotient to zero (line 53) and the remainder to the dividend (line 54).

If the dividend and the divisor are of the same length (line 58) we check if the divisor is less than the dividend, line 60. If it is then, just like above, we set the quotient to zero (line 62), the remainder to the dividend (line 63) and return. If the dividend and the divisor are equal, line 67, the quotient is one (line 69) and the remainder is zero (line 70).

We continue checking two error cases. The size of the quotient must be at least the size of the difference between the size of the dividend and the size of the divisor. That is, if we have $X = (x_n\ x_{n-1} \ldots x_1\ x_0)_b$ and $Y = (y_m\ y_{m-1} \ldots y_1\ y_0)_b$ then the qotient Q in $X = Q \cdot Y + R$ has at most $n - m$ digits and the remainder, R, has at most $m + 1$ digits, where X have $n + 1$ digits and Y have $m + 1$ digits.

So, if the quotient has less than $X_L - Y_L$ digits, line 75, we generate an error. Also, if the remainder is smaller than the length of the divisor (Y_L, line 78) an error is generated.

We are now done with the special cases and the error cases and can continue with the initialization. First the divisor (Y) is normalized, line 81. The number of bits shifted is returned in s. Normalization of the divisor implies that it is multiplied by a constant to make it larger than or equal to $\lfloor \frac{b}{2} \rfloor$, in our case the divisor is multiplied by 2^s. We must also multiply the dividend with the same amount (shift left s bits). We will use R to iteratively compute the remainder. So we shift X, the dividend, left s bits and store the result in R. Thus R is now initialized (line 82). We also set R_L to the length of R (X, line 83) and the quotient to zero (line 84).

For convenience we store the two most significant digits of the divisor in $y1$ and $y0$ (line 85 and 86). These will be used later in estimation of quotient digits. If the divisor is a single digit number we set $y0$ to zero (line 86). We also initialize r and q to refer to (point to) what will be the most significant digits of the remainder and quotient (lines 87 and 88).

Finally, v is initialized to zero (line 89). v is used to hold the most significant digit of the remainder after trial division. v is essentially a reverse carry.

We are now done with the preamble and can go on with the task of dividing.

```
void
mp_div(mp_t Q, mp_t R, mp_t X, mp_t Y)
{
```

```
        mp_size_t XL, YL, RL, i, s;

        mp_limb_t *x, *r, *q;
        mp_limb_t y1, y0, u, v, p2, p1, p0, pt;

        XL = mp_len(X);
        YL = mp_len(Y);

        if (YL == 0) {

                MP_ERR(MP_ERR_DIVZ);
        }

        if (YL == 1) {

                if (*(Y + *Y) == 1) {

                        mp_set(Q, X);
                        mp_zero(R);
                        return;
                }

                if (*(Y + *Y) == 2) {

                        if (mp_even(X))
                                mp_zero(R);
                        else
                                mp_one(R);
                        (void)mp_sr(Q, X, 1);
                        return;
                }

                if (XL == 1) {

                        mp_zero(Q);
                        mp_zero(R);

                        *(Q + *Q) = *(X + *X) /
                                *(Y + *Y);
                        *(R + *R) = *(X + *X) %
```

```
45                              *(Y + *Y);
                        return;
                }

        }
50
        if (XL < YL) {

                mp_zero(Q);
                mp_set(R, X);
55              return;
        }

        if (XL == YL) {

60              if (mp_lt(X, Y)) {

                        mp_zero(Q);
                        mp_set(R, X);
                        return;
65              }

                if (mp_eq(X, Y)) {

                        mp_one(Q);
70                      mp_zero(R);
                        return;
                }
        }

75      if (*Q <= (XL - YL))
                MP_ERR(MP_ERR_RESSIZE);

        if (*R < YL)
                MP_ERR(MP_ERR_RESSIZE);
80
        mp_norm(&s, Y, Y);
        (void)mp_sl(R, X, s);
        RL = mp_len(R);
        mp_zero(Q);
```

```
85          y1 = *(Y + 1 + *Y - YL);
            y0 = (YL > 1) ? *(Y + 2 + *Y - YL) : 0;
            r = R + 1 + *R - RL;
            q = Q + *Q - (RL - YL);
            v = 0;
90
            for (i = 0; i <= (RL - YL); i++) {

                    mp_div_p(q, &u, v, *r, y);
            adjust:
95                  mp_mul_p(&pt, &p0, y0, *q);
                    mp_mul_p(&p2, &p1, y1, *q);

                    p1 += pt;
                    if (p1 < pt)
100                         p2++;
                    if ((p2 > v) || ((p2 == v) &&
                            (p1 > *r)) || ((p2 == v) &&
                            (p1 == *r) &&
                            (p0 > *(r + 1)))) {
105
                            *q -= 1;
                            goto adjust;
                    }

110                 if (mp_div_mulsubn(R, R, Y,
                            *q, (RL - YL) - i)) {

                                    *q -= 1;
                            }
115
                            (void)mp_div_addn(R, R, Y,
                                    (RL - YL) - i);
                    }

120                 v = *r;
                    r++;
                    q++;
            }
```

```
125         (void)mp_sr(Y, Y, s);
            (void)mp_sr(R, R, s);
    }
```

Listing 4.11: Division

Main loop

The main loop of the division routine first estimates the current quotient digit using `mp_div_p` (line 93) to compute $\hat{q}_i$ in

$$\hat{q}_i = \frac{[r_{n-i}\ r_{n-i-1}]_w}{[y_m]_w} = \frac{r_{n-i}b + r_{n-i-1}}{y_m}$$

We then check our guess $\hat{q}_i$. We know from equation 4.7 that the estimate $\hat{q}_i$ above is at most off by two and if it is incorrect it is larger than the true quotient digit. Thus we will have to correct our quotient digit guess at most two times. To detect an incorrect estimate we evaluate $\hat{q}_i \cdot (y_m b + y_{m-1}) > (r_{n-i}b^2 + r_{n-i-1}b + r_{n-i-2})$, if the equality holds we know that $\hat{q}_i$ is too large.

We compute $\hat{q}_i \cdot (y_m b + y_{m-1})$ by multiplying q and y_m on line 95 and by multiplying q and y_{m-1} on line 96. We then continue, adding the middle halves together, line 98, if an overflow occured we add the carry to p_2, lines 99 and 100. We now have $\hat{q}_i \cdot (y_m b + y_{m-1})$ in $[p_2 p_1 p_0]_w$, see figure 4.22.

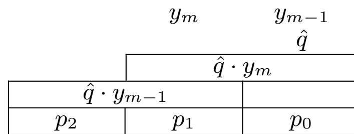

Figure 4.22: Evaluating estimated quotient.

Then we compare $P = \hat{q}(y_m b + y_{m-1})$ to the three most significant digits of the current remainder, $R = r_{n-i}b^2 + r_{n-i-1}b + r_{n-i-2}$.

Say that we have two equal length numbers in the base b, i.e. $X = (x_n\ x_{n-1} \ldots x_1\ x_0)_b$ and $Y = (y_n\ y_{n-1} \ldots y_1\ y_0)_b$ then we know, by equation 4.1 that if $x_n > y_n$ then $X > Y$ and if $x_n < y_n$ then $X < Y$. If

$$\begin{array}{rccc} \hat{q}(y_m b + y_{m-1}): & p_2 & p_1 & p_0 \\ r_{n-i}b^2 + r_{n-i-1}b + r_{n-i-2}: & v & *r & *(r+1) \end{array}$$

Figure 4.23: First comparison.

$x_n = y_n$ then we iteratively continue evaluating $x_{n-i} > y_{n-i}$, $0 < i \leq n$ to decide if $X > Y$.

If we have two base b numbers, $U = (u_2 u_1 u_0)_b$ and $V = (v_2 v_1 v_0)_b$ we can decide if $U > V$ by first comparing u_2 and v_2. If $u_2 > v_2$ then $U > V$ and if $u_2 < v_2$ then $U \ngtr V$. If $u_2 = v_2$ then we compare u_1 and v_1 similarly, and finally we may have to compare u_0 and v_0. Thus we get

$$U > V \quad \text{if} \quad \begin{array}{l} u_2 > v_2 \text{ or} \\ u_2 = v_2 \text{ and } u_1 > v_1 \text{ or} \\ u_2 = v_2 \text{ and } u_1 = v_1 \text{ and } u_0 > v_0 \end{array} \tag{4.8}$$

We compare $p_2 b^2 + p_1 b + p_0$ to $r_{n-i}b^2 + r_{n-i-1}b + r_{n-i-2}$ by comparing each of the digits of the two operands as in equation 4.8, line 101. If we find that $(p_2\ p_1\ p_0)_b$ is greater than $(r_{n-i}\ r_{n-i-1}\ r_{n-i-2})_n$ then $\hat{q}$ must be too large and we adjust our estimate by decreasing it with one (line 106). We repeat the comparison and adjustment until $(p_2\ p_1\ p_0)_b$ no longer is larger than $(r_{n-i}\ r_{n-i-1}\ r_{n-i-2})_b$ and we have a good first guess for the current quotient digit.

We are now ready for a final check using all the digits of the divisor and the current remainder (line 106). We use the `mp_div_mulsubn` routine to compute $R - \hat{q}Yb^i$. If $\hat{q}Yb^i > R$ then an underflow will be indicated when `mp_div_mulsubn` returns a nonzero value and we adjust $\hat{q}$ again (line 113). If there was an underflow we must also add back Yb^i to R, using `mp_div_addn` (line 116). If $\hat{q}Yb^i \ngtr R$ then we already have the correct quotient digit and we are basically done with this iteration. What remains is to update v with the new highest remainder digit $*r$, line 120, and to increase the pointers r and q (lines 125 and 126).

The main loop is repeated $n - m$ times to form the $n - m$ quotient digits of Q in $X = QY + R$. When all iterations are complete the one thing that remains is to undo our normalization, which implies dividing Y and R with 2^s. I.e. we shift Y and R left s bits, lines 125 and 126.

Chapter 5

Other operations

5.1 Comparison

We often need to compare multiple precision numbers. We extend the primitive comparison operators $<$, $\leq$, $>$, $\geq$, $=$ and $\neq$ so we can test the relation between multiple precision numbers.

From equation 4.1 and equation 4.2 we know that, if we have $X = [x_n\ x_{n-1} \ldots x_1\ x_0]_w$ then $[x_n\ x_{n-1} \ldots x_1\ 0]_w \leq X$. Extending the result from the equations 4.1 and 4.2 we see, that if we have the two multiple precision integers X and Y, as $X = [x_n\ x_{n-1} \ldots x_1\ x_0]_w$ and $Y = [y_n\ y_{n-1} \ldots y_1\ y_0]_w$ where $x_i = y_i$ for $i = n \ldots k$, then if $x_{k-1} \leq y_{k-1}$ we have that $X \leq Y$. This reasoning applies to the four comparison operators $<$, $\leq$, $>$, $\geq$, see table 5.1.

$x_{k-1} < y_{k-1}$	$X < Y$
$x_{k-1} \leq y_{k-1}$	$X \leq Y$
$x_{k-1} > y_{k-1}$	$X > Y$
$x_{k-1} \geq y_{k-1}$	$X \geq Y$

Table 5.1: Multiple precision comparison.

Further, if $x_i = y_i$ where $i = n \ldots 0$, i.e. for all i, then $X = Y$. Also if for any i we have that $x_i \neq y_i$ then $X \neq Y$.

The reasoning above applies for all equal length multiple precision numbers. We also wish to be able to compare multiple precision numbers of different sizes and lengths. It is easy to see that if X is longer than Y then X must be greater than Y.

Looking at listing 5.1 we start by finding the lengths of X and Y in X_L and Y_L (lines 9 and 10). If X is longer, line 12, then $X > Y$ and we return 1 (line 14). Else, if X is shorter than Y (line 16) then $X < Y$ and we return -1, line 18.

```
int
mp_cmp(mp_t X, mp_t Y)
{

        mp_size_t i, j;
        mp_size_t XL, YL;
        mp_limb_t *x, *y;

        XL = mp_len(X);
        YL = mp_len(Y);

        if (XL > YL) {

                return 1;

        } else if (XL < YL) {

                return -1;

        } else if (XL == YL) {

                x = X + *X - XL + 1;
                y = Y + *Y - YL + 1;

                for (i = 0; i < XL; i++) {

                        if (*x != *y) {

                                if (*x > *y) {

                                        return 1;
```

```
                        } else {

                                return -1;
35                      }
                }
                x++;
                y++;
        }
40
        return 0;
    }
}
```

Listing 5.1: Comparison

If X and Y are of equal length (line 20) we must do more work. First, we set up pointers to the most significant digits of X and Y respectively on lines 22 and 23. Then we iterate over all digits of X and Y (line 25). If any digits are not equal, line 27, we must have that either $X > Y$ or $X < Y$. We check if the current digit of X is greater than the current digit of Y (line 29. If so then $X > Y$ and we return 1 (line 31). Else $X < Y$ and we return -1 on line 34. If the digits are equal we step the pointers on lines 37 and 38 and go on to the next digit.

If all of the digits of X and Y were equal then X and Y are equal and we return 0 on line 41.

mp_eq:	mp_cmp(X, Y) == 0
mp_neq:	mp_cmp(X, Y) != 0
mp_gt:	mp_cmp(X, Y) == 1
mp_gte:	mp_cmp(X, Y) >= 0
mp_lt:	mp_cmp(X, Y) == -1
mp_lte	mp_cmp(X, Y) <= 0
mp_eqz:	mp_cmp(X, mp_Zero) == 0
mp_neqz:	mp_cmp(X, mp_Zero) != 0

Table 5.2: Comparison and equality operators.

Using the comparison routine described above (in listing 5.1) it is simple to implement the comparison and equality operators, see table 5.2.

5.2 Even and odd

The mathematical approach to determining if a number is even or odd is to compute its remainder modulo 2, i.e. if $N \bmod 2$ is 0 then N is even, if the result is 1 then N is odd. Computing the modulo operation on a multiple precision number is not the most efficient way to determine if it is odd or even. It is more efficient to test if the least significant bit is set.

To determine if a multiple precision number is even or odd we compute the bitwise `AND` of the least significant digit and 1. If the result is one then the number is odd, see listing 5.2.

```
mp_limb_t
mp_odd(mp_t X)
{
        return (X[*X] & 0x1);
}
```

Listing 5.2: Odd

A number which is not odd is even. Also, the number zero is neither odd nor even. So, looking at listing 5.3, we first check if the number is nonzero (line 4).

```
mp_limb_t
mp_even(mp_t X)
{
        if (mp_neqz(X)) {
                return 0 == mp_odd(X);
        } else {
                return 0;
        }
}
```

Listing 5.3: Even

If it is zero, then it can not be even and we return 0 (line 7). If it is nonzero we check if it is odd and return 1 if it is not, on line 5.

5.3 Length

We often need to find the length of a multiple precision number. We have already defined the length as the number digits counted from the

least significant digit, to the last nonzero digit (in section 4.3). We also defined the size as the number of positions in the array representing the number (excluding the cell where the size is stored).

Now, to find the length of a multiple precision number, when we know the size, we start from the most significant position and count the positions until we find the first nonzero digit. Subtracting the number of empty digits from the size gives the length.

Looking at listing 5.4 it all happens on line 7, we start by initializing i to zero and the pointer d to point to the first possibly empty position of the number N.

```
1  mp_size_t
   mp_len(mp_t N)
   {
           mp_size_t  i;
           mp_limb_t *d;
6
           for (i = 0, d = N + 1;
                   *d == 0 && i < *N; d++, i++)
                   ;;

11         return (*N - i);
   }
```

Listing 5.4: Length

Then, while i is less than the size of N, we count the empty positions until we reach the first nonzero digit. That is, we increase d and i while d refers to a zero and while i is less than the size of N. When we exit the `for` loop we have the number of zero positions in i. Finally we return the size of N minus the number of zero positions on line 11.

5.4 Shift

Mathematically, shifting a number left or right, is equivalent to multiplying or dividing by 2^s where s is the shift. Computers normally have efficient native instructions performing shifts to the right and to the left. So, when multiplying or dividing a number by a power of two we want to make use of the more efficient shift operations. Note that we wish the shift operations to be in place safe, so that `mp_sl(X, X, s)` and `mp_sr(X, X, s)` produces the expected result.

Looking at listing 5.5, an implementation of multiple precision left shift, we start by finding the length of X (line 9). We also check for two special cases. First, if the length of X is zero (line 11), then there is nothing to shift and we return doing nothing, line 12. Second, if the requested number of shifts is zero (line 15 then we set R to X and return zero, line 17 and 19. Note that, if the destination and the operand are referring to the same memory cells, we conclude that there is no need to actually copy X to R (line 16).

Before entering the main computational loop of the left shift routine we set up pointers to the least significant digits of X and R (line 22 and 23) find least of the length of X and the size of R in k, line 24, and finally, set the carry c to zero (line 25).

```
mp_limb_t
mp_sl(mp_t R, mp_t X, mp_size_t s)
{
        mp_size_t XL;
        mp_size_t i, k;                                    /* 5 */
        mp_limb_t c, d;
        mp_limb_t *x, *r;

        XL = mp_len(X);
                                                           /* 10 */
        if (XL < 1) {
                return 0;
        }

        if (s == 0) {                                      /* 15 */
                if (R != X) {
                        mp_set(R, X);
                }
                return 0;
        }                                                  /* 20 */

        r = R + *R;
        x = X + *X;
        k = *R < XL ? *R : XL;
        c = 0;                                             /* 25 */

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

```
                d = *x;
                *r = (d << s) | c;
30              c = d >> (8 * sizeof (mp_limb_t) - s);

                r--;
                x--;
        }
35
        if (*R > k) {
                *r = c;
                for (i = k + 1; i < *R; i++) {
                        *(--r) = 0;
40              }
                return 0;
        } else {
                return c;
        }
45  }
```

Listing 5.5: Shift left

Now we enter the main loop, from line 27 to 33. We iterate over the length of the shortest operand, i.e. k times. First we load d with the current digit from X, line 28. Then we shift d s bits to the left and include the bits left over from the previous shift in c and assign the result to the current digit of the result (line 29). On line 30 we extract the bits that we leave for the next iteration, to be included in the next digit of R. Finally we step the pointers r and x on lines 32 and 33 and continue with the next iteration.

When all digits of the shortest operand (X or R) have been processed we check if the size of R is greater than k, i.e. if R is longer than X (line 36). If so then we include the leftover bits from the last iteration of the main loop in the next position of R on line 37 and zero out the rest of R, lines 38 to 39, and return zero (line 41).

If R is not longer than X we simply return the leftover bits from the last shift as overflow (line 43).

The right shift routine follows the same pattern as the left shift discussed above. However, because of the requirement that it be in place safe, it processes the digits in the opposite order from the left shift routine. The initial portion of the routine, lines 9 to 19 are identical. On lines 22 and 23 pointers to the most significant digits of R and X are

initialized. Then, on line 24 the minimum of the size of R and the length of X are determined, in k.

Then the high, unused, part of R is set to zero, lines 26 to 27, the carry c is zeroed on line 27 and we are ready to enter the main loop.

```
mp_limb_t
mp_sr(mp_t R, mp_t X, mp_size_t s)
{
        mp_size_t XL;
        mp_size_t i, k;
        mp_limb_t c, d;
        mp_limb_t *x, *r;

        XL = mp_len(X);

        if (XL < 1) {
                return 0;
        }

        if (s == 0) {
                if (R != X) {
                        mp_set(R, X);
                }
                return 0;
        }

        r = R + 1;
        x = X + 1 + *X - XL;
        k = *R < XL ? *R : XL;

        for (i = 0; i < (*R - k); i++) {
                *r++ = 0;
        }

        c = 0;

        for (i = 0; i < k; i++) {

                d = *x;
                *r = c | (d >> s);
```

```
                c = d << (8 * sizeof (mp_limb_t) - s);

                r++;
                x++;
40        }

        return c;
}
```

Listing 5.6: Shift right

The main loop, lines 34 to 39, shifts the digits of the argument X right s bits. First d is loaded with the current digit from X. Then d is shifted right s bits and the overflow bits from the previous iteration in c are included before assigning the result to the current digit of r, line 36. Next the remaining bits from the current digit are stored in c (line 36). Finally the pointers are increased (lines 38 and 39) and we are ready for the next iteration.

When all digits of X have been processed we return the overflow bits from the last (rightmost) digit.

5.5 Bit operations

We also present routines for getting, setting and clearing bits. We also provide a routine for counting bits, from the least significant up to the most significant set bit.

The three routines for getting, setting and clearing bits are all similar. The arguments are a multiple precision number N and the bit to get, set or clear (n). The three routines consider the least signficant bit of a multiple precision number to be bit zero. I.e. `mp_getbit(N, 0)` gets the value of the least significant bit in N.

Looking at listing 5.7, we start by computing the digit which the bit n is in, line 8. That is, counting from the least significant digit, the bit n must be in digit $\lfloor \frac{n}{w} \rfloor$ where w is the bitwidth of a digit, i.e. the base is 2^w. We also find the index of the bit in the digit, i.e. we compute $n \bmod w$ (line 9). Thus the bit n is bit $l \cdot w + b$, where l is the digit position, counted from zero, and b is the index of the bit, also counted from zero.

We check so that N is indeed at least l digits long, if not we generate an error (line 11). Then we get the digit l from N (line 14) and return

bit b of r. That is, we shift a one b steps to the left and compute the bitwise and of r and the shifted one (line 15).

```
mp_limb_t
mp_getbit(mp_t N, mp_size_t n)
{
        mp_size_t l;
        mp_size_t b;
        mp_limb_t r;

        l = n / w;
        b = n % w;

        if (*N <= l)
                MP_ERR(MP_ERR_INDEX);

        r = N[*N - l];
        return (r & (1 << b));
}
```

Listing 5.7: Get bit n.

The `mp_clrbit` routine is similar to `mp_getbit` and follows the same pattern. The difference is on lines 14 to 16. First we get the digit and mask out the bit we are interested in, line 14. Then we invert the shifted one to get a digit with all bits set except the one we want to clear. We then assign the bitwise and of the digit and the bitmask to the digit (line 15). Finally we return the old value of the bit n. This is sometimes convenient, since we do not have to first execute `mp_getbit` to save a bit that we later want to restore.

```
mp_limb_t
mp_clrbit(mp_t N, mp_size_t n)
{
        mp_size_t l;
        mp_size_t b;
        mp_limb_t r;

        l = n / w;
        b = n % w;

        if (*N <= l)
```

```
                MP_ERR(MP_ERR_INDEX);

        r = N[*N - l] & (1 << b);
15      N[*N - l] &= ~(1 << b);
        return r;
}
```

Listing 5.8: Clear bit n

The routine mp_setbit is similar to mp_getbit and follows the same pattern. mp_getbit. The only difference is on line 15. Instead of computing the bitwise and of the inverted bit pattern we simply assign the bitwise or of the shifted one to the target digit. I.e. the bit b of the target digit will be set. As in the mp_clrbit function we return the old value of the digit.

```
mp_limb_t
mp_setbit(mp_t N, mp_size_t n)
{
        mp_size_t l;
5       mp_size_t b;
        mp_limb_t r;

        l = n / w;
        b = n % w;
10
        if (*N <= l)
                MP_ERR(MP_ERR_INDEX);

        r = N[*N - l] & (1 << b);
15      N[*N - l] |= (1 << b);
        return r;
}
```

Listing 5.9: Set bit n

The routine mp_highbit finds the most significant set bit in a multiple precision number. If we represent the number n in the base 2 then the highest bit set of n corresponds to $\lfloor log_2(n) \rfloor$. Note that, mp_highbit below returns 1 if only the least significant bit is set. This is different from the mp_getbit, mp_clrbit and mp_setbit routines above (listings

5.7, 5.8 and 5.9) where the least significant bit has index zero. I.e. `mp_highbit` returns the index of the highest set bit plus one.

Looking at listing 5.10 we start by finding the length of N (line 7) and then we get the most significant digit (line 8).

```
mp_size_t
mp_highbit(mp_t N)
{
        mp_size_t NL, i;
        mp_limb_t n;

        NL = mp_len(N);
        n = N[*N + 1 - NL];

        if (NL == 0) {

                return 0;
        }

        i = 0;

        while (n) {
                n >>= 1;
                i++;
        }

        return i + (NL - 1) * w;
}
```

Listing 5.10: Find the highest set bit.

If the length is zero, then there are no bits set and we return zero (lines 10 and 12). Else we go on to count the bits in the most significant word. We start by initializing i to zero (line 15) and iterate over n, shifting n right and increasing i in each iteration (lines 17 to 19). Eventually we will have shifted out all bits of n and we have the number of bits to the highest bit, counted from the left, in i.

On line 22 we compute the position of the highest set bit in N as the number of bits below the most significant digit plus the number of bits to the highest bit set in the most significant digit and return the result.

5.6 Signed addition and subtraction

Our representation of multiple presicion numbers as an array of memory cells holding the size of the array and a sequence of digits in the base 2^w does not include any information about the sign of the number. One simple extension to the representation would be to reserve one bit of the cell holding the size for the sign of the number. However, since the need for signed addition and subtraction is limited we instead provide routines for signed addition and subtraction where we must handle the sign separately.

Sign of X	Sign of Y	$\lvert X\rvert \geq \lvert Y\rvert$	$\lvert X\rvert < \lvert Y\rvert$
$X > 0$	$Y > 0$	$(+1) \cdot (X + Y)$	$(+1) \cdot (X + Y)$
$X > 0$	$Y < 0$	$(+1) \cdot (X - Y)$	$(-1) \cdot (Y - X)$
$X < 0$	$Y > 0$	$(-1) \cdot (X - Y)$	$(+1) \cdot (Y - X)$
$X < 0$	$Y < 0$	$(-1) \cdot (X + B)$	$(-1) \cdot (X + Y)$

Table 5.3: Signed addition

First, we note some facts about adding and subtracting numbers both positive and negative. We see that both the sign and the relative absolute values of the two operands are important. For example, if $x > 0$ and $y < 0$ and $x < y$ then the sum $s = x + y$ is equal to $-1 \cdot (|y| - |x|)$. For both addition and subtraction there are eight such cases which we must handle separately. See table 5.3 and table 5.4.

The implementations of signed addition and signed subtraction are straightforward. Looking at listing 5.11, first we determine if $X \geq Y$, and set s accordingly (lines 8 to 14). Then combine the signs of X and Y into a two bit value which correspond to the four cases in table 5.3.

If both X and Y are positive (line 17) then the sum is positive and equal to $X + Y$ so we set the sign of S, the sum to 1 and S itself to $X + Y$ (lines 19 and 22).

```
    mp_limb_t
2   mp_signadd(mp_t S, mp_sign_t *sS,
            mp_t X, mp_sign_t sX, mp_t Y, mp_sign_t sY)
    {

            mp_sign_t s;
7
```

```
        if (mp_gte(X, Y)) {

                s = 1;

12      } else {

                s = 0;
        }

17      switch ((sX << 1) | sY) {

                case ((MP_POS << 1) | MP_POS):

                        *sS = 1;
22                      return mp_add(S, X, Y);
                        break;

                case ((MP_POS << 1) | MP_NEG):

27                      if (s) {

                                *sS = 1;
                                return mp_sub(S, X, Y);

32                      } else {

                                *sS = 0;
                                return mp_sub(S, Y, X);
                        }
37                      break;

                case ((MP_NEG << 1) | MP_POS):

                        if (s) {
42
                                *sS = 0;
                                return mp_sub(S, X, Y);

                        } else {
47
```

```
                    *sS = 1;
                    return mp_sub(S, X, Y);
                 }
                 break;
52
            case ((MP_NEG << 1) | MP_NEG):

                 *sS = 0;
                 return mp_add(S, X, Y);
57               break;

            default:
                 MP_ERR(MP_ERR_NEVER);
       }
62 }
```

Listing 5.11: Signed addition

If X is positive but Y is negative (line 25) we need to know if $X \geq Y$ or not. If $X > Y$ then $S = X - Y$ and S is positive (lines 29 and 30). If $X < Y$ then $S = -(Y - X)$ and we set the sign of S to negative and S to $Y - X$ (lines 34 and 35).

If X is negative and Y is positive (line 39) then we again need to know if $X \geq Y$ or not. If it is (line 41) then we set the sign of S to negative and S to $X - Y$ (lines 43 and 44). If not, i.e. if $X < Y$ then S is positive and $S = X - Y$ (lines 48 and 49). Finally, if both X and Y are negative (line 53) then S is also negative and we set the sign of S accordingly, also we set S to $X + Y$ (line 55 and 56).

Sign of X	Sign of Y	$\|X\| \geq \|Y\|$	$\|X\| < \|Y\|$
$X > 0$	$Y > 0$	$(+1) \cdot (X - Y)$	$(+1) \cdot (Y - X)$
$X > 0$	$Y < 0$	$(+1) \cdot (X + Y)$	$(+1) \cdot (X + Y)$
$X < 0$	$Y > 0$	$(-1) \cdot (X + Y)$	$(-1) \cdot (Y + X)$
$X < 0$	$Y < 0$	$(-1) \cdot (X - B)$	$(+1) \cdot (Y - X)$

Table 5.4: Signed subtraction

Looking at listing 5.12 we see that the signed subtraction is very similar to the signed addition except for the eight cases which are obviously different (see table 5.4).

```
mp_limb_t
mp_signsub(mp_t D, mp_sign_t *sD,
        mp_t X, mp_sign_t sX, mp_t Y, mp_sign_t sY)
{

        mp_sign_t s;

        if (mp_gte(X, Y)) {

                s = 1;

        } else {

                s = 0;
        }

        switch ((sX << 1) | sY) {

                case ((MP_POS << 1) | MP_POS):

                        if (s) {

                                *sD = 1;
                                return mp_sub(D, X, Y);

                        } else {

                                *sD = 0;
                                return mp_sub(D, X, Y);
                        }
                        break;

                case ((MP_POS << 1) | MP_NEG):

                        *sD = 1;
                        return mp_add(D, X, Y);
                        break;

                case ((MP_NEG << 1) | MP_POS):
```

```
43                      *sD = 0;
                        return mp_add(D, X, Y);
                        break;

                case ((MP_NEG << 1) | MP_NEG):

48                      if (s) {

                                *sD = 0;
                                return mp_sub(D, X, Y);

53                      } else {

                                *sD = 1;
                                return mp_sub(D, X, Y);
                        }
                        break;
58
                default:
                        MP_ERR(MP_ERR_NEVER);
        }
}
```

Listing 5.12: Signed subtraction

Note that both `mp_signadd` and `mp_signsub` are in place safe. Thus, for example, `mp_signadd(X, sX, X, sX, Y)` produces the expected result.

5.7 Greatest common divisor

The greatest common divisor, from definition 8, can be computed using Euclid's algorithm.

Consider the two number $a > 0$ and $b > 0$ where $a > b$ and assume $g|a$ and $g|b$. Then $a = gx$ and $b = gy$, $x > 0$ and $y > 0$. Dividing a with b yields q and r such that $a = qb + r$ and $r = a - qb$. Now, since $a = gx$ and $b = gy$ we have $r = g(x - qy)$ and $g|r$. Continuing, assume $h|b$ and $h|r$, then $b = hm$ and $r = hn$. Since $a = qb + r$ we have $a = h(qm + n)$ and $h|a$.

Thus all common factors g of a and b are common factors of b and r. Also, all common factors of h of b and r are common factors of a and b. That is, in particular, for the greatest common divisor we have $\gcd(a, b) = \gcd(b, r)$ where $r = a - qb$ and $r < b$. In fact, $\gcd(a, b)$ is equal to $\gcd(b, s)$ where s is any linear combination of a and b.

So, if we repeatedly compute the remainder of a and b, then of b and r, etc, eventually the remainder will be zero and the final divisor is the greatest common divisor of a and b. This is Euclid's algorithm, or the Euclidean algorithm (algorithm 5) described below.

Algorithm 5. Euclid's algorithm .
Euclid's algorithm computes the greatest common divisor of a and b.

In: Two positive integers a and b, $a \geq b$.

Out: The greatest common divisor of a and b, $\gcd(a, b)$.

1: $a_1 = a$
2: $b_1 = b$
3: compute q_1 and r_1 in $a_1 = q_1 b_1 + r_1$
4: $i = 1$
5: **while** $r_i \neq 0$ **do**
6: $\quad i = i + 1$
7: $\quad a_i = b_{i-1}$
8: $\quad b_i = r_{i-1}$
9: $\quad$ compute q_i and r_i in $a_i = q_i b_i + r_i$
10: **end while**
11: **return** r_{i-1}

Euclid's algorithm, in each step, generates a pair of integers which form a sequence

$$(a_1, b_1), (a_2, b_2), \ldots (a_k, b_k)$$

where $(a_1, b_1) = (a, b)$ and $(a_i, b_i) = (b_{i-1}, r_{i-1})$. The integers a_i and b_i are computed such that $a_i = b_i q_i + r_i$. Now, the algorithm terminates, because the remainder r_i is always less than the divisor b_i and we will have for $i = 1 \ldots k$ that

$$b_1 > b_2 > \cdots > b_k$$

and the algorithm will eventually reach 0, i.e. $b_k = 0$.

At the last iteration we will have $r_k = 0$ and $a_k = q_k b_k$, that is $b_k | a_k$ and $\gcd(a_k, b_k) = b_k$. We know from our discussion above that if g is a common factor of our integers a and b it must also be a common factor of b and r, the remainder of a when divided by b. So, since the sequence $b_1, b_2 \ldots b_k$ is a decreasing sequence of remainders starting with a divided by b, we must have that b_k is the greatest common divisor of a and b.

Euclid's algorithm requires a multiple precision division in step 9. Another algorithm, which does not require multiple precision division is the binary gcd algorithm. The algorithm is also known as Stein's algorithm, after Josef Stein who first published it in 1967.

Algorithm 6. Binary gcd .

The binary gcd algorithm computes the greatest common divisor of two positive integers.

In: Two positive integers X and Y where $X > Y$.

Out: The greatest common divisor of X and Y.

1: $g = 1$
2: **while** even(X) and even(Y) **do**
3: $\quad X = X/2$, $Y = Y/2$, $g = 2 \cdot g$
4: **end while**
5: **while** $X \neq 0$ **do**
6: $\quad$ **while** even(X) **do**
7: $\quad\quad X = X/2$
8: $\quad$ **end while**
9: $\quad$ **while** even(Y) **do**
10: $\quad\quad Y = Y/2$
11: $\quad$ **end while**
12: $\quad t = \frac{|X-Y|}{2}$
13: $\quad$ **if** $X \geq Y$ **then**
14: $\quad\quad X = t$
15: $\quad$ **else**
16: $\quad\quad Y = t$
17: $\quad$ **end if**
18: **end while**
19: **return** $g \cdot Y$

The binary gcd algorithm relies on three facts about the greatest common divisor. Consider the two numbers a and b. First, if both a

and b are even then we can write $a = 2x$ and $b = 2y$. Thus $\gcd(a, b) = \gcd(2x, 2y) = 2 \cdot \gcd(x, y)$. Second, if a is even and b is odd then b is not divisible by two and $\gcd(a, b) = \gcd(a/2, b)$. Third, if both a and b are odd then $a - b$ is even and $|a - b| < max(a, b)$ and $gcd(a, b) = gcd(|a - b|, min(a, b))$. To see why, assume $a > b$ and $\gcd(a, b) = g$ then $a = gx$ and $b = gy$. Now, $a - b$ is a linear combination of a and b and we know from above that $\gcd(a, b) = \gcd(b, g(x - y))$.

Looking at listing 5.13 below, we start by checking three error cases. We require that X, Y and G be the same size and that $X > Y$ (lines 6 to 12). Then we initialize G to one and set up temporary copies of X and Y in TX and TY (lines 15 and 17). Then the initializations are done and we are ready to start the computation of the greatest common divisor of X and Y.

```
void
mp_gcd(mp_t G, mp_t X, mp_t Y)
{
        mp_t TX, TY;
5
        if (*X != *Y)
                MP_ERR(MP_ERR_OPSIZE);

        if (*G < *Y)
10              MP_ERR(MP_ERR_RESSIZEOPLEN);

        if (mp_lt(X, Y))
                MP_ERR(MP_ERR_ARG);

15      mp_one(G);

        TX = mp_tmp(*X);
        mp_set(TX, X);

20      TY = mp_tmp(*Y);
        mp_set(TY, Y);

        while(mp_even(TX) && mp_even(TY)) {

25              (void)mp_sr(TX, TX, 1);
                (void)mp_sr(TY, TY, 1);
```

```
                (void)mp_sl(G, G, 1);
        }

30      while (mp_neqz(TX)) {

                while (mp_even(TX)) {

                        (void)mp_sr(TX, TX, 1);
35              }

                while (mp_even(TY)) {

                        (void)mp_sr(TY, TY, 1);
40              }

                if (mp_gte(TX, TY)) {

                        mp_sub(TX, TX, TY);
45                      (void)mp_sr(TX, TX, 1);

                } else {

                        mp_sub(TY, TY, TX);
50                      (void)mp_sr(TY, TY, 1);
                }
        }

        mp_set(TX, G);
55      mp_mul(G, TX, TY);
}
```

Listing 5.13: Binary gcd algorithm

First, while X and Y (in TX and TY) are both even (line 23) we divide X by two, Y by two and multiply G by two. This corresponds to the first fact above. We use right and left shift for the division and multiplication by two (lines 25 to 27).

Then we enter the while loop on line 30. We iterate until X is zero. First on line 32, if X is even, we divide X by two (line 34) and repeat until X is odd. Then we do the same with Y (lines 37 and 39). This corresponds to the second fact above. Finally, the third fact above, if

$X \geq Y$ (line 42) then we set X to $\frac{X-Y}{2}$ (lines 44 and 45. If instead $Y > X$ (line 47) then we set Y to $\frac{Y-X}{2}$. When X is zero we exit the loop and return $Y \cdot G$ as the greatest common divisor of X and Y.

The extended gcd algorithm computes the greatest common divisor of two integers (X and Y) as well as the two integers a and b that satisfies Bézout's identity (theorem 5) $aX + bY = \gcd(X, Y)$.

Algorithm 7. Extended gcd .
The extended gcd algorithm computes the greatest common divisor of two positive integers (X and Y) and the two integers a and b satisfying $aX + bY = g$.

In: Two positive integers X and Y where $X > Y$.

Out: The greatest common divisor of X and Y, i.e. $g = \gcd(X, Y)$ and integers a and b satisfying $g = aX + bY$.

1: **if** $Y = 0$ **then**
2: $\quad g = X$, $a = 1$, $b = 0$
3: $\quad$ **return** (g, a, b)
4: **end if**
5: $a_2 = 1$, $a_1 = 0$, $b_2 = 0$, $b_1 = 0$
6: **while** $Y > 0$ **do**
7: $\quad$ compute Q and R in $X = QY + R$
8: $\quad X = Y$, $Y = R$
9: $\quad a = a_2 - qa_1$, $a_2 = a_1$, $a_1 = a$
10: $\quad b = b_2 - qb_1$, $b_2 = b_1$, $b_1 = b$
11: **end while**
12: $g = X$, $a = a_2$, $b = b_2$
13: **return** (g, a, b)

Each step of the gcd algorithm computes a divisor. That is, in the first step we divide X by Y and compute $r = X - qY$. The next step is to compute the remainder when Y is divided by r. We continue iterating until the remainder of the division is zero.

The plain gcd algorithm discards the quotients of the division performed in each iteration and only records the remainders. The extended gcd algorithm uses the quotients to compute the Bézout coefficients a and b.

Now, since we are interested in finding the Bézout coefficients, a and b, in each iteration we keep track of the current linear combination (d_i) of X and Y. We see that $a_i = a_{i-2} - qa_{i-1}$ and $b_i = b_{i-2} - qb_{i-1}$.

i	a_i	b_i	q_i	d_i
1	1	0		X
2	0	1		Y
3	$1 - 0q_3$	$0 - 1q_3$	q_3	$X - q_3Y$
4	$0 - 1q_4$	$1 + q_4q_3$	q_4	$Y - q_4(X - q_3Y)$
5	$1 + q_5q_4$	$-q3 - q_5(1 + q_4q_3)$	q_5	$X - q_3Y-$ $q_5(Y - q_4(X - q_3Y))$

Table 5.5: Extended gcd algorithm.

We know that each remainder (divisor) can be expressed as $d_i = a_iX + b_iY$. Also, each d_i can be expressed as $d_i = d_{i-2} - qd_{i-1}$. The computed values are tabulated in table 5.5. The first two rows are the initial values, i.e. $1 \cdot X + 0 \cdot Y$ and $0 \cdot X + 1 \cdot Y$. The first real iteration ($i = 3$) computes $r = X - q_3Y$ and we have the values $a_i = 1 - 0 \cdot q_3$ and $b_i = 0 - 1 \cdot q_3$. Continuing, we see that it suffices to remember the two previous values of a_i and b_i to compute the current value. When the algorithm terminates, with a remainder of zero, we will have the desired coefficients. This is the essence of the extended gcd algorithm, algorithm 7.

The binary extended gcd algorithm (algorithm 8) follows the same pattern as the binary gcd algorithm presented earlier (algorithm 6) but also computes the Bézout coefficients a and b in the same fashion as the extended gcd algorithm (algorithm 7).

Algorithm 8. Binary extended gcd .

The binary extended gcd algorithm computes the greatest common divisor of two positive integers (X and Y) and the two integers a and b satisfying $aX + bY = g$.

In: Two positive integers X and Y where $X > Y$.

Out: The greatest common divisor of X and Y, i.e. $g = \gcd(X, Y)$ and integers a and b satisfying $g = aX + bY$.

1: $g = 1$
2: **while** even(X) and even(Y) **do**
3: $\quad X = \frac{X}{2}$, $Y = \frac{Y}{2}$, $g = 2 \cdot g$
4: **end while**
5: $u = X$, $v = Y$, $A = 1$, $B = 0$, $C = 0$, $D = 1$

```
6: while even(u) do
7:     u = u/2
8:     if A ≡ B ≡ 0 (mod 2) then
9:         A = A/2, B = B/2
10:    else
11:        A = (A+Y)/2, B = (B−X)/2
12:    end if
13: end while
14: while even(v) do
15:    v = v/2
16:    if C ≡ D ≡ 0 (mod 2) then
17:        C = C/2, D = D/2
18:    else
19:        C = (C+Y)/2, D = (D−X)/2
20:    end if
21: end while
22: if u ≥ v then
23:    u = u − v, A = A − C, B = B − D
24: else
25:    v = v − u, C = C − A, D = D − B
26: end if
27: if u = 0 then
28:    a = C, b = D
29:    return (a, b, g · v)
30: else
31:    goto 6
32: end if
```

Looking at listing 5.14, the first lines, line 9 to 29 checks for errors and initializes the temporary variables used in the algorithm. The sizes of X, Y and V must all be the same, which we check on lines 9 to 15. Then we allocate space for the temporary variables G, U, C, D, T, TX and TY (lines 18 to 24) and set TX, our temporary copy of X, and also TY (lines 26 and 27). Next we set up the signs of all our working variables. We consider them all to be positive at start (line 29). Finally we initialize G to one, line 32.

We now go on to the main computation of the algorithm. The `while`-loop on line 34 to line 39 divides X and Y by two and multiplies G by two while both X and Y are even.

```
void
mp_extgcd(mp_t A, mp_sign_t *sA, mp_t B,
        mp_sign_t *sB, mp_t V, mp_t X, mp_t Y)
{

        mp_t  G, U, C, D, T, TX, TY;
        mp_sign_t sC, sD, sU, sV, sX, sY, sT;

        if (*X != *Y)
                MP_ERR(MP_ERR_OPSIZE);

        if (*X != *V)
                MP_ERR(MP_ERR_RESSIZEOPLEN);

        if (*A != *X || *B != *X)
                MP_ERR(MP_ERR_RESSIZEOPlEN);

        G = mp_tmp(*X);
        U = mp_tmp(*X);
        C = mp_tmp(*X);
        D = mp_tmp(*X);
        T = mp_tmp(*X);
        TX = mp_tmp(*X);
        TY = mp_tmp(*Y);

        mp_set(TX, X);
        mp_set(TY, Y);

        *sA = *sB = sC = sD =
                sU = sV = sX = sY = MP_POS;

        mp_one(G);

        while (mp_even(TX) &&
                mp_even(TY)) {

                (void)mp_sr(TX, TX, 1);
                (void)mp_sr(TY, TY, 1);
                (void)mp_sl(G, G, 1);
        }
```

```
        mp_set(U, TX);
        mp_set(V, TY);
        mp_one(A);
45      mp_zero(B);
        mp_zero(C);
        mp_one(D);

again:
50      while (mp_even(U)) {

                (void)mp_sr(U, U, 1);

                if ((mp_eqz(A) ||
55                      mp_even(A)) &&
                        (mp_eqz(B) ||
                        mp_even(B))) {

                        (void)mp_sr(A, A, 1);
60                      (void)mp_sr(B, B, 1);

                } else {

                        mp_signadd(T, &sT, A,
65                              *sA, TY, sY);
                        (void)mp_sr(A, T, 1);
                        *sA = sT;
                        mp_signsub(T, &sT, B,
                        *sB, TX, sX);
70                      (void)mp_sr(B, T, 1);
                        *sB = sT;
                }
        }

75      while (mp_even(V)) {

                (void)mp_sr(V, V, 1);

                if ((mp_eqz(C) || mp_even(C)) &&
80                      (mp_eqz(D) || mp_even(D))) {
```

```
                        (void)mp_sr(C, C, 1);
                        (void)mp_sr(D, D, 1);

85              } else {

                        mp_signadd(T, &sT, C,
                                sC, TY, sY);
                        (void)mp_sr(C, T, 1);
90                      sC = sT;
                        mp_signsub(T, &sT, D,
                                sD, TX, sX);
                        (void)mp_sr(D, T, 1);
                        sD = sT;
95              }
        }

        if (mp_gte(U, V)) {

100             mp_sub(U, U, V);

                mp_signsub(T, &sT, A, *sA, C, sC);
                mp_set(A, T); *sA = sT;
                mp_signsub(T, &sT, B, *sB, D, sD);
105             mp_set(B, T); *sB = sT;

        } else {

                mp_sub(V, V, U);
110
                mp_signsub(T, &sT, C, sC, A, *sA);
                mp_set(C, T); sC = sT;
                mp_signsub(T, &sT, D, sD, B, *sB);
                mp_set(D, T); sD = sT;
115     }


        if (mp_eqz(U)) {

120             mp_set(A, C); *sA = sC;
```

```
            mp_set(B, D); *sB = sD;
            mp_mul(T, G, V);
            mp_set(V, T);

125         return;

      } else {

            goto again;
130   }
}
```

Listing 5.14: Extended Euclid

Next, U is set to the current value of X and V to the current value to Y and the initial values of the coefficients A, B, C and D are set up (lines 42 to 47). Now, $A = 1$, $B = 0$, $C = 0$ and $D = 1$ which corresponds to $X = AX + BY$ and $Y = CX + DY$ i.e. $X = 1 \cdot X + 0 \cdot Y$ and $Y = 0 \cdot X + 1 \cdot Y$.

Continuing, while U is even (U is initially set to the current value of X) we divide U by two, by shifting U right one bit (line 52), then, if A and B are even we divide both by two (again by shifting them right one bit) on lines 59 and 60. If, on the other hand, one of A or B is odd we instead compute, in T, the difference between A and Y (line 64) and assign the result, divided by two, to A (line 66). Also, we compute $B = \frac{B-X}{2}$ on lines 68 and 70.

Next, if V is even (line 75) we do a similar computation to C and D. That is, first we divide V by two (line 77), then if C and D are both even (line 5.14) we set $C = \frac{C}{2}$ and $D = \frac{D}{2}$. If one or both of them are odd we instead compute $C = \frac{C-Y}{2}$ and $D = \frac{D-X}{2}$ (lines 87 to 94).

Now, if $U \geq V$ (line 98) we compute $U = U - V$, $A = A - C$ and $B = B - D$, on lines 102 to 105. On the other hand, if $U < V$ we instead compute $V = V - U$, $C = C - A$ and $D = D - B$ (lines 111 to 114).

The algorithm continues until U is equal to zero. If it is (line 118) we set $A = C$, $B = D$ and $V = G \cdot V$ as our result and terminate. If not, we repeat the computation by continuing at the label `again` (line 49) until U is zero. The values of A and B are the Bézout coefficients in the equation $AX + BY = V$, where $V = \gcd(X, Y)$.

5.8 Modular exponentiation

Modular exponentiation is of fundamental importance to public key encryption. The straightforward method of computing a modular exponent is to compute, first N^e, and then take the modulus, $N^e \bmod M$. If N is an N_L digit number, the result of squaring N will require $2N_L$ digits. Then computing N^e will require eN_L digits. If N is a 512 bit number and e is a 256 bit number N^e will require more than $512 \cdot 2^{255}$ digits. Clearly the straightforward method is not efficient for large e.

Consider the modular multiplication $a \equiv xy \bmod m$. Assume $b \equiv x \bmod m$ and $c \equiv y \bmod m$. Now, by definition 11 we have $\exists i : b-x = im$ and $\exists j : c-y = jm$. We see that the product of x and y is $(b-im)(c-jm)$ which we can write $bc + m(bj + ci + ijm)$. Also, we have by the same definition that $\exists k : a - xy = km$, or that $xy = a - km$. Let $a = bc$ and $km = m(bj + ci + ijm)$ and we have that $xy \bmod m = (x \bmod m) \cdot (y \bmod m) \bmod m$.

This fact makes it possible to compute modular exponentiation using repeated multiplication without using more memory than the $2 \cdot N_L$ digits of N^2. Instead of computing $N \cdot N \ldots N \bmod m$ we can compute

$$N \cdot N \bmod m \cdot N \bmod m \cdots N \bmod m$$

i.e. performing a modular multiplication e times. This alleviates the memory problem in the straightforward solution, unfortunately, if e is large, the number of multiplications that must be performed makes the approach infeasible.

Let e be represented in the base 2, then $e = \sum_{i=0}^{n-1} e_i 2^i$ and we have e as $e = [e_{n-1}e_{n-2} \ldots e_1 e_0]_2$. Now, we can write N^e as

$$N^e = N^{\sum_{i=0}^{n-1}(e_i 2^i)} = \prod_{i=0}^{n-1} (N^{2^i})^{e_i}$$

That is, N^e is the product of the repeated squares of N, i.e. N^2, N^4, N^8, etc, and N if e is odd, or 1 if e is even. So, we have $N^e \pmod{m}$ as

$$N^e \pmod{m} = \prod_{i=0}^{n-1} (N^{2^i})^{e_i} \pmod{m}$$

or

$$N^{e_0} \bmod m \cdot (N^2)^{e_1} \bmod m \cdots (N^{2^{n-1}})^{e_{n-1}} \bmod m$$

This will require $\ln(e)$ multiplications and divisions and require $2 \cdot \ln(N)$ bits of memory to evaluate. This is a reasonably efficient method of computing modular exponentiation.

Algorithm 9. Modular exponentiation .
The modular exponentiation algorithm computes $N^e \bmod M$.

In: Two positive integers N, M and e where $e = \sum_{i=0}^{n} 2^i e_i$. That is, e is an $n+1$ bit number.

Out: The result $N^e \bmod M$

1: $r = 1$
2: **if** $e = 0$ **then**
3: **return** r
4: **end if**
5: $A = N$
6: **if** $e_0 = 1$ **then**
7: $r = A$
8: **end if**
9: **for** $i = 1 \ldots n$ **do**
10: $A = A^2 \bmod M$
11: **if** $k_i = 1$ **then**
12: $r = A \cdot r \bmod M$
13: **end if**
14: **end for**
15: **return** r

Looking at listing 5.15 we have an implementation of the modular exponentiation routine. We start by initializing the result R to one (line 10) then we check if E is zero, if so we return with the result $R = 1$, lines 12 and 14. We find the lengths of N and M on lines 17 and 18. If the size of R is less than the length of M than we generate an error, line 20. We also find the maximum length of N and M (lines 23 to 26) and create two temporary variables T and U which we use in our computation (line 28 and 29. They must be of sufficient length to hold N^2. We set T to N (line 31) and we are done initializing so we go on with the main computation.

```
void
mp_expmod(mp_t R, mp_t N, mp_t E, mp_t M)
{
```

```
5        mp_limb_t One[2] = {1, 1};
         mp_size_t L, NL, ML;
         mp_size_t i, t, e;
         mp_t  T, U;

10       mp_one(R);

         if (mp_eqz(E)) {

                return;
15       }

         NL = mp_len(N);
         ML = mp_len(M);

20       if (*R < mp_len(M))
                MP_ERR(MP_ERR_RESSIZEOPLEN);

         if (ML < NL)
                L = NL;
25       else
                L = ML;

         T = mp_tmp(2 * L);
         U = mp_tmp(2 * L);
30
         mp_set(T, N);


         if (mp_odd(E)) {
35              mp_set(R, N);
         }

         t = mp_highbit(E) - 1;

40       for (i = 1; i <= t; i++) {

                mp_sqr(U, T);
                mp_mod(T, U, M);
```

```
45              if (mp_getbit(E, i)) {

                        mp_mul(U, T, R);
                        mp_mod(R, U, M);
                }
50      }
}
```

Listing 5.15: Binary modular exponentiation

If E is odd, i.e. if $e_0 = 1$ then we set R to N (otherwise R remains set to one), lines 34 and 35. We let t be the index of the highest set bit in E, i.e. the number of significant bits in E minus one (line 38) and enter the main loop on line 40. We iterate from the second bit to the last set bit in E. The temporary variable T is our accumulator, it is initially set to one or N if E is even or odd, respectively. We compute $T^2 \bmod M$ on lines 42 and 43. If bit i (the current iteration) is set (line 45) then we compute $R \cdot T \bmod M$ in R (lines 47 and 48).

This is repeated for all significant bits in E, i.e. until we have processed the highest set bit of E. Then we have our result in R, i.e. $R = N^E \bmod M$.

5.9 Scratch space

Normally, the computer system in use provides a dynamic memory allocator. For example, `malloc`. However, using the external memory allocator for allocation and deallocation of temporary variables during computation impacts performance. We can improve performance by providing a simple and fast memory handler for temporary variables.

The scratch memory allocator is essentially a stack. It is initialized with a memory area reserved for temporary storage. A routine that will use the scratch area saves a mark of the first free word in the scratch area (`mp_tmp_mark`) . Then it allocates temporary variables used in computation (`mp_tmp`). When the routine exits it releases the temporary memory (`mp_tmp_release`).

The scratch space management relies on the variable `mp_scratch` which must be globally visible. It is a record that contains the start and end of the scratch memory buffer, an element pointing to the first unused word of the scratch buffer and a high water mark that can be

used for debugging and tuning of temporary storage. See listing 5.16 for the definition.

```
typedef struct mp_tmp_s {
        mp_limb_t *b;
        mp_limb_t *e;
        mp_limb_t *p;
        mp_size_t max;
} mp_tmp_t;

#define MP_TMP_SCRATCH_DECL mp_tmp_t mp_scratch
#define MP_TMP_DECL mp_mark_t mp_mark
#define MP_TMP_MARK mp_mark = mp_tmp_mark()
#define MP_TMP_RELEASE mp_tmp_release(mp_mark)

MP_TMP_SCRATCH_DECL;
```

Listing 5.16: Scratch space declarations

The `MP_TMP_SCRATCH_DECL` statement defines the variable `mp_scratch`, which must be in global scope. Each routine that uses the scratch memory must, declare a mark variable using the `MP_TMP_DECL` and before using any scratch space it must also take a mark of the beginning of the unused area of the scratch buffer using `MP_TMP_MARK`. Then scratch variables may be allocated using `mp_tmp`. Before exiting the routine the used portion of the scratch area must be released, using `MP_TMP_RELEASE`.

The `mp_tmp_init` routine sets up the scratch area for use. The scratch buffer `b` is a previously allocated memory area with room for n digits. Looking at listing 5.17 we set the members of the global `mp_scratch` record, lines 4 to 6. We save the beginning, b, and the end, e of the scratch buffer. Also the member p points to the first unused word of the scratch area.

```
void
mp_tmp_init(mp_limb_t *b, mp_size_t n)
{
        mp_scratch.b = b;
        mp_scratch.e = b + n;
        mp_scratch.p = b;
}
```

Listing 5.17: Scratch space initialization

The routine mp_tmp_mark returns a mark of the first unused word of the scratch area, i.e. on line 4 the member p of the mp_scratch record is returned.

```
mp_mark_t
mp_tmp_mark()
{
        return mp_scratch.p;
}
```

Listing 5.18: Marking scratch space

Now, the routine mp_tmp allocates an n digit multiple precision number and returns it. Note that an n digit multiple precision number requires $n+1$ memory cells (we store the size at index zero). Looking at listing 5.19 we start by checking so that there is enough scratch space remaining for the requested multiple precision number (line 6). If not we generate an error (line 7).

```
mp_t
mp_tmp(mp_size_t n)
{
        mp_limb_t *l;

        if ((mp_scratch.p + n + 1) > mp_scratch.e) {
              MP_ERR(MP_ERR_SCRATCH);
        }

        l = mp_scratch.p;
        *l = n;
        mp_zero(l);
        mp_scratch.p += (n + 1);
        if ((mp_scratch.p - mp_scratch.b) >
                mp_scratch.max)
                mp_scratch.max = mp_scratch.p -
                        mp_scratch.b;
        return l;
}
```

Listing 5.19: Allocating scratch space

Then, if there is room, we set l to the current start of the unused portion of the scratch area (line 10). This will be the first word of the

multiple precision number returned. We set the first word to n, i.e. the number of digits in the number (line 11). We also initialize the allocated multiple precision number to zero, line 13. Then we increase the used portion of the scratch area with $n + 1$ words on line 13. We also store a high water mark used for debugging and tuning. That is, if the current scratch space usage is more than any previous usage we store the current usage in the element *max* (lines 14 and 16). Finally we return the allocated and initialized multiple precision number, line 18.

```
void
mp_tmp_release(mp_mark_t m)
{
        mp_scratch.p = m;
}
```

Listing 5.20: Reclaiming scratch space

The routine `mp_tmp_release` reclaims the scratch space used by a routine. A routine that allocates scratch space must release the allocated scratch space by a call to the routine `mp_tmp_release` with the stored mark before exiting to deallocate the used scratch space. Looking at listing 5.20 we reset the element that points to the first unused byte of the scratch space to the mark m, on line 4.

5.10 Error handling

Errors that occur during computation must be handled somehow. Commonly, errors are indicated through return values of routines. However, we present an error handling mechanism based on the standard library functions `setjmp` and `longjmp`.

The routine `mp_init` in section 5.11 installs an error handler. Whenever a routine generates an error the error handler will be called. If the user does not supply an error handler a default handler is installed.

The error handling relies on the three global variables; `mp_error`, `mp_handler` and `mp_jumpbuf`. The `mp_error` variable stores the most recent error, `mp_handler` stores a pointer to the error handler and finally, `mp_jumpbuf` stores a context so that the execution can continue at the error handler if an error is generated.

```
typedef void  (*mp_errorhandler_f)(void);
```

```
jmp_buf   mp_jumpbuf;
mp_error_t  mp_error;
mp_errorhandler_f mp_handler;

typedef enum mp_error_e {
       MP_ERR_ALLOC, MP_ERR_SCRATCH, MP_ERR_RESSIZE,
       MP_ERR_OPSIZE, MP_ERR_RESSIZEOPLEN,
       MP_ERR_NEG, MP_ERR_EVENBYTE, MP_ERR_ODD,
       MP_ERR_ARG, MP_ERR_INDEX, MP_ERR_DIVZ,
       MP_ERR_INPLACE, MP_ERR_NOTIMPL, MP_ERR_NEVER
} mp_error_t;

static char *mp_errorstr[] = {
              "memory allocation error",
              "out of scratch space",
              "size of result",
              "length or size of operand(s)",
              "size of result less than"
              "length of parameter(s)",
              "result will be negative",
              "must be even byte",
              "must be odd",
              "argument error",
              "index out of range",
              "division by zero",
              "not in place safe",
              "not implemented",
              "should never happen"
};
```

Listing 5.21: Error handling declarations

The routine mp_err in listing 5.22 generates an error and transfers execution to the error handler. First the error is stored in mp_error, on line 4. Then, if debug is enabled, the eror message is printed (line 5). Finally execution is transfered to the error handler by a longjump, line 6.

```
void
mp_err(char *s1, mp_error_t e)
{
```

```
        mp_error = e;
        MP_DBG(mp_strerror());
        longjmp(mp_jumpbuf, -1);
}
```

Listing 5.22: Generating an error

The routine `mp_strerror`, in listing 5.23 returns a string with a readable description of the error. It uses the error code as an index into an array of strings and returns the string at the index.

```
char *
mp_strerror()
{
        return mp_errorstr[mp_error];
}
```

Listing 5.23: Error messages

The default error handler `mp_errorhandler`, in listing 5.24 simply prints the error message and exits.

```
void
mp_errorhandler()
{
        printf("mp: %s\n", mp_strerror());
        exit(mp_error);
}
```

Listing 5.24: Default error handler

Now, since the error management relies on `setjmp` and `longjmp` it is important that execution does not continue beyond the routine that issued the `setjmp`. That is, an error is fatal and execution must terminate after handling the error, i.e. the error handler must terminate execution. If it does not the behavior is undefined.

5.11 Initialization

The `mp_init` routine initializes the multiple precision library before use. It sets up the scratch space and installs the desired error handler. Also, the `mp_init` routine is responsible for calling the error handler if an error occurs. Looking at listing 5.25 we start by initializing the scratch space

with the buffer b containing n digits. The buffer is provided by the caller (line 5). Then we install the desired error handler in the global variable `mp_handler`, line 6. Finally we set up the execution state so we can receive a call from the `mp_err` routine, on line 7.

```
void
mp_init(mp_errorhandler_f hnd,
        mp_limb_t *b, mp_size_t n)
{
        mp_tmp_init(b, n);
        mp_handler = hnd;
        if (setjmp(mp_jumpbuf) != 0) {
                mp_handler();
        }
}
```

Listing 5.25: Initialization

When `setjmp` is called it will return zero. When a corresponding `longjmp` is issued using the `mp_jumpbuf` state execution will once again continue as if `setjmp` just returned, but with a nonzero return value. Thus if an error has been generated execution will return to line 8 where the error handler is called.

Chapter 6

Random numbers

> "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number – there are only methods to produce random numbers, and a strict arithmetic procedure of course is not such a method."
>
> – John von Neumann.

6.1 Randomness

Randomness is the lack of order, purpose, pattern or definite plan. Something is *random* if it is without pattern, purpose and cause. Something can be, for instance, selected *randomly*, i.e. without cause, plan or pattern.

Say that we have a source of bits. The bits produced by our source can be *biased* or *unbiased* as defined below (definition 21.

Definition 21. Unbiased.

> Let b be a bit which takes the value 0 with probability p and 1 with probability $1 - p$. The *bias* is ϵ in $\epsilon = p - \frac{1}{2}$. If the bias of b is 0 then we say that b is *unbiased*.

We are primarily interested in random bits, that is a bit source that produces unbiased bits.

Definition 22. Truly random.

A bit, b is truly random if it is unbiased and independent.

A sequence is *statistically random* if there are no recognizable pattern or regularity to it. An example of a statistically random sequence is the results of repeated dice rolls. Statistical randomness does not necessarily imply objective unpredictability, i.e. *true randomness*.

We make a distinction between *global* randomness and *local* randomness. In general, randomness often refer to global randomness. The idea is that over time the whole sequence is random, although it may exhibit subsequences that does not appear random. A sequence that exhibit local randomness appears to be random when a subsequence is examined but may not if a larger subsequence is examined.

For example, the simple *linear congruential* random number generator, defined by the recurrence relation

$$x_{n+1} = (ax_n + c) \bmod m \tag{6.1}$$

Where $0 < x_0 < m$ is the *seed*, $m > 0$ is the *modulus*, $0 < a < m$ the *multiplier* and $0 \leq c < m$ the *increment*. The period of a linear congruential random generator is at most m. It will have a full period (m) if c and m are relatively prime, $a - 1$ is divisible by all prime factors of m and $a - 1$ is a multiple of 4 if m is a multiple of 4.

Linear congruential random number generators are not suitable for cryptography. In [6] George Marsaglia found that the numbers generated by a linear congruential generator fall into planes. That is, for a linear congruential generator with the multiplier a, three consecutive integers generated, (x_1, x_2, x_3), fall on the lattice of points described by all linear combinations of the three points $(1, a, a^2)$, $(0, m, 0)$ and $(0, 0, m)$.

By observing the output of a linear congruential generator we can compute m, see [4]. When m is known it is not difficult to set up a system of equations to find a and c.

Another deficiency of the linear congruential generator is that, if the modulus m is a power of 2 then the low order bits of the generated numbers have a far shorter period than the full period. If $m = b^k$, where b is the base, the n:th least significant digit repeats with at most the period b^n, $n < k$.

However, if the modulus, multiplier and increment is carefully chosen, the output of the resulting linear congruential generator is statistically indistinguishable from a sequence drawn at random. In [5], Park and Miller proposes a modulus of $2^{31} - 1$, a multiplier of 16807 and an increment of zero, i.e.

$$x_n = (16807 x_{n-1} + 0) \bmod 2^{31} - 1 \tag{6.2}$$

This linear congruential generator is known as the *minimal standard*, MINSTD. It has been exhaustively tested and is well understood. Also, it is possible to compute the MINSTD x_n on a computer with a wordsize of $w = 32$ by carefully rewriting the recurrence relation. We have $x_n = ax_{n-1} \bmod m$ (c is zero). Now, $ax_{n-1} \bmod m$ is

$$ax_{n-1} - m\lfloor \frac{ax_{n-1}}{m} \rfloor \tag{6.3}$$

Let $m = aq + r$ then $q = \lfloor \frac{m}{a} \rfloor$ and $r = m - a\lfloor \frac{m}{a} \rfloor$. We add and subtract $m\lfloor \frac{x_{n-1}}{q} \rfloor$ to equation 6.3 to get

$$ax_{n-1} - m\lfloor \frac{ax_{n-1}}{m} \rfloor + m\lfloor \frac{x_{n-1}}{q} \rfloor - m\lfloor \frac{x_{n-1}}{q} \rfloor \tag{6.4}$$

After rearranging equation 6.4 we get

$$ax_{n-1} - m\lfloor \frac{x_{n-1}}{q} \rfloor + m\big(\lfloor \frac{x_{n-1}}{q} \rfloor - \lfloor \frac{ax_{n-1}}{m} \rfloor\big) \tag{6.5}$$

We let $\delta(x_{n-1}) = \lfloor \frac{x_{n-1}}{q} \rfloor - \lfloor \frac{ax_{n-1}}{m} \rfloor$. After substituting $\delta(x_{n-1})$ and $m = aq + r$ equation 6.5 becomes

$$ax_{n-1} - (aq + r)\lfloor \frac{ax_{n-1}}{m} \rfloor + m\delta(x_{n-1}) \tag{6.6}$$

After rearranging equation 6.6 we get

$$a\big(x_{n-1} - q\lfloor \frac{x_{n-1}}{q} \rfloor\big) - r\lfloor \frac{x_{n-1}}{m} \rfloor + m\delta(x_{n-1}) \tag{6.7}$$

And we let $\gamma(x_{n-1}) = a\big(x_{n-1} - q\lfloor \frac{x_{n-1}}{q} \rfloor\big) - r\lfloor \frac{x_{n-1}}{m} \rfloor$. So from the original equation we now have

$$ax_{n-1} \bmod m = \gamma(x_{n-1}) + m\delta(x_{n-1}) \tag{6.8}$$

where

$$\gamma(x_{n-1}) = a(x_{n-1} \bmod q) - r\lfloor\frac{x_{n-1}}{q}\rfloor \tag{6.9}$$

$$\delta(x_{n-1}) = \lfloor\frac{x_{n-1}}{q}\rfloor - \lfloor\frac{ax_{n-1}}{m}\rfloor \tag{6.10}$$

It can be proven that if $r < q$ then for $1 \leq x_{n-1} \leq m-1$ the following holds

1. $\delta(x_{n-1})$ is 0 or 1.
2. $ax_{n-1} \bmod q$ and $r\lfloor\frac{x_{n-1}}{q}\rfloor$ are in $0, 1 \ldots m-1$.
3. $|\gamma(x_{n-1})| \leq m-1$.

If x and y are real numbers in $[0,1]$ then $\lfloor x\rfloor - \lfloor y\rfloor$ is either 0 or 1, and thus follows item 1. Item 2 is a consequence of the definition of q and r and the assumption that $r < q$. Item 3 follows from item 2. So, if r is small ($r < q$) then equation 6.8 can be evaluated whithout producing intermediate results larger than $m-1$.

Now, since we know that $1 \leq x_n \leq m-1$ we must have that $\delta(x_{n-1}) = 0$ if $1 \leq \gamma(x_{n-1}) \leq m-1$ and that $\delta(x_{n-1}) = 1$ if $-(m-1) \leq \gamma(x_{n-1}) \leq -1$. Thus we can implement the MINSTD generator without evaluating $\delta(x_{n-1})$. See algorithm 10.

Algorithm 10. MINSTD.
MINSTD computes a locally random number.

In: The modulus m, the multiplier a and the seed x_0.

Out: A locally random number x_n.

1: $q = \lfloor\frac{m}{a}\rfloor$
2: $r = m \bmod a$
3: $h = \lfloor\frac{x_{n-1}}{q}\rfloor$
4: $l = x_{n-1} \bmod q$
5: $t = a \cdot l$
6: $u = r \cdot h$
7: **if** $t > u$ **then**
8: $\quad x_n = t - u$
9: **else**

10: $x_n = t - u + m$
11: **end if**
12: **return** x_n

```
mp_limb_t
mp_minstdrand()
{

#define MINSTD_M ((unsigned)(1 << 31) - 1)
#define MINSTD_A (16807)
#define MINSTD_Q (MINSTD_M / MINSTD_A)
#define MINSTD_R (MINSTD_M % MINSTD_A)

        mp_limb_t l, h, t, u;

        h = x0 / MINSTD_Q;
        l = x0 % MINSTD_Q;
        t = MINSTD_A * l;
        u = MINSTD_R * h;

        if (t > u)
                x0 = t - u;
        else
                x0 = t - u + MINSTD_M;

        return x0;
}
```

Listing 6.1: MINSTD generator

There are numerous tests for statistical randomness. The first was published 1938 by Sir Maurice George Kendall (1907 – 1983) and Bernard Babington Smith in the Journal of the Royal Statistical society.

Kendall and Smith described four tests. The null hypothesis of each test is that each number in the sequence under test has an equal chance of occuring and that patterns in the sequence are distributed equiprobably.

Frequency The frequency test counts the digits and verifies that the digits has roughly the same frequency.

Serial The serial test counts two digits at a time (01, 02, etc) and verifies that each two digit sequence has roughtly the same frequency.

Poker The poker test counts the frequency of certain five digit sequences (the sequences were based on poker hands).

Gap The gap test examines the distance between zeroes in the sequence, i.e. 00 has a distance of zero, 010 a distance of one, etc.

A sequence passing all four tests within a given degree of significance is declared locally random.

Another definition of randomness is *algorithmic randomness*, or Kolmogorov randomness. The Kolmogorov complexity of an object is a measure of the computational resources required to specify (describe) the object. For example, consider the two sequences

$$(1,\ 0,\ 1,\ 0,\ 1,\ 0,\ 1,\ 0,\ \dots)$$
$$(1,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ \dots)$$

The first sequence can be described "repeat 1, 0 n times". The second sequence is not as simple to describe, however, we can describe it using itself, i.e. say "the sequence (1, 0, 0, 1, 1, 0, 1, 0, ...)". We say that the *complexity* of a sequence is the length of its shortest description in some prescribed description language.

Assume we have a description language, such as a programming language, or a Turing machine encoding (see section 3.2 in chapter 3). If P is a program, written in our description language, that outputs x, then P is a *description* of x. The length of the description is the length of P, interpreted as a character string.

Alternatively, consider the *encoding* of a Turing machine M. The encoding is a function that associates M with a string $\langle M\rangle$. If w is input to M, which produces x, then $\langle M\rangle\ w$ is a *description* of x. We define the universal Turing machine, a programmable Turing machine.

Definition 23. Universal Turing machine.

> A *universal Turing machine* U is a Turing machine (see definition 17) with a read-only input tape on which it expects an encoding $\langle M\rangle$ of a Turing machine M. After reading the input tape, U simulates the behaviour of M. If M halts, U leaves only the output of M on the tape and halts.
>
> If $\langle M\rangle$ is an encoding of M as a binary string and $M(x)$ is the partial function computed by M, then $U(\langle M\rangle\ x) = M(x)$.

We say that $d(x)$ is a description of x. The description may be a Turing machine encoding, i.e. $d(x) = \langle M \rangle$. A string x has at least one encoding, the trivial encoding, which describes x as itself. We say that $d(x)$ is the *minimal description* of x if it is the description that uses the least number of symbols.

Definition 24. Kolmogorov complexity.

> Let $d(x)$ be a minimal description of x, then
>
> $$K(x) = |d(x)|$$
>
> is the Kolmogorov complexity of x.

Thus, we consider the Kolmogorov complexity of a string to be the length of the shortest Turing machine encoding of the string. A string is algorithmically random if it is shorter than any program that can compute the string, i.e. the string x is algorithmically random if $K(x) > |x|$.

6.2 Entropy

Entropy, first introduced by Claude E. Shannon is a measure of the average information content in a message. Before we can define entropy we need to define the discrete probability distribution.

Definition 25. Discrete probability distribution.

> A discrete probability distribution on a set S is a function $D : S \to [0, 1] \subset \mathbb{R}$ so that $\sum_{x \in S} D(x) = 1$. We write $x \in X_n$ to mean that x is chosen so that $\forall z \in \{0,\ 1\}^n\ P(x = z) = X_n(z)$.

The distribution U_n is the uniform distribution where all outcomes are equally probable.

Definition 26. Entropy.

> Let X be a discrete random variable with outcomes selected from the set $x_1, x_2 \ldots x_n$. The *entropy* H of X is
>
> $$H(X) = E(I(X))$$

Where $E(X)$ is the expected value of X and $I(X)$ is the information content, or *self-information*, of X. The self-information of X is a random variable. If p is the probability mass function of X (see definition 25 above) then the entropy is

$$H(X) = \sum_{i=1}^{n} p(x_i) I(x_i) = -\sum_{i=1}^{n} p(x_i) \log_b p(x_i)$$

The base b of the logarithm is commonly set to two and we measure information or uncertainty in bits.

Claude Elwood Shannon (1916 – 2001) was an American electronics engineer and mathematician. He is considered the father of information theory. Shannon was born in Petoskey, Michigan. His father was a businessman and his mother was a language teacher. Shannon grew up in Gaylord, Michigan, where he graduated from the Gaylord High school in 1932. Then he enrolled at the University of Michigan from which he graduated in 1936 and continued his studies at the Massachusetts Institute of Technology. In his master's thesis Shannon proved that relays could be used to solve Boolean algebra problems. This is the basic concept that forms the foundation for all digital computers.

Shannon went to the Cold Spring Harbor laboratory for his Ph. D. work and presented his dissertation in 1940. Later that same year Shannon became a national research fellow at the Institute for Advanced studies at Princeton. During the war Shannon worked at Bell Labs and in 1943 he met Alan Turing who was in Washington to share the cryptanalytic methods he developed at Bletchley park. They met daily in the cafeteria and shared ideas.

At the end of the war Shannon prepared a classified paper for Bell Labs, "A Mathematical Theory of Cryptosystems", of which a declassified version were later published in 1949. In a footnote in the classified paper Shannon stated that his intentions were to develop some of the results in a future memorandum on the transmission of information.

The promised paper appeared in the Bell Systems Technical Journal in 1948, this was Shannon's landmark paper "A mathematical theory of communication", and the foundation of the field of Information Theory. In 1956 Shannon returned to MIT to hold an endowed chair. Shannon worked at MIT until 1978.

There are currently five statues of Shannon, one at the University of Michigan, one at MIT, one in Gaylord, Michigan, one at the University of

California, San Diego and the fifth at Bell Labs. Shannon is considered to be one of the greatest scientists of the 20:th century, his communication theory (now information theory) provided the foundation for the digital revolution.

Shannon himself did not know. He was struck by Alzheimer's disease. His wife wrote in his obituary that "he would have been bemused", referring to the revolution he had started.

Consider a discrete random variable X with a set of n equally proble outcomes, $x_i : i = 1 \ldots n$ and $p(x_i) = \frac{1}{n}$. We say that the *uncertainty* of X is $u = \log_b(n)$ Taking the logarithm of the number of possible outcomes means the uncertainty is additive, i.e. if we have X as above and Y, another discrete random variable with a set of m equiprobable outcomes, $\{y_i : i = 1 \ldots m\}$ then we have the uncertainty $u = \log_b(nm) = \log_b(n) + \log_b(m)$.

Since the outcomes of X are equiprobable, we can instead write, for the uncertainty of X, $u = \log_b(\frac{1}{p(x_i)} = -\log_b(p(x_i))$ for all $i = 1 \ldots n$. If the outcomes are not uniformly distributed we define the *surprisal*, $u_i = -\log_b(p(x_i))$ and the average uncertainty becomes

$$u = \sum_{i=1}^{n} p(x_i) u_i = -\sum_{i=1}^{n} p(x_i) \log_b(p(x_i))$$

which is our definition of entropy $H(X)$.

Consider flipping a coin. If the probability of the coin landing heads up is p then the entropy of a coin toss is

$$H(C) = -(p \log_2(p) + (1-p) \log_2(1-p))$$

If $p = 0$ then we have $H(C) = 0$ and if $p = 1$ then we also have $H(C) = 0$, that is, if the outcome is known beforehand (i.e. always heads or always tails) the the entropy is zero. We also see that the entropy is maximized if the coin is fair, i.e. if $p = \frac{1}{2}$.

Consider a source of data that outputs a sequence of symbols. The *entropy rate* is the number of bits per symbol that are required to encode the data emitted from the source. Assuming the symbols emitted from the data source are represented as b bit words then we can define the entropy rate as $J = \frac{H(X)}{b}$, where X is a random variable representing our data source. More formally, the entropy rate of a stochastic process X is defined as

$$H(X) = \lim_{n \to \infty} \frac{1}{n} H(X_1, X_2 \ldots X_n)$$

The stochastic process X is a set $\{X_i : i \in S\}$ where each X_i is a random variable.

We also introduce the concept of minimum entropy. Minimum entropy is always less than or equal to the Shannon entropy. They are equal when the random variable measured is uniformly distributed.

Definition 27. Minimum entropy.

> The minimum entropy of a random variable X with the set of outcomes $\{x_i : i = 1 \dots n\}$ and the probability mass function $p(x_i)$ is
>
> $$\begin{aligned} H_\infty(X) &= \min_{i=1}^{n}(-\log_b p(x_i)) = -(\max_{i=1}^{n} \log_b p(x_i)) \\ &= -\log_b \max_{i=1}^{n} p(x_i) \end{aligned}$$

Minimum entropy is a measure of the worst case amount of randomness in a random variable and is important in our reasoning about random number generation.

6.3 Harvesting entropy

We know from the previous section that information is basically equivalent to uncertainty. We are looking for uncertainty, or randomness. Suppose we have a source X that outputs w bit words. The entropy rate of the bitsource X is $J_X = H(X)/w$. Assume J_X is less than, or even much less than w. To harvest w bits of randomness we will need at least $\frac{w}{J_X}$ w bit words from the entropy source X.

Say that we have $k = \frac{w}{J_X}$ w bit words from our source of entropy X, i.e. $E_X = \{x_1,\ x_2 \dots x_k\}$. We know that the set E_X contain at least w bits of entropy among its $k \cdot w$ bits. Unfortunately it usually difficult to directly extract the w good bits from the $k \cdot w - w$ bad bits.

First we describe two techniques to *de-skew* a bitsource. The first technique is based on parity of a bitsequence, the second is due to von Neumann and equalizes the bias by forming pairs of bits, discarding some pairs and mapping others to the zero or one output bit.

The *parity* of a sequence of bits is one if the sequence contains an odd number of bits and zero if the sequence contains an even number of bits.

Assume we have a source of bits Y, much like the coin flip in section 6.2, where the bias is e. If we draw one bit y from Y, the probability that it is a one is $P(y = 1) = 0.5 + e = p$, and the probability that is it a zero is $P(y = 0) = 0.5 - e = q$. If we draw two bits $[y_1 y_0]$ the probability of the parity being one is

$$
\begin{aligned}
&P(y_1 = 0) \cdot P(y_0 = 1) + P(y_1 = 1) \cdot P(y_0 = 0) = \\
&pq + pq = 2(0.5 + e)(0.5 - e) = 2(0.5^2 - 0.5e + 0.5e - e^2) = \\
&0.5 - 2e^2
\end{aligned}
$$

and the probability that the parity is zero is

$$
\begin{aligned}
&P(y_1 = 0) \cdot P(y_0 = 0) + P(y_1 = 1) \cdot P(y_0 = 1) = \\
&qq + pp = (0.5 - e)(0.5 - e) + (0.5 + e)(0.5 + e) = \\
&0.5 + 2e^2
\end{aligned}
$$

Since $e \leq 0.5$ we see that $2e^2 \leq e$ and we have that the parity of two bits from our bitsource Y has less bias that the individual bits.

For a bitsequence of length n drawn from Y, i.e. the sequence $[y_{n-1}\ y_{n-2}\ \ldots y_1\ y_0]$ the probabilities that the parity is one or zero is the sum of the odd or even terms of the binomial expansion of $(p + q)^n$. These sums are

$$\frac{1}{2}\big((p + q)^n + (p - q)^n\big)$$

and

$$\frac{1}{2}\big((p + q)^n - (p - q)^n\big)$$

Since we have $p = 0.5 + e$ and $q = 0.5 - e$ we have $p + q = 1$ and $p - q = 2e$, thus the expressions above becomes

$$\frac{1}{2}\big(1 + (2e)^n\big) = \frac{1}{2} + 2^{n-1}e^n$$

and

$$\frac{1}{2}\big(1 - (2e)^n\big) = \frac{1}{2} - 2^{n-1}e^n$$

Which one of the expressions above that corresponds to the parity being one or the parity being zero depends on whether n is odd or even.

In any case, we see that by using a sufficiently long sequence of bits we can make the bias of the parity be arbitrary close to 0.5. That is, we say, for a probability ϵ that

$$\frac{1}{2} + 2^{n-1}e^n < \frac{1}{2} + \epsilon$$

which gives

$$n > \frac{log 2\epsilon}{log 2e}$$

so for sufficiently large n we can make the probabilities of the parity be epsilon close to unbiased.

If we instead draw pairs of bits from our source Y, discard pairs of zeroes or pairs of ones and interpret 01 as a 0 and 10 as a 1 the probability of a one or a zero will be

$$P(01) = qp = (0.5 - e)(0.5 + e) = 0.25 - e^2$$
$$P(10) = pq = (0.50e)(0.5 - e) = 0.25 - e^2$$

The bias will be eliminated. However, we will not be able to determine how many bits to draw from Y, since more than half of the drawn bits will be discarded. The expected number of bits drawn from Y to produce n bits is

$$\frac{2n}{0.5 - 2e^2}$$

This technique requires that each bit from Y has the same probability of being 0 or 1 as any other bit in the stream, i.e. that the bias is stationary, and also that the bits are uncorrelated.

It is also possible to use compression to de-skew a bitsource. Theoretically an optimal compression would leave only the good bits, i.e. the actual number of bits of information (uncertainty) in a sequence. In practice, though, general compression algorithms produce output that consist of more than the desired bits of uncertainty, for example preambles, framing, etc.

If we do not have a source of true randomness we will have to rely on one or, preferably, more weaker sources of entropy. A good strategy is to gather entropy from as many sources as possible and combine them to provide the strongest possible output. To combine, or *mix*, entropy sources we need a mixing function. A mixing function is a function that

combines its inputs to produce an output where each output bit is a different, complex, non-linear function of all the input bits. A trivial mixing function is the exclusive-or operation. We have the probabilities,

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Table 6.1: Exclusive-or.

$p_A = 0.5 + e_A$ and $p_B = 0.5 + e_B$ that a source outputs a one and the probabilities $q_A = 0.5 - e_A$ and $q_B = 0.5 - e_B$ that a source outputs a zero. The probability of the result of the exclusive-or being a one is then $p_X = p_A \cdot q_B + q_A \cdot p_B$. Expanding p_X we get

$$\begin{aligned} p_X &= (0.5 + e_A)(0.5 - e_B) + (0.5 - e_A)(0.5 + e_B) \\ &= 0.5 - 2e_A e_B \end{aligned}$$

and we have the bias of the output of the exclusive or of two independent inputs as $2e_A e_B$. Since e_A and e_B is always less than or equal to 1/2 we see that the bias is improved if e_A or e_B is less than 1/2.

Stronger mixing functions are hashing functions, such as SHA* and MD*. From an input of arbitrary size they produce a short fixed-length output which is a mix of all the input bits. The MD* family of hashing functions produce a 128-bit hash, SHA-1 produce 160 bits and other SHA hashes produce up to 512 bits of output. Further, AES and DES can also be used to mix entropy. AES, for example, takes 384 bits of input, 128 bits of data and 256 bits of key, and produces 128 bits of output. Each bit of output is dependent on a complex, non-linear, function of all input bits.

Aqcuiring, or harvesting, entropy is a highly system dependent function. In some cases dedicated hardware is available to produce truly random output. However, usually no such hardware is available and entropy must be gathered from other sources.

Some computer systems have high resolution timers, for instance the Intel architecture provides model specific registers that include a time stamp counter that can be read using an assembler instruction (`RDTSC`)

[12]. Other systems provide similar time stamp functionality or microsecond resolution timers.

Some sources of entropy are: audio or video devices, other device activity, temperature measurements, interrupt timing, and mouse movements or keystroke timings.

Many systems have inputs that digitize data from a real world analog source, such as from a microphone or video camera. The noise from a microphone port with no source connected or a from a video camera with the lens cap on is essentially thermal noise. If the gain of such an unconnected input is increased so that the noise is digitised it may provide a reasonable source of entropy. The digitized input will need to be de-skewed and the status of the input port must be checked so that it is unconnected and in working order.

In [15] and [13] the authors describe a technique to harvest entropy from disk activity. The basic idea is that air turbulence inside hard disk drives randomly disturb the timings of successive reads of the same block from a hard disk. Thus by processing timings of such reads it is possible to generate entropy. It is also possible to extract entropy from timing of network activity, for instance the delay between packets seen on an ethernet.

Some systems have on board support to measure temperature to control the fan used for cooling. If the temperature sensor outputs a high-precision reading the least significant bits of the difference between two such readings may be a source of entropy. Hardware interrupts may also provide a source of entropy by taking high resolution timing measurements between a selected set of interrupts.

One of the simplest methods to implement for entropy harvesting is timing between keystrokes. By making sure that character input is unbuffered the timing between user keystrokes provide a simple source of entropy.

6.4 Random number generator

A *random number generator* is a device that generates a sequence of numbers (bits) that are random. Random numbers can be generated based on the outcome of a physical process, for instance rolling a die, measuring thermal noise, atmospheric noise, or something similar.

True randomness is hard to find. Many physical sources of randomness are biased. And generally, if a physical source of randomness is

available, it often does not produce a sufficient number of random bits per second.

A *pseudo-random number generator* is an algorithm that generate (compute) numbers (bits) with good random properties.

We introduce yet another type of Turing machine, the probabalistic Turing machine. The probabalistic Turing machine is a non-deterministic Turing machine which randomly selects its transition. The difference between the two types of Turing machines is that the non-deterministic Turing machine presented in section 3.4 always makes the correct (best) choice in any given situation, and the probabalistic Turing machine simply makes a choice randomly.

Definition 28. Probabalistic Turing machine .

> A *probabalistic Turing machine* is a non-deterministic Turing machine which at each point randomly, according to some probability distribution, chooses between its available transitions.

If the available transitions are equally probable we can implement a probabalistic Turing machine using a deterministic Turing machine with an extra tape of random bits (the random tape).

The indistinguishable concept defined below is the base for our definition of randomness, or pesudo-randomness. Assume we have a probabalistic Turing machine A that, given a string x, decides the distribution of x. We say that, if the difference in probability that the probabalistic Turing machine A decides between distributions X_n and Y_n is less than $\frac{1}{Q(n)}$ where $Q(n)$ is a polynomial, then X_n and Y_n are indistinguishable.

Definition 29. Polynomial time indistinguishable.

> Let X_n and Y_n be probability distributions on $\{0,\ 1\}^n$. I.e. if $x \in X_n$ then $x \in \{0,\ 1\}^n$ and x is selected according to the distribution X_n. $\{X_n\}$ is *polynomial time indistinguishable* from $\{Y_n\}$ if, for all probabalistic Turing machines (A) and for all polynomials (Q), there exists an n_0 such that for all $n > n_0$ we have
>
> $$|P_{x \in X_n}(A(x) = 1) - P_{x \in Y_n}(A(x) = 1)| < \frac{1}{Q(n)}$$

That is, for sufficiently long strings there is no probabalistic Turing machine that can decide whether the string was sampled from X_n or Y_n.

Definition 30. Pseudo random.

> We say that $\{X_n\}$ is *pseudo random* if it is polynomial time indistinguishable from $\{U_n\}$.

We can now define the pseudo random number generator as a program (description) that, given a shorter sequence of bits, outputs a longer sequence of bits which is pseudo-random.

Definition 31. Pseudo random generator.

> A polynomial time deterministic program
>
> $$G : \{0,\ 1\}^k \to \{0,\ 1\}^l$$
>
> is a pseudo random generator if
>
> 1. $l > k$
> 2. $\{G_l\}_l$ is pseudo random.
>
> G_l is the distribution on $\{0,\ 1\}^l$ where $g \in G_l$ is obtained by setting $g = G(x)$ and x is selected from U_k ($x \in U_k$).
>
> Thus, G_l and U_l are polynomial time indistinguishable.

We call the sequence $x \in U_k$ the *seed*. The pseudo-random generator expands the seed into the sequence $\{G_l\}$. It is possible to prove that pseudo-random generators as defined in definition 31 exists. The proof is not presented here, see instead [7]. For practical applications we must select the seed in such a way that is is unpredictable, i.e. we are interested in a seed with the highest possible entropy.

Definition 32. Next bit test.

> A *next bit test* is a statistical test which takes a prefix of a sequence as input and outputs a prediction of the next bit of the sequence.
>
> We say that a pseudo-random generator *passes* the next bit test A if, for all polynomials Q there exists an integer k_0 such that for all $k > k_0$ and $n < k$ the following holds
>
> $$P_{g \in G_k}(A(g_1,\ g_2 \ldots g_n) = g_{n+1}) < \frac{1}{2} + \frac{1}{Q(k)}$$

It is possible to prove that if a pseudo-random generator G passes the next bit tests then G passes all statistical tests, see [7].

6.5 The Blum-Blum-Shub generator

The Blum-Blum-Shub pseudo random number generator is a pseudo random number generator that passes the next bit test. Blum-Blum-Shub is *cryptographically secure*. It is computatinally intensive and is not appropriate for general use. It relies on the intractability of the integer factorization problem for its (provable) security. The Blum-Blum-Shub generator is as secure as RSA encryption.

The Blum-Blum-Shub equation is

$$x_{n+1} = x_n^2 \bmod M$$

where M is a product of two large primes p and q. The output of the Blum-Blum-Shub generator is the least significant bit of x_{n+1}. The two primes p and q must be congruent to $3 \pmod 4$.

Algorithm 11. Blum-Blum-Shub.

The Blum-Blum-Shub algorithm computes a cryptographically secure random bit.

In: The seed s.

Out: A random bit z_n.

1: generate p and q, each congruent to $3 \pmod 4$
2: $M = p \cdot q$
3: $x_0 = s^2 \pmod M$
4: $x_n = x_{n-1}^2 \pmod M$
5: $z_n = x_n \bmod 2$

The output is a sequence $[z_1\ z_2\ z_3 \ldots]$, i.e. the parity of each element in the sequence $[x_1,\ x_2,\ x_3 \ldots]$ given by the equation $x_n = x_{n-1}^2 \pmod M$.

For a thorough description of the Blum-Blum-Shub generator and a proof of its security see Junods paper [16]. We will continue with a presentation of its iplementation. Our implementation of the Blum-Blum-Shub generator consists of three routines, first the initialization in

mp_blum_blum_shub_init, then the seeding in mp_blum_blum_shub_seed and finally the generator, in mp_blum_blum_shub_rand.

The initialization essentially consists of computing the modulus M, by finding two Blum primes, p and q and multiplying them to form M. Looking at listing 6.2 we start by making sure the requested bitsize of the modulus is a multiple of eight, i.e. an even byte (line 9). If not we generate an error, line 10. Next we compute the number of digits required to hold the requested b bits, line 13 and then we allocate two temporary variables to hold p and q (lines 16 and 17).

The state that must be kept for the Blum-Blum-Shub generator is held in a global variable, mp_bbs_state, a record with two elements, a pointer to the modulus and a pointer to the current value, X_0. On lines 19 and 20 we allocate space to hold the modulus and current value and set the pointers of the state record to the allocated space.

Next, on line 22, we generate a number of small primes for use in the mp_findprime routine. The mp_smallprimes and mp_findprime routines are discussed in section 7.1 and section 7.4 of chapter 7 (listings 7.1 and 7.4).

On line 24 and line 28 we use mp_findprime to find the two Blum primes p and q. By setting the parameters lb to 3 and inc to 4 we ensure that the prime returned by mp_findprime is a Blum prime, i.e. congruent to $3 \pmod 4$.

Finally we multiply p and q to form the modulus M and assign it to our state variable mp_bbs_state.M, previously allocated. This concludes the initialization.

It is important that the random numbers provided by the argument function rnd are of good quality. Preferably the random number generator function provided as an argument to the mp_blum_blum_shub_init routine is a true random number source, i.e. that rnd is a good entropy source.

```
void
mp_blum_blum_shub_init(mp_size_t b, mp_rand_f rnd)
{

5       mp_t  P, Q, M;
        mp_size_t l;
        mp_limb_t sp[MP_BBS_SMALLPRIMES];

        if ((b % 8) != 0) {
```

```
10              MP_ERR(MP_ERR_EVENBYTE);
        }

        l = b / (8 * sizeof (mp_limb_t)) +
                (b % sizeof (mp_limb_t) ? 1 : 0);
15
        P = mp_tmp(l);
        Q = mp_tmp(l);

        mp_bbs_state.M = mp_new(l);
20      mp_bbs_state.X0 = mp_new(l);

        mp_smallprimes(sp, MP_BBS_SMALLPRIMES);

        mp_findprime(P, b / 2, MP_BBS_MILLER_RABIN_T,
25              sp, MP_BBS_SMALLPRIMES, 0x80000000,
                0x3, 4, rnd);

        mp_findprime(Q, b / 2, MP_BBS_MILLER_RABIN_T,
                sp, MP_BBS_SMALLPRIMES, 0x80000000,
30              0x3, 4, rnd);

        mp_mul(mp_bbs_state.M, P, Q);
}
```

Listing 6.2: Blum-Blum-Shub initialization

In listing 6.3 we present the a routine that seeds the Blum-Blum-Shub generator. We hold the global state of the Blum-Blum-Shub generator in the record `mp_bbs_state` and the routine `mp_blum_blum_shub_seed` simply assigns the provided seed to the member `X0` of the record holding the state. However, we first check that the seed is not equal to one (line 4). If it is then an error is generated (line 6). Finally, on line 9 we assign the seed S to `mp_bbs_state.X0` and we are done.

```
void
mp_blum_blum_shub_seed(mp_t S)
{
        if (mp_eq(S, mp_One)) {
5
                MP_ERR(MP_ERR_ARG);
```

```
        }

        mp_set(mp_bbs_state.X0, S);
}
```

Listing 6.3: Seeding the Blum-Blum-Shub pseudo random number generator

The routine presented in listing 6.4 computes a random digit using the Blum-Blum-Shub pseudo random number generator. It iterates the Blum-Blum-Shub generator to output the required number of random bits needed to fill a w-bit word.

First, on line 9 we allocate temporary space for a multiple precision number capable of holding the current value squared, then we initialize r to zero (line 11.

```
mp_limb_t
mp_blum_blum_shub_rand()
{
        mp_size_t i;
        mp_limb_t r;

        mp_t  R;

        R = mp_tmp(2 * *(mp_bbs_state.X0));

        r = 0;

        for (i = 0; i < 8 * sizeof (mp_limb_t); i++) {

                mp_sqr(R, mp_bbs_state.X0);
                mp_mod(mp_bbs_state.X0, R,
                        mp_bbs_state.M);

                r <<= 1;
                r |= mp_getbit(mp_bbs_state.X0, 0);
        }

        return r;
}
```

Listing 6.4: The Blum-Blum-Shub pseudo random number generator

Then we enter the main loop of the generator (line 15). We will iterate w times, to produce w bits. We evaluate the Blum-Blum-Shub equation, $x_n = x_{n-1}^2 \bmod M$, by first squaring the current value (line 15) and then computing the remainder (line 16). Now we have the new value (x_n) in our state variable `mp_bbs_state.X0`.

We shift r right one bit to make room for the random bit we computed in the current iteration (line 19). Next we extract the random bit from the state (`mp_bbs_state.X0`) and add it to the other random bits in R (line 20). When all w iterations are complete we have a random digit in r. On line 23 we return the random digit and our work is done.

Chapter 7

Finding prime numbers

7.1 Introduction

A *prime* number is a number that is divisible only by one and itself. A *composite* number is a number which is not prime, i.e. it is composed of prime factors (it is a product of prime factors). We know from Euclid's second lemma (theorem 7) that there are an infinite number of primes. But how do we find one?

The simplest approach to finding prime numbers is to select a number, x, and divide it with all numbers y_i less than x, i.e. $1 < y_i < x$. If any of the y_i divides x evenly, i.e. if $x = qy + r$ where $r = 0$, then x is not a prime number. If no y_i divides x evenly then x is divisible only with 1 and itself, thus x is a prime. This process we call *trial division*.

We can make significant improvements to our trial division algorithm. First note that it suffices to perform trial division up to $\sqrt{x}$, that is $1 < y_i \leq \sqrt{x}$. Since, if there is a factor of x greater than $\sqrt{x}$ then there must also be another factor of x less than $\sqrt{x}$. Thus the lesser factor will be found in the set $\{y_i \mid 1 < y_i \leq \sqrt{x}\}$.

Naive trial division is not a feasible method for finding large prime numbers. Trial division requires $O(\sqrt{x})$ divisions. If x is large, say in the order of 2^{512} we will have to perform, in the worst case 2^{256} divisions, which is approximately 10^{77} divisions. On a computer that can perform one division every nanosecond it would take approximately 140 years to check one 512-bit prime number.

A better algorithm is the *sieve of Eratosthenes*. Eratosthenes of Cyrene (c. 276 BC – c. 195 BC) was a Greek mathematician. Eratosthenes was born in Cyrene (today Libya). Erathostenes studied in Alexandria. In 236 BC he was appointed to be the third librarian of the Great Library of Alexandria.

Eratosthenes invented a system of latitude and longitude, he also calculated the circumference of the earth as well as the earth's axis tilt. The Suda, an ancient Greek historical encyclopedia (10:th century), states that he was called *Beta* by his contemporaries, since he supposedly proved himself to be the second best in the world in almost any field.

The sieve of Eratosthenes is an algorithm for finding all prime numbers up to a specified value. It is efficient for small primes.

Algorithm 12. Sieve of Eratosthenes.

In: An upper bound n.

Out: A list of prime numbers less than or equal to n.

1. Create a list of numbers from two to n.
2. Strike out all multiples of two from the list.
3. The next number m in the list that has not been stricken is a prime.
4. Strike out all multiples of m.
5. Repeat steps 3 and 4 until $m > \sqrt{n}$.
6. All numbers remaining on the list, i.e. not stricken, are prime numbers.

The space complexity of the sieve of Eratosthenes is in $O(n)$ If we are looking for prime numbers in the proximity of 2^{512} we realize that the memory requirements of the sieve of Eratosthenes disqualifies it, we would need roughly 10^{146} 1GB disks to list all the numbers from two to 2^{512}.

Clearly neither trial division nor the sieve of Eratosthenes alone are feasible algorithms for finding large prime numbers. We need to find another approach. However, trial division with a small number of primes can be used to enhance the performance of other, more subtle algorithms to find large prime numbers.

Algorithm 13 below computes the n first prime numbers by trial division. The algorithm generates a sequence $(p_0,\ p_1,\ \ldots,\ p_k)$ of prime numbers. To compute the prime p_{k+1} we start with the number $p_{k+1} = p_k + 1$ and test it for divisibility with any of the previously computed primes, i.e. with all the numbers in the sequence $(p_0,\ p_1,\ \ldots,\ p_k)$. If it is divisible by any of the p_i, $0 \leq i \leq k$ then it is composite and we increase p_{k+1}, i.e. $p_{k+1} = p_{k+1} + 1$ and repeat the trial divisbility test. When we find a number, p_{k+1}, that is not divisible with any of the previous p_i in the sequence we conclude that p_{k+1} must be a prime and we add p_{k+1} to the sequence.

The computation is repeated until there are n prime numbers in the sequence.

Algorithm 13. Finding small primes.

In: A positive integer, n, $n > 1$.

Out: A sequence $(p_0,\ p_1,\ \ldots,\ p_n) = (2,\ 3,\ 5,\ \ldots)$ of the n first primes.

1: $p_0 = 2$
2: **for** $i = 1 \ldots n$ **do**
3: $\quad p_i = p_{i-1}$
4: $\quad$ **repeat**
5: $\quad\quad p_i = p_i + 1$
6: $\quad\quad f =$ FALSE
7: $\quad\quad j = 0$
8: $\quad\quad$ **while** $j < i$ and $f \neq$ TRUE **do**
9: $\quad\quad\quad$ **if** $p_j | p_i$ **then**
10: $\quad\quad\quad\quad f =$ TRUE
11: $\quad\quad\quad$ **end if**
12: $\quad\quad\quad j = j + 1$
13: $\quad\quad$ **end while**
14: $\quad$ **until** $f =$ FALSE
15: **end for**

In listing 7.1 we present an implementation of algorithm 13. The sequence is generated in the n long array pointed to by p. We start computation with the minimal sequence, (2), i.e. we set p_0 to 2 on line 10. Then we iterate lines 12 to 27 from 1 to n to compute the $n-1$ next primes.

```
void
mp_smallprimes(mp_limb_t *p, mp_size_t n)
{
        mp_limb_t f;
        mp_size_t i, j;

        if (n < 1)
                MP_ERR(MP_ERR_ARG);

        p[0] = 2;

        for (i = 1; i < n; i++) {

                p[i] = p[i - 1];

                do {
                        p[i]++;
                        f = 0;
                        for (j = 0; j < i && f == 0; j++) {

                                if (0 == (p[i] % p[j])) {

                                        f = 1;
                                }
                        }
                } while (f);
        }
}
```

Listing 7.1: Small primes

On line 14 we initialize p_i to p_{i-1} and then we enter the trial division loop on line 16. First p_i is increased by one to form the first candidate (line 17). The flag f is used to indicate if p_i has any factors among the previously generated primes and is initialized to zero on line 18. Then we continue with the trial division, lines 19 to 27. We iterate from 0 to i and we will exit the loop if we have found a factor (if f is nonzero, line 19.

If the remainder of p_i when divided by p_j, $0 \leq j \leq i$ is zero then the current p_j is a factor of p_i and the current p_i is not a prime (line 21

and 23). If so then we exit the loop and add one to p_i to form the new candidate prime (17).

Eventually we encounter a p_i that does not have a factor among the previously generated primes and we exit the loop (line 26), now with p_i added to the sequence of primes. The process is repeated until all the n requested primes have been generated.

The implementation in listing 7.1 is designed to find *small* primes, thus it uses the native division (remainder) operator of the computer. The maximum prime number that can be computed by the routine is $p \leq 2^w$ where w is the wordsize of the computer.

To test a large probable prime against a sequence of small prime numbers we use algorithm 14 below. It takes as input an m long sequence of prime numbers p and a number to be tested for divisibility, n. The algorithm iteratively divides n with each of the p_i, $0 \leq i \leq m$. If any of the p_i divides n, i.e. the remainder of the division is zero, then m can not be prime. If no p_i divides n then n may be prime. We will have to test it more thoroughly.

Algorithm 14. Trial division by m small primes.

In: A positive integer n and a list of prime numbers $p = (p_0,\ p_1,\ \ldots p_m)$.

Out: An answer to the question whether n is divisible by any p_i, $0 \leq i \leq m$.

1: **for** $i = 0 \ldots m$ **do**
2: $k = p_i$
3: Compute q and r in $n = qp_i + r$
4: **if** $r = 0$ **then**
5: **return** DIVISIBLE
6: **end if**
7: **end for**
8: **return** NOT-DIVISIBLE

An implementation of algorithm 14 is presented in listing 7.2. The routine takes as input a list of small primes in p and a multiple precision number N that is to be tested for divisibility by any of the p_i in the m-element list.

```
mp_limb_t
mp_trialdivision(mp_t N, mp_limb_t *p, mp_size_t m) {
```

```
        mp_size_t i;
        mp_limb_t P[2] = { 1, 0 };
        mp_t  Q, R;
7
        Q = mp_new(*N);
        R = mp_new(*N);

        for (i = 0; i < m; i++) {
12
                P[1] = p[i];

                mp_div(Q, R, N, P);

17              if (mp_eqz(R)) {

                        mp_free(Q);
                        mp_free(R);

22                      return 0;
                }
        }

        mp_free(Q);
27      mp_free(R);

        return 1;
}
```

Listing 7.2: Trial division

The loop, lines 11 to 24, performs m divisions of N. On line 13 the multiprecision number P is constructed from the small prime p_i. The trial division is performed on line 15. If the remainder R is equal to zero (line 17) then N can not be a prime and we exit the routine. If the remainder is not equal to zero we continue testing N against the rest of the primes in the list p. If neither of them divides N, N passes our trial division test and we declare N as not being divisible by any of the p_i, line 29.

7.2 Prime number theorem

The prime number theorem was first proven in 1896 by the French mathematician Jacques Salomon Hadamard (1865 – 1963) and the Belgian mathematician Charles-Jean Étienne Gustave Nicolas, Baron de la Vallée Poussin (1866 – 1962). They discovered the proof independently from each other. The proof builds on complex analysis and the characteristics of the Riemann zeta function.

Theorem 22. Prime number theorem.

> Let $\pi(n)$ be the number of primes less than or equal to n. The limit of the quotient of $\pi(n)$ and $\frac{n}{\ln n}$ approaches 1 as n approaches infinity, i.e.
>
> $$\lim_{n\to\infty} \frac{\pi(n)}{\frac{n}{\ln n}} = 1$$
>
> This is the *asymptotic law of the distribution of prime numbers* which we restate in asymptotic notation:
>
> $$\pi(n) \sim \frac{n}{\ln n}$$

Proof. The proof of the prime number theorem is complicated and not presented in this text.

In 1796 Adrien-Marie Legendre (1752 – 1833), a french mathematician, conjectured that $\pi(x)$ is approximately $\frac{x}{ln(x)-B}$ where $B = 1.08...$. He based his conjecture on tables of prime numbers by Anton Felkel, an Austrian mathematician born in 1740 and Baron Jurij Bartolomej Vega (1754 – 1802) a Slovenian mathematician. Carl Friedrich Gauss (see section 2.5 also provided an approximation based on the logarithmic integral $li(x) = \int_0^x \frac{dt}{ln(t)}$, that $\pi(x) \sim li(x)$. As usual when his results was not complete and above criticism, Gauss never published his approximation.

The Russian mathematician Pafnuty Lvovich Chebyshev (1821 – 1894) set out to prove the asymptotic law of distribution of prime numbers. Chebyshev proved a weaker form of the prime number theorem. That is, he proved that, if the limit of $\frac{\pi(x)}{x/ln(x)}$ as $x \to \infty$ exists then it will

be equal to one. Chebyshev also proved that the ratio is bounded above and below by two constants nearly one for all x.

The most significant paper written about the distribution of prime numbers written by Riemann and published in 1859 titled “On the number of primes less than a given magnitude” [3].

Georg Friedrich Bernhard Riemann (1826 – 1866) was a prominent German mathematician, his main contributions were to the field of analysis and differential geometry, except for his single paper on number theory.

Riemann was born in Breselenz, near Danneberg, in what is now Germany. Riemann was the second of six children. Riemanns father was a pastor, his mother died early. He was prone to nervous breakdowns and was very shy. He was mathematically gifted as a child.

He studied the bible intensively but ofted drifted into mathematics. He amazed his teachers with his mathematical ability. In 1846 he started studies in philology and theology to become a priest, as his father. Later, in the spring of the same year, his father had raised enough money to send Riemann to the University of Göttingen, to study mathematics, abandoning the plans to become a priest. At the University of Göttingen he met Carl Friedrich Gauss and attended his lectures. Next year, in 1847, he moved to Berlin to attend lectures by Jacobi, Dirichlet, Steiner and Eisenstein. After two years he returned to Göttingen, in 1849.

Riemann held his first lectures in 1854. By founding the field of Riemannian geometry he enabled Einstein’s to develop his general relativity theory. In 1857 an attempt was made to promote Riemann to extraordinary professor at the University of Göttingen. The attempt failed but Riemann was finally granted a regular salary. After Dirichlet’s death in 1859 Riemann was promoted to head of the mathematics departement at Göttingen. He married in 1862, to Elise Koch who later gave birth to a daughter. Riemann died of tuberculosis in 1862 during a journey to Italy. He was buried in the cemetary in Biganzolo, Verbania.

In his paper, possibly the single most important paper on the distribution of prime numbers, Riemann introduced revolutionary ideas. The most important one was that the distribution of the prime numbers seem to be connected to the location of the zeroes of the analytically extended Riemann zeta function of a complex variable.

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} = \frac{1}{1^s} + \frac{1}{2^s} + \frac{1}{3^s} + \dots$$

Riemann introduced complex analysis to the study of $\pi(x)$. Hadamard and de la Vallée Poussin extended Riemanns ideas and successfully proved the prime number theorem in 1896. Later, during the 20:th century, several other proofs were found, among them the elementary proofs of Atle Selberg and Paul Erdös in 1949.

The prime number theorem roughly describes how the prime numbers are distributed. An implication of the prime number theorem is that if we select a large number n, then the probability that n is prime is approximately $1/ln(n)$. Stated differently, the average distance between prime numbers in the neighborhood of n is approximately $ln(n)$. Another implication is that the n:th prime number, p_n is approximately $nln(n)$. The approximation becomes better (the error smaller) when n approaches infinity.

Now, we are interested in finding prime numbers in the proximity of 2^{512}. The prime number theorem gives that around 2^{512} approximately one in 355 (i.e. $ln(2^{512})$) numbers are prime numbers. If we have an algorithm that can check (or verify) if a number is prime we could start with a random number n around 2^{512} and sequentially check n, $n+1$, $n+2$ and so on. We would have to check, on average, 178 candidates.

7.3 Probabalistic primality tests

Let n be an odd positive integer. The set $W(n) \subset \mathbb{Z}_n$ is defined by

1. For an $a \in \mathbb{Z}_n$ it is possible to check in deterministic polynomial time whether $a \in W(n)$.
2. If n is prime $W(n) = \emptyset$.
3. If n is composite $|W(n)| \geq \frac{n}{2}$.

If n is composite the elements of $W(n)$ are called *witnesses* to the compositenes of n and the elements of $L(n) = \mathbb{Z}_n - W(n)$ are called *liars*.

We want to test the number n. We select an a, $a \in \mathbb{Z}_n$ at random and we check if $a \in W(n)$. If $a \in W(n)$ then n is composite, if $a \notin W(n)$ then n may be prime. An integer n that is believed to be prime based on a probabalistic primality test is a *probable prime*.

Theorem 23. Square roots of unity.

Let p be a prime number. For $x \in \mathbb{Z}_n$ and $x \not\equiv 1 \pmod{p}$ and $x \not\equiv -1 \pmod{p}$ we have

$$x^2 \not\equiv 0 \pmod{p}$$

That is there are no nontrivial square roots of $1 \pmod{p}$.

Proof. Assume the contrary, that x is a nontrivial square root of 1 $\pmod{p}$, i.e. that $x \not\equiv \pm 1 \pmod{p}$. Then we have

$$\begin{aligned} x^2 &\equiv 1 \pmod{p} \\ (x-1)(x+1) &\equiv 0 \pmod{p} \end{aligned}$$

Now, x is a nontrivial square root so x is not $\pm 1 \pmod{p}$. Further, both $x-1$ and $x+1$ must be coprime to p (p is a prime). Thus neither $x-1$ nor $x+1$ divides p. Since, by Euclid's lemma, theorem 6, we know that if a prime does not divide either of two factors then it can not divide the product of the two factors. We have a contradiction. We conclude that

$$(x-1)(x+1) \not\equiv 0 \pmod{p}$$

and x can not be nontrivial.

Now, we have Fermat's little theorem, theorem 15, which states that

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

if p is prime and p is coprime to a.

So, let n be an odd prime, then $n-1$ is even and we can rewrite $n-1$ as $2^s r$, where $s, r \in \mathbb{Z}$ and r is odd. For each a in $\mathbb{Z}_n$ we have that either

$$a^r \equiv 1 \pmod{n} \tag{7.1}$$

or

$$(a^r)^{2^t} \equiv -1 \pmod{n}, \text{ for some } t,\ 0 \le t \le s-1 \tag{7.2}$$

by theorem 23 above (substituting a^r for x).

If we choose an integer $a \in \mathbb{Z}_n$ and compute the sequence

$$a^r \bmod n,\ (a^r)^2 \bmod n,\ (a^r)^{2^2} \bmod n,\ \ldots, (a^r)^{2^s} \bmod n$$

we have, by Fermat's little theorem, that the sequence must end with 1 or else n is not a prime.

Assuming n is a prime, if we start with $a^{n-1} = a^{2^s r}$ and repeatedly take the square root of a^{n-1}, i.e. our sequence above, then the result should be either 1 or -1. If we have -1 then equation 7.2 holds. If we exhaust every power of two without satisfying equation 7.2 then the first equation (equation 7.1) must hold.

Thus, we must have that the elements of the sequence must be either 1 or -1 if n is prime. If any of the elements of the sequence is not equal to ± 1 then n is not a prime, and n is composite. We say that a is a witness to the compositeness of n.

Miller-Rabin primality test

The Miller-Rabin primality test is based on the reasoning above. I.e. if we can find an a (a *base*) such that

$$a^r \not\equiv 1 \pmod{n}$$

and

$$a^{2^t r} \not\equiv -1 \pmod{n}\ \forall t,\ 0 \leq t \leq s-1$$

then a is a witness to n being composite. If not then n is a probable prime.

Note that the test only conclusively answers if n is composite. For some inputs the test will erroneously declare n as not being composite. In fact it can be shown that for any odd integer n the Miller-Rabin test will fail with a probability less than $\frac{1}{4}$.

If we test n once, then the probability of the composite n being declared a prime is less than $\frac{1}{4}$. If we repeat the test with a different a the probability of n being composite even though the test, twice, declared n to be prime is $(\frac{1}{4})^2 = \frac{1}{16}$. Thus by rerunning the test t times we can establish with arbitrary probability that n is prime, i.e. we have n being a prime with the probability $1 - (\frac{1}{4})^t$.

Summarizing the reasoning above, we have the he Miller-Rabin probabalistic primality test algorithm as below in algorithm 15.

Algorithm 15. Miller-Rabin probabalistic primality test.

In: An odd integer $n \geq 3$ and a parameter $t \geq 1$.

Out: An answer "COMPOSITE" or "PRIME".

1: Write $n - 1 = 2^s r$ such that r is odd.
2: **for** $i = 1 \dots t$ **do**
3: Choose a random integer a, $2 \leq a \leq n - 2$.
4: $y = a^r \bmod n$.
5: **if** $y \neq 1$ and $y \neq n - 1$ **then**
6: $j = 1$
7: **while** $j \leq s - 1$ and $y \neq n - 1$ **do**
8: $y = y^2 \bmod n$
9: **if** $y = 1$ **then**
10: **return** COMPOSITE
11: **end if**
12: $j = j + 1$
13: **end while**
14: **if** $y \neq n - 1$ **then**
15: **return** COMPOSITE
16: **end if**
17: **end if**
18: **end for**
19: **return** PRIME

In listing 7.3 we present an implementation of the Miller-Rabin probabalistic primality test. In lines 9 to 25 we initialize and set up for the main computation in lines 27 to 67.

First, on line 9 we check so that the number to be tested for primality, P, is odd. If not we generate an error. On lines 12 to 18 we allocate space for working variables. The size of our working variables is twice the size of P to make room for N^2 which we need to compute as part of the algorithm (on line 51). On line 20 and 21 we initialize N to P and find the length of P in N_L. On lines 23 to 25 we set R and N_1 to $N - 1$. N_1 will be used later in comparisons as part of the test.

Now we compute R and s in $N - 1 = 2^s R$, lines 27 to 31. R is already set to $N - 1$ so we set s to zero. Then we shift R right, i.e. we divide R by two as long as R is even. When the `do-while` loop has terminated we will have s and R in $N - 1 = 2^s R$.

```
mp_limb_t
mp_miller_rabin(mp_t P, mp_size_t t, mp_rand_f rnd)
{
        mp_t  N, N_1, R, A, Y, T, U;
```

```
5       mp_size_t NL;
        mp_limb_t s;
        mp_limb_t i, j;

        if (mp_even(P))
10              MP_ERR(MP_ERR_ODD);

        N = mp_new(2 * *P);
        N_1 = mp_new(*N);
        R = mp_new(*N);
15      A = mp_new(*N);
        Y = mp_new(*N);
        T = mp_new(*N);
        U = mp_new(*N);

20      mp_set(N, P);
        NL = mp_len(N);

        mp_set(R, N);
        mp_sub(R, R, mp_One);
25      mp_set(N_1, N);

        s = 0;
        do {
                s++;
30              (void)mp_sr(R, R, 1);
        } while (mp_even(R));

        for (i = 1; i <= t; i++) {

35              do {
                        mp_rand(A, NL * 8 * sizeof (mp_limb_t),
                                rnd);
                } while (!mp_gt(A, mp_One) || !mp_lt(A, N_1));

40              mp_set(T, A);
                mp_set(U, R);
                mp_expmod(Y, T, U, N);

                if (mp_eq(Y, mp_One) || mp_eq(Y, N_1)) {
```

```
45
                        continue;
                }

                for (j = 1; j <= s - 1; j++) {
50
                        mp_sqr(U, Y);
                        mp_mod(T, U, N);
                        mp_set(Y, T);

55                      if (mp_eq(Y, mp_One)) {

                                r = 0; /* composite */
                                goto exit;
                        }
60
                        if (mp_eq(Y, N_1)) {

                                break;
                        }
65              }

                if (mp_neq(Y, N_1)) {

                        r = 0; /* composite */
70                      goto exit;
                }

        }

75      r = 1; /* probable prime */

exit:
        mp_free(N);
        mp_free(R);
80      mp_free(A);
        mp_free(Y);
        mp_free(N_1);
        mp_free(T);
        mp_free(U);
```

```
85
        return r;
}
```

Listing 7.3: Miller-Rabin probabalistic primality test

We continue with the main part of the test. We repeat the test t times, line 33. First we randomly select A so that $1 < A < N - 1$, lines 35 to 38. Then we compute $Y = A^R \bmod N$, lines 40 to 42. If Y is equal to one or equal to $N - 1$ then we have that $A^R \equiv 1 \pmod N$ and by equation 7.1 P may be prime so we continue with the next A (line 46). If Y is not equal to one and not equal to $N - 1$ we continue checking the other case, i.e. equation 7.2.

We generate each square in the sequence $(A^R)^{2^j} \bmod N$ where $1 \leq j \leq s-1$ in the `for`-loop on lines 49 to 65. On lines 51 to 53 we compute $Y = Y^2 \bmod N$. On line 55 we test if $Y = 1 \pmod N$, if so then P is not a prime and we will return *composite*, else we increase j and try again.

If, at the end of any iteration, we find that Y is not equal to $N - 1$ (line 67), i.e. if $Y \not\equiv -1 \pmod N$, then P can not be a prime and N fails the test. We exit the routine indicating that N is a composite number (lines 69 and 70).

If we do not encounter any evidence for P being composite then P may be prime and we return *prime*, line 75.

In both cases, we clean up our allocated working variables, lines 78 to 84 and return a value indicating the result for N on line 86.

7.4 Generating large prime numbers

The Miller-Rabin probabalistic primality test gives us a method to test if a given number is prime. That is the Miller-Rabin algorithm only allows us to answer the question "Is the number n prime?". Prime number *testing* differs from prime number *generation*.

We know from the Prime number theorem, theorem 22, that around the number x, the probability of a nearby number being prime is approximately $1/ln(x)$. As we discussed at the end of section 7.2, around 2^{512} approximately one number in 355 is prime. This fact, together with the Miller-Rabin algorithm form the basis of our strategy for generation of large prime numbers.

If we are interested in a prime in the region of 2^s we select an s bit random number, n, making sure that n is odd and that the s:th bit is set (so that indeed it is in the region of 2^s). We then execute the Miller-Rabin test on the number n t times. If it passes then we know that n is prime with a probability of $1 - (\frac{1}{4})^t$.

We can improve the performance of our method by being more selective with our candidate, the number n. By performing trial division (algorithm 14) on n with a number of small primes we improve the quality of our prime candidates and thus the efficiency of our large prime number generation.

An implementation of large prime number generation, that uses trial division and the Miller-Rabin probabalistic primality test is presented in listing 7.4.

The routine generates a prime number N based on the following input; b is the desired bitwidth of N, t is the Miller-Rabin parameter, p is a list of small primes that is n long. The parameter h determines how many of the most significant bits should be set in N.

First we determine how many words N need to be to accomodate the desired b bits. That is, we compute l, the number of words required for b bits and compare it to the size of N, on lines 12 to 17. If N is not sufficiently large we generate an error. We also set up our increment in the multiple precision number I on line 20 and initialize the success flag f (line 22) to zero.

```
void
mp_findprime(mp_t N, mp_size_t b, mp_size_t t,
             mp_limb_t *p, mp_size_t n,
             mp_limb_t hb, mp_limb_t lb,
5            mp_limb_t inc, mp_rand_f rnd) {

      mp_size_t l;
      mp_size_t i, c;
      mp_limb_t I[2] = {1, 0};
10    mp_limb_t f;

      if ((b % 8 * 2) != 0)
            MP_ERR(MP_ERR_EVENBYTE);

15    l = b / (8 * w);
```

```
        if (*N < 1)
                MP_ERR(MP_ERR_RESSIZE);

20      I[1] = inc;

        f = 0;

        mp_rand(N, b, rnd);
25
        do {

                N[*N + 1 - l] |= hb;
                N[*N] |= lb;
30
                f = mp_trialdivision(N, p, n);
                if (f) {
                        f = mp_miller_rabin(N, t);
                }
35
                if (f) {
                        return;
                } else {
                        if (inc == 0) {
40                              mp_rand(N, b, rnd);
                        } else {
                                mp_add(N, N, I);
                        }

45      } while (1);
}
```

Listing 7.4: Find prime number

We start the main computation on line 24 by generating a first candidate, `mp_rand` generates a b bit random number using the pseudo random number generator specified by the function pointer `rnd`. We enter the main loop on line 26. We will iterate until we have found a probable prime, that passes both the trial division test and the Miller-Rabin test (lines 26 to 45).

The random number generated on line 24 will be a number between 0 and $2^b - 1$. We start by shaping it according to the patterns specified

in hb and lb. That is, we set the highest bits to the bitwise `OR` of the highest digit and hb and we also set the lowest bits to the bitwise `OR` of the lowest digit and lb, lines 28 and 29. Now we have the prime number candidate in N.

On line 31 we use the trial division test to make sure N is not divisible by any of the small primes in the n long list p. If it is, then **`mp_trialdivision`** will return 0 and some p_i in p is a factor of N and f will be set to zero. If not, i.e. if no p_i in p is a factor of N then f is set to one. Next, if f is set and N is not divisible by any of the small primes in p then we execute the Miller-Rabin test, on line 33. Again, f is set to one if N passes the Miller-Rabin test, if not f is set to zero.

The process is repeated until f is set, i.e. while N fails both the trial division test and the Miller-Rabin test. If it passes both, then N is a probable prime and we are done. We detect this on line 36 and return with N as our industrial strength prime.

If our candidate failed both tests then we generate a new candidate. If the increment is set to zero, i.e. $inc = 0$ then a new candidate is randomly generated, line 40, if not we add the increment to our current candidate to form the new candidate, line 42

Chapter 8

RSA

> "It has a surprising amount of vigor."
>
> – R. Rivest

8.1 RSA

RSA, named after its inventors, Ronald Rivest, Adi Shamir and Leonard Adleman, was discovered in 1977.

In 1976, Whitfield Diffie and Martin Hellman published their landmark paper, "New Directions in Cryptography"[17]. They proposed a new method for key exchange that would revolutionize secure communication in a world where networks and telecommunication were to be abundantly available. They had found a mechanism to exchange keys where all required information were publicly available, yet the exchanged key were known only to the exchanging parties. The method works only for key exchange, not for actual encryption. In the paper Diffie and Hellman presented the sketch for a public key cryptosystem but left for their colleagues to find the easily computed but difficult to invert trap door function.

In the fall of 1976 the three friends, Ronald, Adi and Leonard, after reading the Diffie-Hellman paper, set out to find an algorithm that satisfied the specification.

Rivest had joined the MIT computer science departement in 1974, Adleman and Shamir joined the MIT mathematics departement in 1975.

Rivest and Shamir were interested in cryptography while Adleman was a number theorist. The three of them had a common interest in computational complexity. One day in November Adleman dropped by Rivest's office. Rivest was reading the Diffie-Hellman paper and enthusiastically explained the key ideas to Adleman. Adleman, being a number theorist, was not impressed. However, both Shamir and Rivest were interested.

The three of them had offices nearby each other. Rivest and Shamir came up with scheme after scheme that they presented to Adleman, who, often after only a few minutes of thinking, would find fatal flaws. One night, around midnight, Rivest called Adleman with a new idea. Adleman immediately realized that it was a good idea. He responded: "Congratulations, Ron, that should work". And it did.

> Find two prime numbers p and q and multiply them, $n = pq$. Find a number $e < (p-1)(q-1)$ and relatively prime to $(p-1)(q-1)$, calculate the multiplicative inverse d modulo $(p-1)(q-1)$ of e. The public key is (n, e) and the private key is (n, d).
>
> To encrypt a message m calculate the ciphertext $c = m^e \pmod{n}$. To decrypt the ciphertext c calculate $m = c^d \pmod{n}$.

Rivest wrote up a paper. When Adleman got it he saw that the authors were listed in the usual alphabetical order. He protested. Adleman thought he had not done enough even to be listed as an author. Rivest disagreed. Adleman agreed to be listed last, he thought the paper would be the least interesting paper he would ever write and that it would appear in an obscure journal, read by none. He was wrong.

The RSA cryptosystem is deceptively simple. RSA is secure because it is difficult to factor large numbers. The public key is (e, n). If we could somehow compute d we could decrypt any intercepted message encrypted with the key (e, n). Since d is the multiplicative inverse of e modulo $\Phi(n)$, we can easily compute d if we know $\Phi(n)$.

Now, $\Phi(n) = (p-1)(q-1)$, so we need to find p and q. We know, by construction of the cryptosystem, that $n = pq$. If we can factor n into it's two prime factors we can compute $\Phi(n)$ and e's multiplicative inverse. It turns out that factoring n is difficult if p and q are large. Factoring is difficult in the sense that factoring large numbers is computationally intensive, i.e. it takes a long time.

The best published algorithm is currently the general number field sieve (GNFS). It has an asymptotic running time of

$$O\big(e^{c(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}}\big)$$

where c is $\left(\frac{64}{9}\right)^{\frac{1}{3}}$ [20].

Definition 33. RSA public and private key.

> Find two prime numbers, p and q. Multiply them to form the modulus, $n = pq$. Select a number $e < \Phi(n)$ and relatively prime to $\Phi(n)$, i.e. $\gcd(e, \Phi(n)) = 1$. Calculate d, the multiplicative inverse of e modulo $\Phi(n)$. The public key is (n, e). The private key is (n, d).

Definition 34. RSA encryption.

> To encrypt a message m, where $0 \le m < n$, compute c, the ciphertext, in
>
> $$c = m^e \pmod{n} \tag{8.1}$$
>
> where e and n are the public key.

Definition 35. RSA decryption.

> To decrypt a ciphertext c, compute m, the message, in
>
> $$m = c^d \pmod{n} \tag{8.2}$$
>
> where d and n are the private key.

We wish to show that encryption and decryption gives the intended result. We have encryption as $c = m^e \pmod{n}$ (equation 8.1) and decryption as $m = c^d \pmod{n}$ (equation 8.2). Combining equation 8.1 and equation 8.2 gives

$$m = c^d = (m^e)^d = m^e d \pmod{n}$$

We will show that equation 8.3 holds for all m, $0 \le m < n$.

$$m = m^{ed} \pmod{n} \tag{8.3}$$

We have, by definition 33, that $n = pq$, where p and q are both prime. We also have that $\gcd(e, \Phi(n)) = 1$ and d is computed so that it is the multiplicative inverse of e mod $\Phi(n)$. So we have

$$ed \equiv 1 \pmod{\Phi(n)}$$

So, by definition 11 we have that there exists a k such that

$$ed - 1 = k\Phi(n)$$

and

$$\Phi(n)|ed - 1$$

We also know that, by theorem 18, if p and q are relatively prime, then

$$\Phi(n) = \Phi(pq) = \Phi(p)\Phi(q)$$

Thus we can write

$$\Phi(p)\Phi(q)|ed - 1$$

also, since if $mn|a$ then $m|a$ and $n|a$ (theorem 2, property 5) we can write

$$\Phi(p)|ed - 1$$
$$\Phi(q)|ed - 1$$

By definition 7 we know that if $\Phi(p)|ed - 1$ then there exists an integer k such that

$$ed - 1 = k\Phi(p)$$

and by theorem 16 we have that

$$ed - 1 = k(p - 1)$$

So, now consider the integer m, $0 \leq m < n$, in

$$m^{ed} \pmod{p}$$

we have

$$m^{ed} = m^{ed-1+1} = m \cdot m^{ed-1} = m \cdot m^{k(p-1)} \pmod{p} \tag{8.4}$$

Now, since p is prime, m is either relatively prime to p or a multiple of p, i.e. either $\gcd(m,p) = 1$ or $\gcd(m,p) = p$. First, if $\gcd(m,p) = 1$ we have, by Fermat's little theorem (theorem 15), that

$$m^{p-1} \equiv 1 \pmod{p}$$

and since we know, by theorem 10, property 3, that if $a \equiv b \pmod{p}$ then $a^k \equiv b^k \pmod{p}$, we can say that

$$m^{k(p-1)} \equiv 1^k \pmod{p} \tag{8.5}$$

Combining equation 8.4 and equation 8.5 we get

$$m^{ed} = m \cdot m^{k(p-1)} = m \pmod{n}$$

Thus, we have, if $\gcd(m,p) = 1$ that

$$m^{ed} \equiv m \pmod{p}$$

Second, if $\gcd(m,p) = p$, i.e. m is a multiple of p, we know that if $p|m$ then $p|m^k$ (theorem 2, property 5) and we have

$$m^{ed} \equiv 0 \pmod{p}$$

and also since $m \equiv 0 \pmod{p}$ we have that

$$m^{ed} \equiv m \pmod{p}$$

Thus we see that, for all m, the desired equation holds, i.e.

$$m^{ed} \equiv m \pmod{p}$$

and by the same reasoning, the same holds for $m^{ed} \equiv m \pmod{q}$.

Since p and q are relatively prime we have by theorem 13 that

$$m^{ed} \equiv m \pmod{pq}$$

and by symmetry (theorem 11, property 2) that

$$m \equiv m^{ed} \pmod{n}$$

Since we require m to be less than n, i.e. $0 \leq m > n$ there is only one integer that satisfies $m \equiv m^{ed} \pmod{n}$.

8.2 RSA Implementation

Key generation for the RSA cryptosystem is straightforward. We generate two primes p and q such that the two highest bits are set. We say that a b bit number is a number greater than 2^{b-1}, i.e. a number where the b:th bit is set. The product of two $\frac{b}{2}$-bit numbers with only the highest bits set is

$$2^{\frac{b}{2}-1} \cdot 2^{\frac{b}{2}-1} = 2^{b-2}$$

which is not enough, so to form n we must multiply two $\frac{b}{2}$-bit primes with (at least) their two highest bits set. Then we get n as

$$n \geq (2^{\frac{b}{2}-1} + 2^{\frac{b}{2}-2})^2 = 2^{b-2} + 2 \cdot 2^{b-3} + 2^{b-4} = 2^{b-1} + 2^{b-4}$$

So, to form a b bit number, n, from two prime factors, p and q, the factors must be greater than $2^{\frac{b}{2}} + 2^{\frac{b}{2}-1}$, i.e. their two highest bits must be set.

We multiply the factors p and q to form n. Then we compute $(p-1)$ and $(q-1)$ and their product, to form $\Phi(n)$ which we use to compute the multiplicative inverse, d, of e by using the extended Euclidean algorithm. Then we return n, e and d as the two pairs (n, e) and (n, d), the public and private key respectively.

Looking at listing 8.1, an implementation of the RSA key generation algorithm, we start by checking for errors in the input. The number of bits requested must be even, they must be a multiple of eight (an octet) and they must be at least 64 (lines 12 to 25).

We then, on line 28 compute the number of digits required to hold a multiple precision number with the requested number of bits. Then we allocate the results, that will be returned, in N, M and D, also we allocate the temporary variables used in the computation.

We need a set of small primes for the `mp_findprime` routine, we allocate space for them on lines 44 to 47. If the memory allocation failed, we generate an error (lines 47 to 49). Then we use trial division (`mp_smallprimes`) to generate the required number of primes in `sp`. Now, we are ready to generate the two primes p and q. We use trial division and the Miller-Rabin test, implemented in `mp_findprime` (lines 54 and 58). We now have everything we need to compute the keys.

```
void
rsa_keygen(rsa_keypair_t *k, mp_t E,
```

```
        unsigned int bits, mp_rand_f rnd)
{
        mp_size_t b;
        mp_limb_t *sp;

        mp_t  P, Q, D, N, M, Phi;
        mp_t   X, Y, A, B, V;
        mp_sign_t sA, sB, sD;

        if (bits & 0x1) {

                MP_ERR(MP_ERR_ARG);
        }

        if (bits < 64) {

                MP_ERR(MP_ERR_ARG);

        }

        if (bits % 8) {

                MP_ERR(MP_ERR_EVENBYTE);
        }

        b = ((bits / 2) + (8 * (sizeof (mp_limb_t) - 1))) /
                (8 * sizeof (mp_limb_t));

        N = mp_new(2 * b);
        D = mp_new(2 * b);
        M = mp_new(2 * b);

        P = mp_tmp(2 * b);
        Q = mp_tmp(2 * b);
        Phi = mp_tmp(2 * b);
        A = mp_tmp(2 * b + 1);
        B = mp_tmp(2 * b + 1);
        V = mp_tmp(2 * b + 1);
        X = mp_tmp(2 * b + 1);
        Y = mp_tmp(2 * b + 1);
```

```
        sp = (mp_limb_t *)malloc(RSA_SMALLPRIMES *
45              sizeof (mp_limb_t));

        if (sp == 0) {

                MP_ERR(MP_ERR_ALLOC);
50      }

        mp_smallprimes(sp, RSA_SMALLPRIMES);

        mp_findprime(P, bits / 2, RSA_MILLER_RABIN_T,
55              sp, RSA_SMALLPRIMES, 0xc0000000, 0x1,
                2, rnd);

        mp_findprime(Q, bits / 2, RSA_MILLER_RABIN_T,
                sp, RSA_SMALLPRIMES, 0xc0000000, 0x1,
60              2, rnd);

        mp_mul(N, P, Q);
        mp_set(M, N);

65      mp_sub(P, P, mp_One);
        mp_sub(Q, Q, mp_One);
        mp_mul(Phi, P, Q);


70      mp_zero(A); sA = MP_POS;
        mp_zero(B); sB = MP_POS;
        mp_zero(V);
        mp_set(X, Phi);
        mp_set(Y, E);
75
        mp_extgcd(A, &sA, B, &sB, V, X, Y);

        if (mp_neq(V, mp_One)) {

80              MP_ERR(MP_ERR_ARG);
        }
```

```
        if (sB == MP_NEG) {
                mp_set(X, Phi);
85              mp_signadd(D, &sD, B, sB, X, MP_POS);
        } else {
                mp_set(D, B);
        }

90      k->k_prv.N = N;
        k->k_prv.D = D;
        k->k_pub.N = M;
        k->k_pub.E = mp_new(*E);
        mp_set(k->k_pub.E, E);
95 }
```

Listing 8.1: RSA key generation

First we compute n, as $n = p \cdot q$ on line 62. We also make a copy of n in m (line 63). Then we compute $(p - 1)$ and $(q - 1)$ on lines 65 and 66. We multiply them to form Φ on line 67.

On lines 70 to 74 we set up the arguments to the extended Euclidean algorithm, which we execute on line 76. The routine `mp_extgcd` will return the greatest common divisor of Φ and e, in v as well as the Bézout coefficients a and b in $ax + by = v$. The coefficient b is the multiplicative inverse to e modulo Φ.

The greatest common divisor of e and Φ must be one, so we check v on line 78. If it is not equal to one we generate an error (line 80). Now, if the Bézout coefficient b is negative we add Φ to b to find the positive d (lines 83 to 87). If b is positive, we just set it to d.

Finally we set up the data of the record used to return the public and private key on lines 86 to 94.

Encryption consists of computing $c = m^e \pmod{n}$. Looking at listing 8.2 we start by checking the input for errors, first, if the size available for the encrypted data is less than the length of the modulus, n, then later decryption will fail so we generate an error (lines 8 to 10). Next, we check so that the message m is less than the modulus n. If not we generate an error since the cryptosystem will fail (lines 13 to 15). Finally, on line 18, the ciphertext, in c, is computed as $c = m^e \pmod{n}$.

```
void
rsa_encrypt(mp_t C, mp_t M, rsa_pubkey_t *k_pub)
{
```

```
        mp_limb_t NL;
5
        NL = mp_len(k_pub->N);

        if (*C < NL) {
10              MP_ERR(MP_ERR_RESSIZEOPLEN);
        }

        if (mp_gt(M, k_pub->N)) {
15              MP_ERR(MP_ERR_OPSIZE);
        }

        mp_expmod(C, M, k_pub->E, k_pub->N);
}
```

Listing 8.2: RSA encryption

Decryption is very similar to encryption. Looking at listing 8.3, first we check for errors. That is, the size available for the decrypted message must be big enough to hold a message of size N_L, the length of the modulus (lines 9 to 11). Then we check so that c is not greater than n, if it is then the cryptosystem will fail (lines 14 to 16). Finally we are ready to decrypt c into the message m, on line 19 we compute $m = c^d \pmod{n}$.

```
1 void
  rsa_decrypt(mp_t M, mp_t C, rsa_prvkey_t *k_prv)
  {

        mp_limb_t NL;
6
        NL = mp_len(k_prv->N);

        if (*M < NL) {
11              MP_ERR(MP_ERR_RESSIZEOPLEN);
        }

        if (mp_gt(C, k_prv->N)) {
```

```
16              MP_ERR(MP_ERR_OPSIZE);
        }

        mp_expmod(M, C, k_prv->D, k_prv->N);
}
```

Listing 8.3: RSA decryption

The two routines `rsa_oi` and `rsa_io` convert a string of octets to a multiple precision integer and vice versa. These routines are both used to translate a message, represented as an octet string into a number that can be encrypted. The encrypted number is then represented as an octet string for transmission. And the reverse process is performed before and after decryption to arrive at the original message.

Looking at listing 8.4 we start by computing the number of octets to process. We convert only the number of octets (n) from the input string t that will fit in the output multiple precision number T (line 8). On lines 11 and 12 we initialize d to point to the first available digit in the output number and c to point to the first octet in the input string.

The main loop starts on line 14, it will be iterated once for each octet to convert. On line 16 we check if the current digit is full. I.e. if there no more room for octets. If this is not the first iteration (iteration zero) and if we are at an even multiple of the number of octets in a digit, then we should go on to the next digit. Which we do on lines 17 and 18, that is, we decrease the pointer d to point to the next more significant digit of T and we zero the digit before filling it with data. On line 21 we convert the octet (pointed to by c) by shifting it into place and inserting it in the digit pointed to by d. Last, we decrease the pointer c to point to the next octet in t.

```
mp_limb_t
rsa_oi(mp_t T, char *t, mp_size_t t_len)
{
        mp_size_t n, i;
5       mp_limb_t *d;
        unsigned char *c;

        n = t_len > *T * sizeof (mp_limb_t) ?
                *T * sizeof (mp_limb_t) : t_len;
10
```

```
        d = T + *T;
        c = t + t_len - 1;

        for (i = 0; i < n; i++) {
15
                if ((0 == (i % sizeof (mp_limb_t))) && i) {
                        d--;
                        *d = 0;
                }
20
                *d |= *c << (8 * (i % sizeof (mp_limb_t)));
                c--;
        }

25      for (i = n; i < *T * sizeof (mp_limb_t); i++) {

                if ((0 == (i % sizeof (mp_limb_t))) && i) {
                        d--;
                        *d = 0;
30              }
        }

        return n;
}
```

Listing 8.4: Octet string to multiple precision integer

After the main loop on lines 14 to 22 what remains is to clear the remaining digits of T, if any. We iterate from n to the number of octets in T (line 25). If i is nonzero and an even multiple of the number of octets in a digit (line 27) we decrease the pointer d to point to the next digit and clear that digit (lines 28 and 29). Finally, we return n, the number of octets that were converted, and we leave the resulting multiple precision number in T.

Looking at listing 8.5, first we get the length of the input multiple precision number T in T_L. Then, on line 12 we find the number of octets in the most significant digit of T. If we closely examine line 12 we see that first we set l to zero and x to the most sinificant digit of T. Then we repeat, while x is nonzero. For each iteration we increase l by one and shift x left one octet (eight bits). Thus, when we exit the loop we will have the bitlength of the most significant digit of T, in octets, in l.

On line 15 we compute the required number of octets to convert T to an octet string. If the length of the provided output string is less than what is required then an error is generated (lines 17 to 19). We check if the length of the output string is longer than the number of octets in the input data (line 22) and limit the number of iterations accordingly. On lines 24 and 25 we set up the pointers d and c to point to the current input digit and current output octet, respectively.

```
1   mp_limb_t
    rsa_io(char *t, mp_size_t t_len, mp_t T)
    {
            mp_size_t n, i;
            mp_size_t TL, L, l;
6           mp_limb_t x;
            mp_limb_t *d;
            char  *c;

            TL = mp_len(T);
11
            for (l = 0, x = *(T + 1 + *T - TL); x; l++, x >>= 8)
                    ;;

            L = (TL - 1) * sizeof (mp_limb_t) + l;
16
            if (L > t_len) {

                    MP_ERR(MP_ERR_RESSIZEOPLEN);
            }
21
            n = t_len > L ? L : t_len;

            d = T + *T;
            c = t + n - 1;
26
            for (i = 0; i < n; i++) {

                    if (0 == (i % sizeof (mp_limb_t)) && i) {
                            d--;
31                  }
```

```
                *c = (*d >> 8 *
                      (i % sizeof (mp_limb_t))) & 0xff;
                c--;
36          }

        return n;
}
```

Listing 8.5: Multiple precision integer to octet string

Then we enter the main loop, lines 27 to 35. First, if this is not the first iteration (iteration zero) and if we are at an even multiple of the number of octets in a digit, then we go on to process the next digit (lines 29 and 30). On line 33 we extract the current octet from the current digit and assign it to $*c$. Finally, we step the pointer c and we are done with this iteration.

When all iterations are executed, we have the octet string in t and the number of converted octets is returned in n.

Bibliography

[1] Paranjape, K. P., Some lectures on number theory, elliptic curves and cryptology.

[2] Brinch Hansen, P., Multiple-length division revisited: a tour of the minefield, Software – practice and experience 24, June 1994, pp. 579–601, 1994, John Wiley Sons, Ltd.

[3] Riemann, B., On the Number of Prime Numbers less than a Given Quantity (Über die Anzahl der Primzahlen under einer gegebenen Grösse), Monatsberichte der Berliner Akademie, Nov. 1859, http://www.maths.tcd.ie/pub/HistMath/People/Riemann/Zeta/EZeta.pdf

[4] Haldir, How to crack a Linear Congruential Generator, source unknown, http://www.reteam.org/papers/e59.pdf

[5] Park, Stephen, K., Miller, Keith, W., Random number generators: good ones are hard to find, Communications of the ACM, Oct 1988, Vol 31, No 10.

[6] Marsaglia, G., Random numbers fall mainly in the planes, Proceedings of the National Academy of Sciences of the United States of America, June 1968, Vol 61, Issue 1, pp. 25-28.

[7] Goldwasser, S., Bellare, M., Lecture Notes on Cryptography, June 2008, http://www-cse.ucsd.edu/~mihir/papers/gb.pdf

[8] Eastlake, D., Schiller, J., Crocker, S., Randomness Requirements for Security, June 2005, IETF, RFC 4086,
http://www.ietf.org/rfc/rfc4086.txt

[9] Eastlake, D., Jones, P., US Secure Hash Algorithm 1 (SHA1), September 2001, IETF, RFC 3174, http://www.ietf.org/rfc/rfc3174.txt

[10] Secure Hash Standard (SHS), October 2008, National Institute of Standards and Technology, Information Technology Laboratory, FIPS 180-3, http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf

[11] Barker, E., Kelsey, J., Recommendations for Random Number Generation Using Deterministic Random Bit Generators (Revisited), March 2007, National Institute of Standards and Technology, Information Technology Laboratory, NIST SP 800-90, http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf

[12] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction set reference N-Z, March 2009, Intel.

[13] Geisler, M., Krøigård, M., Danielsen, A., About Random Bits, December 2004, http://www.cecm.sfu.ca/~monaganm/teaching/CryptographyF08/random-bits.pdf

[14] Davis, D., Ihaka, R., Fenstermacher, P., Cryptographic Randomness from Air Turbulence in Disk Drives, 1994, Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology, Springer-Verlag, http://world.std.com/~dtd/random/forward.ps

[15] Jakobsson, M., Shriver, E., Hillyer, B. K., Juels, A., A Practical Secure Physical Random Bit Generator, 1998, Proceedings of the 5th ACM Conference on Computer and Communications Security, ACM Press, http://www.rsa.com/rsalabs/staff/bios/ajuels/publications/ physicalrand/physicalrand.ps

[16] Junod, P., Cryptographic Secure Random Number Generation: The Blum-Blum-Shub Generator, August 1999, http://crypto.junod.info/bbs.pdf

[17] Diffie, W., Hellman, M. E., New Directions in Cryptography, Nov 1976, IEEE Transactions on Information Theory, vol. IT-22, pp. 644-654.

[18] Rivest, R. L., Shamir, A., Adleman, L., On Digital Signatures and Public Key Cryptosystems, Feb. 1978, Communications of the ACM, vol. 21, pp. 120-126.

[19] Toms hardware, July 2009, http://www.tomshardware.com/charts/2009-desktop-cpu-charts/SiSoftware-Sandra-2009-SP3-Processor-Arithmetic,1389.html

[20] Pomerance, C., A Tale of Two Sieves, December 1996, Notices of the AMS, http://www.ams.org/notices/199612/pomerance.pdf

Index

accepting states, 25
addition, 40
    algorithm, 42
    digit position n, 75
    implementation, 44
    implementation, primitive, 43
    long, 41
    modulo, 13, 15
    multiple precision, 43
    primitive, 40, 42
    signed, 101
    signed, implementation, 101
address, 39
Adleman, Leonard, 167
AES, 139
air turbulence, 140
algorith
    trial division, 153
algorithm, 28
    addition, 42
    binary extended gcd, 111
    binary gcd, 107
    Blum-Blum-Shub generator, 143
    division, 7, 68
    Euclid's, 106
    extended gcd, 110
    finding small primes, 151
    greatest common divisor, 106
    Miller-Rabin primality test, 159
    MINSTD, 130
    modular exponentiation, 118
    multiplication, 57
    RSA decryption, 176
    RSA encryption, 175
    RSA key generation, 172
    sieve of Eratosthenes, 150
    subtraction, 49
algorithmic randomness, 132
alphabet, 23
array
    notation, 39
asymptotic notation, 28

Bézout coefficients, 8
Bézout numbers, 8
Bézout's identity, 8
Bézout, Étienne, 8
base, 33, 38
base change, 35
bias, 137
biased, 127

big O notation, 28
binary, 37
binary digit, 37
binary extended cd
    algorithm, 111
binary gcd
    algorithm, 107
    implementation, 108
binary modular exponentiation, 118
bit, 37
    clear, 97
    clear, implementation, 98
    count, implementation, 100
    counting, 97
    get, 97
    get, implementation, 98
    highest, 97
    set, 97
    set, implementation, 99
bits, 134
Blum-Blum-Shub generator
    algorithm, 143
    implementation, generation, 146
    implementation, initialization, 144
    implementation, seeding, 145
    pseudo random number generator, 143

carry, 41
cell, 4
    memory, 39
cells, 24
certificate, 30
Chebyshev, Pafnuty Lvovich, 155
ciphertext, 168
class
    NP, 30
    NP-complete, 30
    NPC, 30, 31
    P, 30
    polynomial, 29
clear bit, 97
Cobham's thesis, 30
coin flip, 135
comparison, 89
    implementation, 90
complexity
    addition, 42
    division, 69
    multiplication, 57
    space, 28
    subtraction, 49
    time, 28
complexity class, 29, 30
composite, 9, 149
compression, 138
computer, 26, 37
    limitations, 37
congruence, 13
conversion, 177
coprime, 9
counting bits, 97
cryptographically secure, 143

datatype, 39
de-skew, 136
    compression, 138
    discard pairs, 138
    exclusive-or, 139
    parity, 138
decision problem, 28
decrypt, 168
decryption, 169, 176
dereference, 40
derivable, 24
DES, 139
deterministic Turing machine, 29

Diffie, Whitfield, 167
discrete probability distribution, 133
distribution
  discrete probability, 133
divides, 5
divisibility
  divisor, 5
  properties, 6
division, 63
  addition, digit position n, 75, 76
  algorithm, 68
  implementation, 82
  initialization, 81
  long, 65
  main loop, 86
  main routine, 81
  multiple precision, 74, 81
  multiply and subtract, 78
  normalization, 74
  primitive, 63, 69
  primitive, implementation, 72
division algorithm, 7
divisor, 5
  common, 6
  greatest common, 6
  quotient, 8

elementary arithmetic, 37
Elements, 9
elements, 3
empty set, 4
encrypt, 168
encryption, 169, 175
entropy, 133
  definition, 133
  harvesting, 136, 139
  minimum, 136
  mix, 138
  rate, 135
  sources, 140
entropy source, 138
equivalence class, 5
equivalence relation, 5
Erathosthenes, 150
error handling, 123
  declarations, 123
  error message, 125
  generating error, 124
  handler, 125
estimate
  quotient, 68
Euclid, 9
  first theorem, 10
  lemma, 10
  second theorem, 10
Euclid's algorithm, 106
Euclid's lemma, 10
Euler totient, 20
Euler's theorem, 21
Euler, Leonhard Paul, 18
even, 92
  implementation, 92
exclusive-or, 139
expansion, 33
extended gcd
  algorithm, 110

factor, 5
factoring, 169
factors
  prime, 149
Felkel, Anton, 155
Fermat's little theorem, 17, 158
Fermat, Pierre de, 15
final states, 25
finding large primes, 163
  implementation, 164

Fundamental theorem of arithmetic, 11

Gauss, Carl Friedrich, 11, 12, 155
general number field sieve, 169
generating large primes, 163
  implementation, 164
get bit, 97
global randomness, 128
grammar, 24
greatest common divisor, 6, 105
  algorithm, 106
  binary extended gcd, 111
  binary gcd, 107, 110
  implementation, 108, 112
growth rate, 29
guess, 86

h(x), 38
Hadamard, Jacques Salomon, 155
halfword, 38
harvesting entropy, 139
head, 24
Hellman, Martin, 167
highest bit, 97

immediately derivable, 24
implementation, 37
  addition, 44
  addition, digit position n, 76
  addition, signed, 101
  binary extended gcd, 112
  binary gcd, 108
  Blum-Blum-Shub generator, generation, 146
  Blum-Blum-Shub generator, initialization, 144
  Blum-Blum-Shub generator, seeding, 145
  clear bit, 98
  comparison, 90
  count bits, 100
  division, 82
  error, declarations, 123
  error, generating error, 124
  error, handler, 125
  error, message, 125
  Euclid's extended algorithm, 112
  even, 92
  finding large primes, 164
  finding small primes, 151
  get bit, 98
  greatest common divisor, 108
  initialization, 126
  integer to octet conversion, 177
  left shift, 94
  length, 93
  Miller-Rabin primality test, 160
  MINSTD, 131
  modular exponentiation, 118
  multiplication, 61
  multiply and subtract, 79
  normalization, 74
  octet to integer conversion, 177
  odd, 92
  primitive addition, 43
  primitive division, 72
  primitive multiplication, 59
  primitive subtraction, 50
  RSA decryption, 176
  RSA encryption, 175
  RSA key generation, 172

- scratch memory, allocating, 122
- scratch memory, declarations, 121
- scratch memory, initialization, 121
- scratch memory, marking, 122
- scratch memory, reclaiming, 123
- set bit, 99
- shift right, 96
- subtraction, 51
- subtraction, signed, 103
- temporary memory, allocating, 122
- temporary memory, declarations, 121
- temporary memory, initialization, 121
- temporary memory, marking, 122
- temporary memory, reclaiming, 123
- trial division, 151, 153

indistinguishable, 141
infinitude of primes, 10
information, 133
information content, 133, 134
initialization, 125
- implementation, 126

instructions, 25
integer, 5
integer to octet conversion, 177
integers, 5
intel, 140

key exchange, 167
key generation, 172
keystrokes, 140
Kleene star, 23
Kolmogorov complexity, 133
Kolmogorov randomness, 132

l(x), 38
language, 24
large primes, finding, 163
least significant, 38, 39
left shift, 94
Legendre, Adrien-Marie, 155
length, 39, 92
- implementation, 93

liars, 157
limitations, 37
linear congruential random number generator, 128
local randomness, 128
long
- addition, 41
- division, 65
- multiplication, 54
- subtraction, 48

lucky guess, 29

maximum digit, 38
MD*, 139
Miller-Rabin primality test, 159, 163, 165
- algorithm, 159
- implementation, 160

minimal standard, 129
minimum entropy, 136
MINSTD, 129
- algorithm, 130
- implementation, 131

mix, 138
mixing function, 138
- encryption, 139
- hash, 139
- strong, 139

modular
  addition, 15
  cancellation, 15
  congruence, 13
  exponentiation, 117
  multiplication, 15
  subtraction, 15
modular arithmetic, 12
modular exponentiation, 117
  algorithm, 118
  implementation, 118
modulo
  addition, 13
  equivalence relation, 14
  exponentiation, 117
  multiplication, 14
  subtraction, 13
modulus, 169
most significant, 38
multiple precision, 37
  addition, 40, 43, 44
  addition, signed, 101
  comparison, 90
  division, 63, 74, 81, 82
  equality, 91
  even, 92
  left shift, 94
  multiplication, 53, 60, 61
  odd, 92
  right shift, 96
  subtraction, 47, 50, 51
  subtraction, signed, 101, 103
multiple precision arithmetic, 39
multiplication, 53
  algorithm, 57
  implementation, 61
  long, 54
  moduli, 15
  modulo, 14
  multiple precision, 60
  primitive, 53, 58
  primitive, implementation, 59
multiply and subtract, 78
  index, 79

negative number, 35
next bit test, 142
non-deterministic Turing machine, 29
non-terminal, 24
normalization, 74
  implementation, 74
normalized, 66, 70
NP, 30
NP-complete, 31
NPC, 30, 31
number, 33
number system, 33
numeral, 33

objective unpredicability, 128
octet to integer conversion, 177
odd, 92
  implementation, 92
overflow, 37

P, 30
parity, 136
partition, 4
partitioning relation, 4
period, 128
pointer, 39
  dereference, 40
polynomial time, 29
polynomial time indistinguishable, 141
positional numeral system, 33
Poussin, Charles-Jean Étienne Gustave Nicolas, 155

primality test, 149, 150
  Miller-Rabin, 159
  probabalistic, 157, 159
prime, 9, 149
  candidate, 164
  composite, 9
  coprime, 9
  distribution, asymptotic law, 155
  Euler's theorem, 21
  factors, 149
  Fermat's little theorem, 17
  finding, 149
  finding, algorithm, 151
  finding, implementation, 151
  finding, large, 163
  generating, large, 163
  probabalistic test, 157, 159
  probability, 163
  probable, 157, 165
  relatively, 9
  test, 149, 150, 157
prime number theorem, 155, 163
primitive
  addition, 40, 42
  addition, implementation, 43
  division, 63, 69
  division, implementation, 72
  multiplication, 53, 58
  multiplication, implementation, 59
  subtraction, 47, 49
  subtraction, implementation, 50
primitive datatype, 37
primitive operation, 37
primitive operations, 33
private key, 169
probabalistic Turing machine, 141
probability
  discrete distribution, 133
probable prime, 157, 165
productions, 24
pseudo random, 142
pseudo random generator, 142
public key, 169
public key cryptosystem, 167

quotient, 63
  estimate, 66, 68, 70
quotient digit, 66

random, 127
  air turbulence, 140
  algorithmic, 132
  Blum-Blum-Shub generator, 143
  global, 128
  Kolmogorov, 132
  linear congruential generator, 128
  local, 128
  MINSTD, 129
  number generator, 140
  period, 128
  pseudo, 141, 142
  pseudo generator, 142
  seed, 142
  statistically, 128
  test, 131
  true, 128, 140
  truly, 128
random number, 165
randomly, 127
randomness, 127, 136
reducible, 31
reduction, 31
reflexive, 4, 14
relation

equivalence, 5
modulo, 14
partitioning, 4
relatively prime, 9, 21
remainder, 63
representation, 33
multiple precision integer, 39
rewriting system, 24
Riemann, Georg Friedrich Bernhard, 156
right shift, 95
Rivest, Ronald, 167
RSA, 167
decrypt, 168
decryption, 169, 176
encrypt, 168
encryption, 169, 175
implementation, 172
key generation, 172

scratch memory
allocating, 122
declarations, 121
initialization, 121
marking, 122
reclaiming, 123
scratch space, 120
seed, 142
self-information, 134
set, 3
empty, 4
partition, 4
set bit, 97
SHA*, 139
Shamir, Adi, 167
Shannon, Claude Elwood, 134
shift, 93, 95
left, 94
right, 95, 96
sieve of Eratosthenes, 150
algorithm, 150
size, 39
smallest, 5
space complexity, 28
square roots, 157
start state, 25
starting symbol, 24
state, 25
accepting, 25
final, 25
start, 25
state register, 25
statistically random, 128
subset, 4
subtraction, 47
algorithm, 49
implementation, 51
long, 48
modulo, 13, 15
multiple precision, 50
primitive, 47, 49
primitive, implementation, 50
signed, 101
signed, implementation, 103
surprisal, 135
symmetric, 4, 14

table, 25
tape, 24
temperature, 140
temporary memory, 120
allocating, 122
declarations, 121
initialization, 121
marking, 122
reclaiming, 123
terminals, 24
theorem

Bézout's identity, 8
base change, 35
divisibility, 6
division algorithm, 7
equivalence relation, 14
Euclid's lemma, 10
Euler's theorem, 21
Fermat's little, 158
Fermat's little theorem, 17
fundamental theorem of arithmetic, 11
infinitude of primes, 10
modular addition, 13, 15
modular cancellation, 15
modular multiplication, 14
modular subtraction, 13, 15
multiplying moduli, 15
partitioning relation, 4
prime number, 155, 163
square roots of unity, 157
totient evaluation, 20
unique representation, 33
time complexity, 28
timers
high resolution, 140
totient, 20
totient evaluation, 20
transition function, 25
transitive, 4, 14
trial division, 149, 164, 165
algorithm, 153
implementation, 151, 153
truly random, 128
truncation, 38
Turing machine, 24
deterministic, 29
example, 26
non-deterministic, 29
probabalistic, 141
universal, 132
Turing, Alan Mathison, 26

unbiased, 127
uncertainty, 134–136
underflow, 37
unique representation, 33
universal Turing machine, 132
unsigned number, 37

Vega, Baron Jurij Bartolomej, 155
verifier, 30

well defined, 4
well ordering principle, 5
witness, 157, 159
word, 24
wordsize, 37
maximum, 38
wrap, 37
wraparound, 37

Z, 5